**Universidad de Santiago de Chile**

**Facultad de Ingeniería**

**Departamento de Ingeniería Informática**

**Paradigmas de Programación**

**Paradigmas de Programación**

**Proyecto semestral de laboratorio**

Versión Preliminar - actualizada al 19-08-2019

Durante el semestre se trabajarán distintos paradigmas de programación, los cuales serán abordados mediante cuatro lenguajes de programación en cuatro entregas de laboratorio. Para el desarrollo satisfactorio del laboratorio del curso se requerirá desarrollar un videojuego en 2D.

El videojuego en su entrega final (Laboratorio N°4) poseerá gráficos en dos dimensiones similar a los de juegos como pac-man, rodents revenge, space invaders y bomberman (ver Figura 1).

|  |
| --- |
| rodent1.PNGbomberman3.pnghqdefault.jpg |

*Figura 1. Capturas de pantalla de los videojuegos Rodents Revenge, Bomberman y Battleship.*

# 

# Descripción general

Varios aspectos del videojuego quedarán abiertos a la creatividad de cada estudiante, sin embargo se debe cumplir con requisitos mínimos los cuales se especifican en las siguientes secciones.

Los primeros tres laboratorios tendrán interfaces claramente definidas (esto es, formatos de lectura, cabeceras de funciones, etc). **El no respetar estas interfaces/prototipos implicarán la obtención de la nota mínima y por consecuencia la reprobación del laboratorio.**

Para este semestre se ha definido un juego al estilo de **Angry Birds y Worms**. Al final del semestre los alumnos de Paradigmas de Programación habrán implementado distintas versiones de este juego en distintos lenguajes de programación, según los paradigmas que los rigen. A continuación se listan de forma general características del juego que serán abordadas de forma obligatoria o complementaria durante el semestre en los distintos proyectos. Los detalles para cada laboratorio se listan posteriormente.

* Tema del juego: Juego al estilo de Angry Birds y Worms en una versión muy simplificada (ver Figura 2 y video de referencia en

|  |  |
| --- | --- |
|  |  |
| <https://www.youtube.com/watch?v=wmcCZmItJrQ> | <https://www.youtube.com/watch?v=S6lrRqst9Z4> |

*Figura 2. Captura de pantalla de ejemplo de Angry Birds y Worms.*

* El escenario del juego debe ser de tamaño NxM a partir de los recursos ofrecidos por los lenguajes de programación y paradigmas subyacentes.
* Se trata de un juego por turnos en modalidad Jugador vs CPU. Jugador y CPU tienen turnos intercalados (1 a la vez) pudiendo realizar una única acción en cada turno, el cual para la versión final del juego tiene una duración máxima.
* En esta versión del juego el jugador controlará a un grupo de *P* personajes que se ubican en el extremo izquierdo de la pantalla, mientras que la CPU controla a un grupo de *E* enemigos dispuestos en el extremo derecho de la pantalla.
* El jugador puede controlar a sus personajes para desplazarlos y disparar proyectiles hacia los enemigos a partir de un ángulo específico.
* En el espacio pueden haber objetos que obstaculicen la trayectoria de proyectiles (árboles, rocas, etc.) o que sirvan de plataforma para sostener a los personajes controlados por jugador o CPU.
* El juego culmina cuando uno de los dos equipos (Jugador o CPU) eliminan completamente al equipo contrario.
* Las variantes que se puedan implementar en el juego tales como variedad de proyectiles, armamento especial, extra turnos, inteligencia de enemigos, etc. se podrán implementar a través de requisitos extras según las indicaciones señaladas en cada enunciado.
* La física que gobierna el movimientos de los proyectiles debe ser “realista”, esto es, que se calcule adecuadamente/coherentemente velocidad, dirección, ángulos, etc (movimiento parabólico).
* Las reglas anteriormente descritas corresponden a lo mínimo que deben lograr en todos los laboratorios, el resto de requisitos son dependientes del laboratorio en específico.

**Comodines:** Durante el semestre cada alumno dispondrá de **CUATRO comodines (uno para cada laboratorio)** que podrá utilizar **solo en el siguiente caso**:

* **Si su calificación** en los laboratorios 1, 2, 3 y parcialmente en el 4 (Ya sea por requerimientos funcionales, no funcionales, informe) **es mayor que 2.0 e inferior a 4.0**.

El comodín permitirá al alumno entregar lo necesario (requerimientos funcionales, no funcionales, informe de usuario) para alcanzar la nota mínima requerida (4.0) en la correspondiente entrega de laboratorio para la aprobación al final del semestre.

Antes de proceder al uso de comodines, y si lo amerita, se recomienda a los alumnos que empleen las instancias de recorrecciones de laboratorio.

En caso de requerir el uso de comodín, podrá hacer entrega del correspondiente laboratorio corregido al final del semestre. La fecha específica y espacio de entrega serán informados por los profesores (toda entrega se realiza a través de UsachVirtual).

Finalmente, por temas de tiempo considerando el avance del semestre, en el laboratorio final se podrá hacer un uso limitado de comodines (en caso de disponer de estos). Esto quedará a disposición de los profesores correctores dependiendo del nivel de los elementos que son calificados como insuficientes y la factibilidad de poder completarlos en los plazos para el cierre oficial del semestre dentro de las semanas lectivas 16~17.

# Laboratorio 1 (Paradigma Funcional - Lenguaje Scheme)

Versión 1.0

(Cambios menores pueden incorporarse en futuras versiones a fin de aclarar o corregir errores)

**Fecha de Entrega:** 30 de Septiembre de 2019, hasta las 23:55 a través de Usach Virtual

**Entregables:** Archivo ZIP con el siguiente nombre de archivo: lab1\_rut\_ApellidoPaterno.zip (ej: lab1\_12123456\_Perez.zip). Notar que no incluye dígito verificador. Dentro del archivo se debe incluir:

1. Informe en formato Word o PDF.
2. Carpeta con código fuente.
3. Archivo leeme.txt para cualquier instrucción especial para ejecución u otro, según aplique.
4. Autoevaluación.txt donde debe incluir una lista completa de todos los requisitos funcionales y no funcionales señalando una evaluación para cada uno según la siguiente escala:
   1. 0: No realizado.
   2. 0.25: Implementación con problemas mayores (funciona 25% de las veces o no funciona)
   3. 0.5: Implementación con funcionamiento irregular (funciona 50% de las veces)
   4. 0.75: Implementación con problemas menores (funciona 75% de las veces)
   5. 1: Implementación completa sin problemas (funciona 100% de las veces)
5. Archivo repositorio.txt que incluya la URL de su repositorio en GitHub el que deberá además ser compartido con los profesores y ayudantes al momento de la evaluación.

El nombre del archivo con el código fuente debe seguir el siguiente formato: NombreArchivo\_rut\_Apellidos.extension (ej: tetris\_12123456\_PerezPeña.rkt)

**Objetivo del laboratorio:** Aplicar y demostrar los conocimientos adquiridos en cátedra con respecto al paradigma de programación funcional usando el lenguaje de programación Scheme.

**Resultado esperado:** Programa para la generación de tableros, validación de tableros, ejecución de partidas y evaluación de partidas.

**Informe del proyecto:** Extensión no debe superar las **15 planas** de contenido (se excluyen: portada, índice, referencias). El informe debe incluir introducción, descripción breve del problema, análisis del problema, diseño de la solución (esquematizada y explicada brevemente), consideraciones de implementación (ej: algoritmos de cálculos de trayectoria, algoritmos especiales que haya implementado, organización del código), **instrucciones de uso con ejemplos claros de uso**, resultados obtenidos, evaluación completa y conclusiones.

**Repositorio en Git del Proyecto:** El laboratorio deberá ser desarrollado incluyendo el sistema de control de versiones GIt y utilizando GitHub como servidor remoto. El repositorio debe ser privado durante la elaboración del proyecto. Cualquier inconsistencia entre lo entregado en Usach Virtual y la versión final del repositorio será evaluado con nota mínima.

**Evaluación:** El Informe (Inf), requerimientos funcionales (RF), requerimientos no funcionales (RNF), ejecución (Eje) se evalúan por separado. La nota final de este laboratorio (NL) se calcula de la siguiente forma considerando sólo la calificación base a partir del cumplimiento de los RF y RNF obligatorios.

**Requerimientos No Funcionales obligatorios. Nota que algunos son ineludibles, esto quiere decir que al no cumplir con dicho requerimiento, su proyecto será evaluado con la nota mínima.**

1. **(obligatorio)** Incluir autoevaluación de cada uno de los requerimientos funcionales (obligatorios y extras) solicitados.
2. **(obligatorio)** La implementación debe ser en el lenguaje de programación Scheme.
3. **(obligatorio)** Usar DrRacket versión 6.11
4. **(obligatorio)** Se deben utilizar funciones estándar de dicho lenguaje y no aquellas propias Racket. Regirse por standard R6RS (esto último es opcional si desea integrar con C#).
5. **(obligatorio)** No hacer uso de función “define” dentro del cuerpo de otras funciones, de set! o funciones similares para emular el trabajo con variables.
6. **(1 pts)** Todas las funciones deben estar debidamente comentadas. Indicando descripción de la función, argumentos y retornos. En caso de que la función sea recursiva, indicar el tipo de recursión utilizada y el porqué de esta decisión.
7. **(obligatorio)** Respetar la definición de función en términos de conjunto de salida (dominio) y llegada (recorrido) sin efectos colaterales.
8. **(1 pts)** Estructurar su código en archivos independientes. Un archivo para cada TDA implementado y uno para el programa principal donde solo se dispongan las funciones requeridas en el apartado de requerimientos obligatorios.
9. **(2.5 pts)** Historial de trabajo en Github tomando en consideración la evolución en el desarrollo de su proyecto en distintas etapas. Se requieren **al menos 10 commits** distribuidos en un periodo de tiempo **mayor o igual a 2 semanas**. A modo de ejemplo, si hace todos los commits el día antes de la entrega del proyecto, este ítem tendrá 0 pts. De manera similar, si hace dos commits dos semanas antes de la entrega final y el resto los concentra en los últimos dos días, tendrá una evaluación del 25% para este item (0.375 pts).Por el contrario si demuestra constancia en los commits (con aportes claros entre uno y otro) a lo largo del periodo evaluado, este ítem será evaluado con el total del puntaje.
10. **(obligatorio)** Al final de su código incluir al menos 3 ejemplos de uso para cada una de las funciones correspondientes a requerimientos funcionales obligatorios y los extra. **Solo se revisarán aquellas funciones para las que existan los ejemplos provistos.**

**Requerimientos Funcionales Obligatorios. Para que el requisito sea evaluado, DEBE cumplir con el requisito base señalado, en caso contrario la función no será evaluada.**

1. **(0.4 pts)**. Implementar abstracciones apropiadas para el problema. Para la implementación debe regirse por estructura de implementación de TDA vista en clases: Representación, Constructores, Funciones de Pertenencia, Selectores, Modificadores y Otras Funciones. Procurar hacer un uso adecuado de esta estructura a fin de no afectar la eficiencia de sus funciones. En el resto de las funciones se debe hacer un uso adecuado de la implementación del TDA. Solo implementar las funciones necesarias dentro de esta estructura. Dejar claramente documentado con comentarios en el código qué corresponde a la estructura base del TDA. La estructura del TDA debe considerar la existencia de un estado del juego (i.e., PLAYING, VICTORY, DEFEAT, DRAW), nivel de dificultad, puntaje, parámetros de enemigos, obstáculos y cualquier otro recurso que se incluya en el juego.

***Requisito base: Especificar representación de manera clara para cada TDA implementado. Implementar constructores, funciones de pertenencia, y según se requiera selectores, modificadores y otras funciones que pueda requerir para las otras funciones.***

1. **(0.4 pts)** createScene: Función que retorna un escenario válido de tamaño NxM posicionando la cantidad de enemigos iniciales (parámetro ‘E’). Se debe usar recursión lineal (RL) en esta función o en alguna de las complementarias. El parámetro seed se refiere a la semilla usada para generar valores pseudoaleatorios (ver Nota 2). El jugador siempre parte al lado izquierdo del escenario en estado estacionario con orientación en . El parámetro ‘D’ representa la dificultad del escenario, lo que afecta por ejemplo la cantidad y posición de los obstáculos en el escenario.

***Requisito base: Debe emplear recursión natural o de cola.***

*(createScene N M E D seed)*

**Nota 1:** Un escenario válido debe poder ubicar la cantidad de enemigos especificados en el parámetro ‘E’. La factibilidad de poder disponer todos los enemigos en el escenario dependerá del tamaño de este. Si no es factible crear el escenario, se debe retornar nulo.

**Nota 2:** El parámetro seed se refiere a la semilla para generar valores pseudoaleatorios (ver ejemplo provisto en UsachVirtual para su referencia de cómo generar estos números sin violar aspectos del paradigma funcional. Pueden usar esta función y/o adaptarla).

Los parámetros N, M, E, D y seed son números enteros.

1. **(0.2 pts)** checkScene: Función que recibe un escenario de juego (scene) y permite verificar si éste cumple con los criterios para ser considerado como un escenario válido (Ej: Que es realmente un espacio con dimensiones válidas, enemigos, obstáculos y jugador válidamente ubicados dentro de este, esto según su representación debe incluir validaciones que eviten la superposición de elementos en el escenario (Ej: Que las coordenadas de dos enemigos sean las mismas). El retorno de la función es de tipo booleano. Verdadero en caso de que se trate de un espacio válido y falso en el caso contrario.

***Requisito base: Debe emplear recursión natural o de cola, siendo el estilo de recursión empleado distinto al usado en la función createScene.***

*(checkScene scene)*

1. **(1.5 pts)** play: Función que permite disparar proyectil en el turno del jugador tomando en consideración la ejecución de las siguientes acciones en orden: desplazamiento del personaje (member) que se desea operar en el turno hacia la izquierda o derecha (*move*), especificación del ángulo de disparo de proyectil (*angle*) y la función de trayectoria (tf), actualización de la escena, y la ejecución del turno del enemigo. El retorno de la función es un nuevo escenario con el resultado tras ejecutar el disparo del proyectil, la correspondiente respuesta del equipo enemigo (en caso de seguir operativo luego de la acción del jugador) y la actualización del estado del juego (ej: VICTORY, DEFEAT, DRAW).

En relación a los parámetros:

* angle: valor entre -90° 0 90°, sin embargo, estos valores dependerán además de la inclinación que tenga cada miembro del equipo de acuerdo a su ubicación en el terreno. Por ejemplo, si se mueve -90° y se encuentra en un plano empinado en 20°, entonces el movimiento real será en -70° en el escenario para así seguir sobre la superficie en que se encontraba el jugador.
* move: indica la cantidad de pasos (valores enteros positivo para desplazamientos a la derecha y negativos para desplazamientos a la izquierda) que dará el miembro del equipo que será usado en el turno a la derecha o a la izquierda. Considerar posibles caídas a otras regiones del terreno o incluso fuera de la escena lo que significa la pérdida del miembro del equipo y por consiguiente la pérdida del turno (incluso la pérdida de la partida si se trataba del último miembro activo).
* member: corresponde a número entero que señala el personaje del equipo se quiere controlar en el turno.
* tf: corresponde a una función que definirá la trayectoria y propiedades físicas especiales que tendrá el proyectil (ej: teledirigido, aceleración rápida, movimiento parabólica, etc.). Está función otorga versatilidad a su juego pudiendo experimentar con distintas funciones.

***Requisito base: Implementar función de manera currificada y uso de función de orden superior.***

*(play scene member move tf angle seed)*

*Para aclarar mas este punto: La función play se llamará así:*

*(Se(((((play scene) member) move) tf) angle) seed)*

1. **(1.5 pts)** playLazy: Función que permite disparar proyectil en el turno del jugador tomando en consideración la ejecución de las siguientes acciones en orden: desplazamiento del personaje (member) que se desea operar en el turno hacia la izquierda o derecha (*move*), especificación del ángulo de disparo de proyectil (*angle*) y la función de trayectoria (tf), actualización de la escena, y la ejecución del turno del enemigo. El retorno de la función es una lista infinita donde cada elemento de la lista es una actualización parcial de la escena tras el disparo del proyectil por parte del jugador o del enemigo, pudiendo ver la posición del proyectil de acuerdo a los parámetros de la función de trayectoria tf transcurrido un tiempo t. Cabe señalar que cada elemento de la lista indicará el estado del juego (ej: VICTORY, DEFEAT, DRAW). El resultado final de la función playLazy debe ser equivalente al de la función play. Esto se refiere al último elemento generado dependiendo de la frecuencia de muestreo específicada para a través de t (para un t pequeño se tendrán muchos elementos en la lista infinita antes de alcanzar el resultado final relativo al término de una jugada, mientras que para un t muy grande, incluso el primer elemento podría corresponder al resultado final de una jugada)

En relación a los parámetros:

* angle: valor entre -90° 0 90°, sin embargo, estos valores dependeran además de la inclinación que tenga cada miembro del equipo de acuerdo a su ubicación en el terreno. Por ejemplo, si se mueve -90° y se encuentra en un plano empinado en 20°, entonces el movimiento real será en -70° en el escenario para así seguir sobre la superficie en que se encontraba el jugador.
* move: indica la cantidad de pasos (valores enteros positivo para desplazamientos a la derecha y negativos para desplazamientos a la izquierda) que dará el miembro del equipo que será usado en el turno a la derecha o a la izquierda. Considerar posibles caídas a otras regiones del terreno o incluso fuera de la escena lo que significa la pérdida del miembro del equipo y por consiguiente la pérdida del turno (incluso la pérdida de la partida si se trataba del último miembro activo).
* member: corresponde a número entero que señala el personaje del equipo se quiere controlar en el turno.
* tf: corresponde a una función que definirá la trayectoria y propiedades físicas especiales que tendrá el proyectil (ej: teledirigido, aceleración rápida, movimiento parabólica, etc.). Está función otorga versatilidad a su juego pudiendo experimentar con distintas funciones.
* t: corresponde al tiempo sobre el cual se realizará el cálculo de la posición del proyectil en base a la función tf (los elementos son generados hasta tiempo t) en cada instante de tiempo. Se le puede considerar como frecuencia de muestreo. Esto corresponde a la tasa de refresco en que se realizará el cálculo de cada elemento de la lista. Por ejemplo en una mirada de cuadros por segundo t=1 quiere que podremos ver trayectoria del proyectil en cada cuadro. Por otro lado desde la mirada del tiempo en segundos, t=5s quiere decir que podremos ver en cada elemento de la lista un escenario con el proyectil en tiempo 0, 5, 10, 15 , etc.

***Requisito base: usar evaluación perezosa y función de orden superior. No currificar.***

*(playLazy scene member move tf t angle seed)*

1. **(0.5 pts)** *scene>string*: Función que recibe un escenario de entrada y entrega una representación del mismo como un string posible de visualizar de forma comprensible al usuario. Debe hacer uso del char ‘\n’ para los saltos de línea. **No utilice las funciones write y display dentro de esta función**, ya que debe retornar un string el cual pueda luego ser pasado como argumento a la función “write” o “display” para poder visualizarlo de forma comprensible al usuario.

***Requisito base: el string resultante al ser impreso por la función write o display debe ilustrar con claridad los elementos de la escena (scene) de una manera cercana a como luciría en una representación gráfica del juego (procure usar carácteres distintos para representar personajes del equipo controlado por el jugador, los controlados por CPU, proyectiles de ambos enemigos, terreno de juego, obstaculos)***

*(scene->string scene)*

**Requerimientos extra (se consideran si y sólo si la evaluación de los requerimientos obligatorios es igual o superior a 4.0. La nota máxima que se puede alcanzar es un 7.0 en RF).**

1. **(1.5 pts)** demoLazy: Función que permite realizar una simulación de una partida del juego donde a partir de un escenario (scene) generado, ambos equipos tienen un control autónomo de sus personajes en lo que respecta a movimientos y disparos. El retorno de la función es una simulación entregada a través de una lista infinita donde cada ítem de la lista es una instantánea del estado del escenario en los turnos t+1, t+2, t+3, ….. t+n hasta que se alcance el estado de VICTORY (todos los enemigos destruidos), DEFEAT jugador destruido por los enemigos (que cayó fuera de la zona de juego o que tocó cierta parte del escenario que lo pueda destruir: Fuego, agua, espinas, etc) o DRAW (empate que podría darse en caso de que al disparar proyectil se destruye a los últimos miembro del equipo contrario pero como efecto colateral (onda expansiva, objetos, etc.) también se destruye a los del equipo que realizó el disparo.

***Requisito base: usar evaluación perezosa.***

*(demoLazy scene seed)*

1. **(0,5 pts)** Considerar 4 o más tipos de trayectorias de disparo (funciones) con características diferentes, estos deben estar claramente descritos en el informe/manual de usuario.
2. **(0,5 pts)** Considerar escenarios con obstáculos que puedan generar daño al jugador y/o enemigo que los toca (Ej: Fuego, agua, cerco eléctrico, etc). Estos deben estar claramente descritos en el informe/manual de usuario y ser debidamente representados en la función “scene->string”.
3. **(0,5 pts)** Considerar escenarios y/o obstáculos del escenario que puedan ser mutables en el tiempo o al tener interacción con los jugadores/enemigos (Ej: Al recibir un disparo se destruye un muro, se puede caer una superficie si recibe un disparo o si tiene a un jugador/enemigo sobre este debido a su peso). Estos deben estar claramente descritos en el informe/manual de usuario.
4. **(0,5 pts)** *scene>xml*: Función que recibe un escenario de entrada y entrega una representación del mismo en formato XML. Usted debe implementar la función de conversión y no usar funciones disponibles en Racket.

*(scene>xml scene)*

1. **(0,5 pts)** *scene>json*: Función que recibe un escenario de entrada y entrega una representación del mismo en formato JSON. Usted debe implementar la función de conversión y no usar funciones disponibles en Racket.

*(scene>json scene)*

1. **(1,5 pts)** Establecer un menú de juego que según la entrada por teclado del usuario haga uso de todas las funciones descritas anteriormente y permitir una experiencia de juego completa (Acá se permite utilizar funciones display y/o write ), sin embargo esto no implica que las funciones descritas en los requerimientos funcionales básicos y/o adicionales tengan que ser diferentes en sus cabeceras a lo descrito en estas.

# Laboratorio 2 (Paradigma Lógico - Lenguaje Prolog)

Versión 1.0

(Cambios menores pueden incorporarse en futuras versiones a fin de aclarar o corregir errores)

**Fecha de Entrega:** 17 de octubre de 2019, hasta las 23:55 a través de Usach Virtual

* **Entregables:** Archivo ZIP con el siguiente nombre de archivo: lab2\_rut\_ApellidoPaterno.zip (ej: lab2\_12123456\_Perez.zip). Notar que no incluye dígito verificador. Dentro del archivo se debe incluir:

1. Informe en formato Word o PDF.
2. Carpeta con código fuente.
3. Archivo leeme.txt para cualquier instrucción especial para ejecución u otro, según aplique.
4. Autoevaluación.txt donde debe incluir una lista completa de todos los requisitos funcionales y no funcionales señalando una evaluación para cada uno según la siguiente escala:
   1. 0: No realizado.
   2. 0.25: Implementación con problemas mayores (funciona 25% de las veces o no funciona)
   3. 0.5: Implementación con funcionamiento irregular (funciona 50% de las veces)
   4. 0.75: Implementación con problemas menores (funciona 75% de las veces)
   5. 1: Implementación completa sin problemas (funciona 100% de las veces)
5. Archivo repositorio.txt que incluya la URL de su repositorio en GitHub el que deberá además ser compartido con los profesores y ayudantes al momento de la evaluación.

El nombre del archivo con el código fuente debe seguir el siguiente formato: NombreArchivo\_rut\_Apellidos.extension (ej: worms\_12123456\_PerezPeña.rkt)

**Objetivo del laboratorio:** Aplicar y demostrar los conocimientos adquiridos en cátedra con respecto al paradigma de programación lógico usando el lenguaje de programación Prolog.

**Resultado esperado:** Programa para la generación de tableros, validación de tableros, ejecución de partidas y evaluación de partidas.

**Informe del proyecto:** Extensión no debe superar las **15 planas** de contenido (se excluyen: portada, índice, referencias). El informe debe incluir introducción, descripción breve del problema, análisis del problema, diseño de la solución, consideraciones de implementación, **instrucciones de uso con ejemplos claros de uso**, resultados obtenidos, evaluación completa y conclusiones.

**Evaluación:** El Informe (Inf), requerimientos funcionales (RF), requerimientos no funcionales (RNF), ejecución (Eje) se evalúan por separado. La nota final de este laboratorio (NL) se calcula de la siguiente forma considerando sólo la calificación base a partir del cumplimiento de los RF y RNF obligatorios.

**Requerimientos No Funcionales obligatorios. Nota que algunos son ineludibles, esto quiere decir que al no cumplir con dicho requerimiento, su proyecto será evaluado con la nota mínima.**

1. **(obligatorio)** Incluir autoevaluación de cada uno de los requerimientos funcionales (obligatorios y extras) solicitados.
2. **(obligatorio)** La implementación debe ser en el lenguaje de programación Prolog.
3. **(obligatorio)** Usar Swi-Prolog versión 7.6.x.
4. **(1.5 pts)** Todos los predicados deben estar debidamente comentados. Indicando descripción de la función, argumentos de entrada y de salida.
5. **(2.5 pts)** Historial de trabajo en Github tomando en consideración la evolución en el desarrollo de su proyecto en distintas etapas. Se requieren **al menos 10 commits** distribuidos en un periodo de tiempo **mayor o igual a 10 días**. A modo de ejemplo, si hace todos los commits el día antes de la entrega del proyecto, este ítem tendrá 0 pts. De manera similar, si hace dos commits dos semanas antes de la entrega final y el resto los concentra en los últimos dos días, tendrá una evaluación del 25% para este item (0.375 pts). Por el contrario si demuestra constancia en los commits (con aportes claros entre uno y otro) a lo largo del periodo evaluado, este ítem será evaluado con el total del puntaje.
6. **(obligatorio)** Al final de su código incluir al menos 3 ejemplos de uso para cada uno de los predicados correspondientes a requerimientos funcionales obligatorios y los extra que realice. **Solo se revisarán aquellos predicados para los que existan los ejemplos provistos.**

**Requerimientos Funcionales Obligatorios.**

1. **(0.5 pts)**. Implementar abstracciones apropiadas para el problema. Para la implementación debe regirse por estructura de implementación de TDA vista en clases: Representación, Constructores,Predicados de Pertenencia, Selectores, Modificadores y otros predicados que operen sobre la representación. Solo aquello que utilice. Procurar hacer un uso adecuado de esta estructura a fin de no afectar la eficiencia de sus predicados principales. En el resto de los predicados debe efectivamente hacer un uso adecuado de la implementación del TDA. Solo implementar los predicados necesarios dentro de esta estructura. Dejar claramente documentado con comentarios en el código qué corresponde a la estructura base del TDA. La estructura del TDA debe considerar la existencia de un estado del juego (i.e., PLAYING, VICTORY, DEFEAT, DRAW), nivel de dificultad, puntaje, parámetros de enemigos, obstáculos y cualquier otro recurso que se incluya en el juego.

***Requisito base: Especificar representación de manera clara para cada TDA implementado. Especificar dominios, predicados, metas y todas las cláusulas necesarias.***

1. **(0.5 pts)** createScene: Predicado que permite consultar si es posible crear un escenario válido de tamaño NxM posicionando la cantidad de enemigos iniciales (parámetro ‘E’) en un nivel de dificultad D. El parámetro seed se refiere a la semilla usada para generar valores pseudoaleatorios (ver Nota 2). De ser posible, el escenario se encuentra en la variable de salida “Scene”..El parámetro ‘D’ representa la dificultad del escenario, lo que afecta por ejemplo la cantidad de personajes del equipo del jugador y posición de los obstáculos en el escenario (si hubieran).

*createScene(N, M, E, D, Seed, Scene).*

**Nota 1:** Un escenario válido debe poder ubicar la cantidad de enemigos especificados en el parámetro ‘E’. La factibilidad de poder disponer todos los enemigos en el escenario dependerá del tamaño de este. Si no es factible crear el escenario, se debe retornar nulo.

**Nota 2:** El parámetro seed se refiere a la semilla para generar valores pseudoaleatorios (ver ejemplo provisto en UsachVirtual para su referencia de cómo generar estos números sin violar aspectos del paradigma funcional. Pueden usar esta función y/o adaptarla para prolog).

* **Nota 3:** Los escenarios para este laboratorio pueden estar construidos explícitamente de forma estática como parte de la base de conocimiento de su programa, estos espacios deben incluir todos los enemigos ya posicionados. El predicado *createScene* en este caso debe consultar la base de conocimientos en base a los parámetros de la consulta y proporcionar un escenario válido a través de la variable de salida “Scene”.

Para este caso se deben implementar:

3 escenarios de 5 x 10 (Con 2, 4 y 5 enemigos posicionados inicialmente)

2 escenarios de 10 x 12 (Con 4 y 6 enemigos posicionados inicialmente)

1 escenarios de 20 x 20 (Con 8 enemigos posicionados inicialmente)

Los parámetros N, M, E, D y Seed son números enteros. Mientras que “Scene” corresponde a su TDA utilizado para representar un escenario de juego.

1. **(0.5 pts)** checkScene: Predicado que permite consultar si un escenario (scene) cumple con los criterios para ser considerado como un escenario válido (Ej: Que es realmente un espacio con dimensiones válidas, enemigos, obstáculos y jugador válidamente ubicados dentro de este, esto según su representación debe incluir validaciones que eviten la superposición de elementos en el escenario (Ej: Que las coordenadas de dos enemigos sean las mismas).

*checkScene(Scene).*

1. **(0.5 pts)** moveMember: Predicado que permite consultar si es posible mover un personaje de algún equipo en cierta dirección. Recibe como entrada un escenario en “SceneIn” y debe verificar el desplazamiento del personaje (parámetro *Member*) que se desea operar en el turno hacia la izquierda o derecha (parámetro *MoveDir*). El escenario resultante luego de generar el movimiento del personaje se encuentra en la variable “SceneOut”.

*moveMember(SceneIn, Member, MoveDir, Seed, SceneOut).*

1. **(1 pts)** shoot: Predicado que permite consultar el valor que debe tomar SceneOut para reflejar el estado de un scenario tras el disparo ejecutado por Member hasta CPU en el scenario SceneIn usando un disparo de tipo ShootType en ángulo Angle. El resutlado de un shoot considera solo el disparo desde Jugador a CPU.

*shoot(SceneIn, Member, ShootType, Angle, Seed, SceneOut).*

1. **(1 pto)** updateScene: Predicado que permite consultar el valor debe tomar SceneOut para ser el estado del escenario SceneIn en tiempo (t+1). En este caso, SceneOut considera el desplazamiento y efectos de un proyectil desde tiempo t hasta tiempo t+1 con todo lo que ello implica. Esto es, caída de objetos si los hubiera, eliminación de un enemigo, cambio del estado del juego (VICTORY, DEFEAT, DRAW).~~avance de los proyectiles~~

*updateScene(SceneIn, Seed, SceneOut).*

1. **(0.5 pts)** *sceneToString*: Predicado que permite consultar qué valor debe tomar SceneStr a partir de Scene para que el predicado sea verdadero. En este caso SceneStr corresponde a una representación String del scenario que sea posible de visualizar de forma comprensible al usuario. Debe hacer uso del char ‘\n’ para los saltos de línea. **No utilice los predicados *write* y *display* dentro de este predicado *sceneToString***, ya que en la variable de salida “SceneStr” debe ser un string el cual pueda luego ser usado por el predicado “write” o “display” para poder visualizarlo de forma comprensible al usuario en la consola de Prolog.

***Requisito base: El string resultante al ser impreso por la función write o display debe ilustrar con claridad los elementos de la escena (scene) de una manera cercana a como luciría en una representación gráfica del juego (procure usar caracteres distintos para representar personajes del equipo controlado por el jugador, los controlados por CPU, proyectiles de ambos enemigos, terreno de juego, obstáculos)***

*sceneToString(Scene, SceneStr).*

**Requerimientos extra (se consideran si y sólo si la evaluación de los requerimientos obligatorios es igual o superior a 4.0. La nota máxima que se puede alcanzar es un 7.0 en RF).**

1. *play*: Predicado que permite consultar por un turno de juego completo, esto implica mover el personaje de juego, disparar, desplazar el proyectil disparado actualizando el escenario y repetir estos mismos pasos por parte de un personaje enemigo. Para esto debe hacer uso de los predicados descritos en los requisitos básicos “moveMember”, “shoot” y “updateScene” (**2 pts**).

*play(SceneIn, Member, MoveDir, ShootType, Angle, Seed, SceneOut).*

1. *scene2xml*: Predicado equivalente a sceneToStr con la diferencia que el resultado entregado está en formato XML **(0.5 pto)**. Usted debe implementar los predicados de conversión y no usar recursos disponibles en Swi-prolog para transformar a XML.

*scene2xml( Scene, SceneXmlStr).*

1. *space2json*: Predicado equivalente a sceneToStr con la diferencia que el resultado entregado está en formato JSON **(0.5 pto)**. Usted debe implementar los predicados de conversión y no usar recursos disponibles en Swi-prolog para transformar a XML.

*space2json( Scene, SceneJsonStr).*

1. createScene: Predicado que permite consultar por scenarios generados de forma dinámica usando números aleatorios (a partir del parámetro de entrada “Seed”). Este predicado no usa la base de conocimientos sino que es capaz de encontrar nuevos escenarios válidos **(1,5 pts). Debe implementar este predicado aparte del descrito en el requerimiento funcional N° 1. Puede llamar a este predicado “createScene”.**