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 18/03/2022

Durante el semestre se trabajará en distintos paradigmas de programación, los cuales serán puestos en práctica mediante cuatro lenguajes de programación en cuatro entregas de laboratorio. El tema de laboratorio de este semestre se enfoca en un juego de mesa conocido como Dobble, Ojo de Lince, Lupa, Lince, El Lince o Spot It, entre otros (a lo largo de este enunciado se le denominará Dobble). En el siguiente video se presenta una breve explicación del juego: Dobble Kids Cómo Jugar

Los estudiantes desarrollarán soluciones en los distintos paradigmas para (1) la generación de las piezas y (2) permitir que dos o más jugadores puedan jugar en algunas de las modalidades que permite el juego.

El software en su versión final (Laboratorio N°4) deberá tener una interfaz gráfica que permita a un grupo de usuarios jugar de manera interactiva en una modalidad por turnos y con tiempo.

1. Descripción general

Varios aspectos del software quedarán abiertos a la creatividad de cada estudiante, sin embargo se debe cumplir con requerimientos mínimos obligatorios, 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 posibilida de reprobar el laboratorio.**

Para este semestre se trabaja sobre el juego Dobble. Al final del semestre los estudiantes de Paradigmas de Programación habrán implementado distintas versiones del juego (tanto para la generación de las piezas como para jugarlo) en distintos lenguajes de programación según los paradigmas que los rigen. A continuación se listan de forma general características del software que serán abordadas de forma obligatoria o complementaria durante el semestre en los distintos proyectos. Los detalles para cada laboratorio se listan posteriormente.

En Dobble se cuenta con un conjunto finito de cartas (piezas o tarjetas). En cada tarjeta se listan *n* elementos (figuras, números, letras, etc.) diferentes. Además, para cualquier par de cartas, existe un y sólo un elemento en común. A partir del conjunto de cartas, existen diversas modalidades de juego en las que pueden participar 2 o más jugadores. Algunos ejemplos de estas modalidades (hay muchas más) son:

1) Stack: Se disponen el conjunto de cartas apiladas por el reverso en el tablero. Luego se retiran las dos primeras cartas de la pila y se voltean para revelar su contenido sobre la mesa. A continuación, el primer jugador que identifique el elemento común entre estas dos cartas, se queda con las cartas. El juego termina cuando no quedan más cartas por voltear y gana aquel jugador que tiene más cartas.
2) EmptyHandsStack: Cada jugador parte con un número igual de cartas en su poder y debe quedar una pila de cartas al centro. Las cartas asignadas al jugador y las del stack deben estar volteadas sin revelar su contenido. A continuación, se voltea una carta desde la pila y todos los jugadores voltean una de las que tienen en su poder. El primero que identifique coincidencias entre su carta y la de la pila, puede descartar su carta junto a la que se volteó desde la pila. La carta descartada y la de la pila se reubican en la base de ésta. Por otro lado, el resto de los jugadores vuelve a recoger su propia carta. El juego continúa hasta que uno de los jugadores quede sin cartas en su poder, quién será el ganador.
3) EmptyHandsAllPlayers:Una variante de la modalidad 2 es que los jugadores pueden identificar coincidencias entre la carta que voltearon y las de los otros jugadores o la que se volteo desde la pila. El jugador que primero identifique coincidencias, puede descartar su carta y la de la pila ubicandolas en la base de ésta.
4) Otras modalidades de juego las puedes ver en: Video Demo - Lince Go!

En este laboratorio se jugará en modalidades como las anteriores con turnos. Esto es, un jugador juega a la vez ya sea directamente sobre la pila (modalidad 1) o contra la pila (modalidad 2: EmptyHandsStack).

Si se decide, una partida puede terminar prematuramente y se determina un ganador a partir del último estado del juego.

El juego Dobble y en particular la generación de las piezas se rige por propiedades matemáticas dentro de los cuales destacan los números primos, los planos proyectivos finitos y el plano de fano, entre otros. Te recomendamos revisar el siguiente video donde se explica el juego y cómo se generan sus piezas: How to make the card game Dobble (and the Maths behind it!). Además, se incluye como material complementario del video la documentación de un algoritmo para generar las piezas y que te puede resultar de utilidad para desarrollar este proyecto.

Adicionalmente disponemos para ti algunos videos y documentos que podrían ser de ayuda:

- https://www.youtube.com/watch?v=-g3r-uvDey0 (español)
- https://www.youtube.com/watch?v=lhTMr8RzUr8
- https://www.youtube.com/watch?v=VTDKqW_GLkw
- https://openprairie.sdstate.edu/cgi/viewcontent.cgi?article=1016&context=jur
- https://en.wikipedia.org/wiki/Dobble

Si lo estimas conveniente, para este proyecto puedes elaborar tus propios algoritmos, tomando en cuenta las restricciones conocidas hasta ahora del problema. De esta manera podrías usar fuerza bruta (**lo cual no se recomienda para valores de n muy grandes**), backtracking, heurísticas o el algoritmo antes señalado. El proyecto se evaluará con distintos valores de n, por lo que procura que tu solución no haga un uso indiscriminado de recursos, siendo idealmente fuerza bruta la última opción.

A continuación se listan las funcionalidades que serán cubiertas de manera completa o parcial en los cuatro laboratorios:

1. Generar un set (conjunto) de cartas válido a partir de un conjunto de elementos.
2. Determinar si un set de cartas dado es válido
3. A partir de una sola carta, determinar cuántos elementos se requieren para generar un set válido.
4. A partir de una sola carta, determinar cuántas cartas se pueden generar en el set.
5. A partir de un set incompleto, determinar cuales son las cartas que faltan para completarlo.
6. Permitir la generación manual de un set de cartas donde el usuario produce las cartas individualmente y las va insertando en el set con las correspondientes verificaciones para garantizar que la carta generada es válida dentro del set.
7. Permitir que dos o más jugadores jueguen por turnos en alguna de las modalidades lo que puede incluir funcionalidades como:
    a. Registrar jugadores en la partida
    b. Apilar y ordenar cartas
    c. Repartir cartas
    d. Controlar turnos
    e. Determinar ganador
    f. Contabilizar puntaje
    g. Conocer el estado del juego
    h. Terminar un juego
    i. Pasar un turno

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 inferior a 4.0 y superior o igual a un 1.1 (es decir, que al menos entrego algo)**. **No entregar laboratorio será calificado con un 1.0 y por tanto no tendrá derecho a usar comodín.**

El comodín permitirá al alumno entregar lo necesario (requerimientos funcionales, no funcionales y/o informe) para alcanzar la nota mínima requerida (4.0) en la correspondiente entrega de laboratorio para la aprobación al final del semestre. **Es fundamental que los estudiantes siempre entreguen un avance (dentro de los plazos establecidos para cada laboratorio) que permita alcanzar al menos una calificación igual o superior a 2.0 para poder optar al uso de comodines**. **Recordar que para aprobar el laboratorio de esta asignatura, todos los laboratorios deben tener una calificación igual o mayor a 4.0.**

Se recomienda no emplear los comodines como una forma de aplazar la entrega de sus laboratorios, puesto que irá acumulando trabajo hacia el final de semestre. Por lo mismo, procure trabajar en el laboratorio dentro de los plazos señalados y entregarlo aún cuando no haya podido terminarlo.

Para el caso del uso de comodín por versionamiento solo dispondrá de 2 oportunidades. Si el estudiante tiene una calificación menor a 4.0 debido al no uso o uso inadecuado de git (frecuencia, constancia, periodo), el estudiante podrá enmendar esta calificación haciendo uso efectivo de git en los siguientes laboratorios o en los anteriores. Para estos casos, el estudiante debe enviar la solicitud de aplicación de comodín por versionamiento al término del semestre a través de la plataforma habilitada para tales efectos. De ser aceptado el comodín, la nota de cierre del correspondiente laboratorio será un 4.0.

Procure realizar la autoevaluación rigurosa de cada laboratorio de manera que conozca a priori la calificación estimada que podría recibir. De esta manera se espera que tengan claridad sobre si debe o no usar comodín y en qué funcionalidades debería concentrarse para alcanzar la nota mínima. Las calificaciones de laboratorio tomarán tiempo, por lo que se recomienda NO esperar a tener una calificación del laboratorio para recién determinar si requiere usar un comodín. Esto lo puede determinar a priori con una autoevaluación a conciencia.

Si su autoevaluación indica que su calificación hubiese sido mayor o igual a 4.0, se recomienda a lo/as estudiantes empleen las instancias de recorrecciones de cada laboratorio antes de proceder a trabajar en el comodín. Si su autoevaluación fue a conciencia y rigurosa, pero aún así obtiene una calificación inferior a 4.0, podría tratarse de (1) un error/omisión en la revisión, (2) problemas en la implementación/ejecución que no fueron identificados por el/la estudiante, (3) problemas de versionamiento, interpretación, compilación debido a elementos que no fueron debidamente documentados y (4) aplicación incorrecta del paradigma, entre otras razones.

En caso de requerir el uso de comodín, podrá hacer entrega del correspondiente laboratorio corregido al final del semestre por el canal habilitado para estos efectos. La

fecha específica y espacio de entrega estará disponible en CampusVirtual al final del semestre.

Finalmente, por temas de tiempo considerando el avance del semestre, en el laboratorio final (Laboratorio 4) se podrá hacer un uso acotado del comodín. Esto quedará a disposición de los profesores correctores dependiendo de los aspectos calificados como insuficientes y la factibilidad de poder completarlos en los plazos para el cierre oficial del semestre dentro de las semanas lectivas (17 semanas).

Instrucciones generales para la entrega de TODOS los laboratorios

Versión 1.0

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

(Sus dudas las puede expresar en este mismo enunciado, incluso puede responder a preguntas de compañeros en caso de que conozca la respuesta)

Entregables: Archivo ZIP con el siguiente nombre de archivo: labN_rut_ApellidoPaterno.zip (ej: lab1_12123456_Perez.zip). **Notar que no incluye dígito verificador**. Dentro del archivo se debe incluir:

1. Informe en formato 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 requerimientos funcionales y no funcionales señalando una evaluación para cada uno según la siguiente escala:
    a. 0: No realizado.
    b. 0.25: Implementación con problemas mayores (funciona 25% de las veces o no funciona)
    c. 0.5: Implementación con funcionamiento irregular (funciona 50% de las veces)
    d. 0.75: Implementación con problemas menores (funciona 75% de las veces)
    e. 1: Implementación completa sin problemas (funciona 100% de las veces)
5. Archivo repositorio.txt que incluya la URL de su repositorio privado en GitHub[1] el que deberá además compartirlo con los profesores y ayudantes al momento de la evaluación a la cuenta ***paradigmasdiinf*** de GitHub (no lo comparta antes ya que las invitaciones expiran después de un tiempo). El nombre final del repositorio a compartir debe ser el mismo del entregable en Campus Virtual.

El nombre de todos los archivos con código fuente, informes, autoevaluación, etc. debe seguir el siguiente formato: NombreArchivo_rut_Apellidos.extension (ej: labDobble_12123456_PerezPeña.rkt; funciones_12123456_PerezPeña.rkt)

Se aplicarán descuentos en caso de no incluir estos datos e inclusive la nota mínima si el programa no puede pasar por los procedimientos automáticos de revisión por no respetar esta instrucción.

Informe del proyecto: Extensión no debe superar las **10 planas** de contenido (se excluyen: portada, índice, referencias, anexos), para más detalle de requerimientos del informe ver descripción en nuestro espacio en Campus Virtual. El informe debe incluir introducción, descripción breve del problema, análisis del problema, diseño de la solución (esquematizada y explicada brevemente), consideraciones de implementación (ej:

[1] Ver tutorial de Git en Campus Virtual donde se explica como activar el Student Pack para poder crear repositorios privados de forma gratuita.

algoritmos, bibliotecas), **instrucciones 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 TODA la elaboración del proyecto**. Cualquier inconsistencia entre lo entregado en Campus Virtual y la versión final del repositorio será evaluado con nota mínima. **Para el proceso de revisión se considerará la versión subida a CampusVirtual**.

Integridad: Lo/as estudiantes deben desarrollar los laboratorios de forma individual. Si bien lo/as estudiantes pueden apoyarse y discutir aspectos del proyecto, el diseño e implementación del mismo, la solución construida se debe producir de manera individual. No se pueden compartir diseños o implementaciones, tampoco se pueden obtener de terceros. Cómo parte de la revisión existen procedimientos de revisión de copia/plagio tanto a nivel de código como en los informes. **Al detectarse plagio, el trabajo en cuestión no será revisado por no contar con evidencia concreta sobre el resultado de aprendizaje esperado para el correspondiente laboratorio. Por lo anterior, será calificado con un 1.0. El/la estudiante tampoco podrá hacer uso de comodín en este caso ya que es el equivalente a no haber entregado dicho laboratorio**. **Por último se iniciará investigación sumaria a través de la Facultad de Ingeniería con el objeto de determinar las sanciones correspondientes.**

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.

$$if\ (Inf\ >=\ 4.0\ \&\&\ (RF\ +\ 1.0)\ >=\ 4.0\ \&\&\ (RNF\ +\ 1.0)\ >=\ 4.0)\ then$$
$$NL\ =\ 0.1\ \cdot\ Inf\ +\ 0.7\ \cdot\ (RF\ +\ 1.0)\ +\ 0.2\ \cdot\ (RNF\ +\ 3.0)$$
$$else$$
$$NL\ =\ min(Inf,\ RF\ +\ 1.0,\ RNF\ +\ 1.0)$$

;Respecto de este caso, si el estudiante entrega el laboratorio
;y algunas de sus calificaciones es 1.0 por no entrega (ej: Informe)
;**será calificado con un 1.1** para que pueda optar a comodín.
;Por lo anterior, procure siempre hacer una entrega.

Laboratorio 1 (Paradigma Funcional - Lenguaje Scheme)

Versión 1.0

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

(Sus dudas las puede expresar en este mismo enunciado, incluso puede responder a preguntas de compañeros en caso de que conozca la respuesta)

Fecha de Entrega: Ver calendario clase a clase donde se señala el hito

Objetivo del laboratorio: Aplicar conceptos del paradigma de programación funcional usando el lenguaje de programación Scheme en la resolución de un problema acotado.

Resultado esperado: Programa para generar un conjunto válido de cartas del juego Dobble y además permitir a un grupo de usuarios jugar.

Profesor responsable: Roberto González (al hacer consultas en este documento, procurar hacer la mención a **@Roberto Gonzalez Ibanez** (roberto.gonzalez.i@usach.cl) para que las notificaciones de sus consultas lleguen al profesor correspondiente)

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

1. **(obligatorio) Autoevaluación:** Incluir autoevaluación de cada uno de los requerimientos funcionales solicitados.

2. **(obligatorio) Lenguaje:** La implementación debe ser en el lenguaje de programación Scheme.

3. **(obligatorio) Versión:** Usar DrRacket versión 6.11 o superior

4. **(obligatorio) Standard:** Se deben utilizar funciones estándar del lenguaje. No emplear bibliotecas externas. Si considera integrar su solución posteriormente con C# en el laboratorio 4 debe regirse por standard R6RS.

5. **(obligatorio) No variables:** No hacer uso de función define **en combinación** con otras como set! (o similares) para emular el trabajo con variables.

6. **(1 pts) Documentación:** Todas las funciones deben estar debidamente comentadas. Indicando descripción de la función, tipo de algoritmo/estrategia empleado (ej: fuerza bruta, backtracking, si aplica) argumentos de entrada (dominio) y retorno (recorrido). En caso de que la función sea recursiva, indicar el tipo de recursión utilizada y el porqué de esta decisión.

7. **(obligatorio) Dom->Rec:** Respetar la definición de función en términos de conjunto de salida (dominio) y llegada (recorrido) sin efectos colaterales, además del nombre de las mismas (respetar mayúsculas y minúsculas).

8. **(1 pts) Organización:** Estructurar su código en archivos independientes. Un archivo para cada TDA implementado y uno para el programa principal donde se dispongan sólo las funciones requeridas en el apartado de requerimientos funcionales. Debe usar la función require/provide.

9. **(2.5 pts) Historial:** 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 (no espere a terminar la materia para empezar a trabajar en el laboratorio. Puede hacer pequeños incrementos conforme avance el curso)**. Los criterios que se consideran en la evaluación de este ítem son: fecha primer commit, fecha último commit, total commits y máximo de commits diarios. A modo de ejemplo (y solo como una referencia), 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 ítem (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) Ejemplos:** 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.**

11. **(obligatorio) Prerrequisitos:** Para cada función se establecen prerrequisitos. Estos deben ser cumplidos para que se proceda con la evaluación de la función implementada. Ej: Para evaluar la función login, debe estar implementada la función register.

Requerimientos Funcionales. **Para que el requerimiento sea evaluado, DEBE cumplir con el prerrequisito de evaluación y requisito de implementación. En caso contrario la función no será evaluada. El total de requerimientos permiten alcanzar una nota mayor que 7.0, por lo que procura realizar las funciones que consideres necesarias para alcanzar un 7.0. Si realizas todas las funciones y obtienes el puntaje máximo, la nota asignada será igualmente un 7.0. El puntaje de desborde se descarta.**

1. **(0.5 pts TDAs)**. Especificar e implementar abstracciones apropiadas para el problema. Recomendamos leer el enunciado completo y ver material complementario a fin de que analice el problema y determine el o los TDAs y representaciones apropiadas para la implementación de cada uno. Luego, planifique bien su enfoque de solución de manera que los TDAs y representaciones escogidos sean aplicables a ambos tipos de funciones.

   Para la implementación debe regirse por la estructura de especificación e 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 (ej: usar selectores, modificadores, constructores, según sea el caso. No basta con implementar un TDA y luego NO hacer uso del mismo). **Solo implementar las funciones estrictamente necesarias dentro de esta estructura**.

   A modo de ejemplo, si usa una representación basada en listas para implementar un TDA, procure especificar e implementar funciones específicas para selectores (ej: en lugar de usar car o cdr, realice implementaciones o establezca sinónimos para los selectores con nombres que resulten apropiados para el TDA. Por ejemplo (define miselector car) o (define miselector (lambda (param) …… car …….)).

   Dejar claramente documentado con comentarios en el código aquello que corresponde a la estructura base del TDA. Una estructura base que deberá considerar para el resto de las funciones corresponde a “game” que corresponde una partida de Dobble y que actúa como contenedor de los jugadores, set de cartas, jugadas, turno y estado del juego, entro otros elementos. A su vez, otros tipos de datos a modelar son los del conjunto de cartas, las cartas, los elementos de las cartas y los jugadores, entre otros.

   Debe contar además con representaciones complementarias para otros elementos que considere relevantes para abordar el problema.

   Especificar representación de manera clara para cada TDA implementado (en el informe y en el código a través de comentarios). Luego implementar constructores, funciones de pertenencia, y según se requiera selectores, modificadores y otras funciones que pueda requerir para las otras funciones requeridas a continuación.

   Las funciones especificadas e implementadas en este apartado son complementarias (de apoyo) a las funciones específicas de los dos TDAs que se señalan a continuación. Su desarrollo puede involucrar otros TDAs y tantas funciones como sean necesarias para abordar los requerimientos.

2. **(0.7 pts) TDA cardsSet - constructor**. Función constructora de conjuntos válidos de cartas para el juego Dobble.

Nombre función	cardsSet
Prerrequisitos para evaluación	Haber especificado los TDAs según lo señalado en el requerimiento 1.
Requisitos de implementación	- Usar estructuras basadas en listas - Hacer uso explícito de alguno de los tipos de recursividad revisados en clases. En esta implementación no debe trabajar de forma declarativa usando funciones provistas por el lenguaje que encapsulan el proceso recursivo. - Hacer uso efectivo de funciones de orden superior
Dominio	Elements (list) X numE(int) X maxC(int) X rndFn (fn) ● Elements: Lista desde donde se puede obtener una muestra de elementos (números, letras, figuras, etc.) para construir el conjunto de cartas. ● numE: Entero positivo que indica la cantidad de elementos esperada en cada carta ● maxC: Entero que indica la cantidad máxima de cartas que se busca generar en el conjunto. Si maxC <=0 se producen todas las cartas posibles para un conjunto válido. ● rndFn: Función de aleatorización que debe garantizar transparencia referencial. Se proporciona función de ejemplo al final del enunciado. Es opcional implementar otra. La función se puede usar internamente para aleatorizar aspectos en la generación de cartas tales como orden de los elementos, tamaño de los elementos, etc.
Recorrido	*cardsSet*
Ejemplo de otras funciones asociadas al mismo TDA u otros TDAs (Solo de referencia. Cada estudiante define sus propios TDAs)	*(firstCard ….)* *(lastCard …)* *(nextCard ….)* *etc….*
Ejemplo de uso	;lista de elementos de tipo string (cardsSet (list “A” “B” “C”) 2 -1 randomFn) ;lista de elementos a partir de TDA element definido por el usuario (cardsSet (list (element “A”) (element “B”) (element “C”)) 2 -1 randomFn)

3. **(0.7 pts) TDA cardsSet - dobble?:** Función que permite verificar si el conjunto de cartas en el conjunto corresponden a un conjunto válido. Esto es, que en cada tarjeta se listan *n* elementos (figuras, números, letras, etc.) diferentes. Además, para cualquier par de cartas, existe un y sólo un elemento en común. Si se cumplen estas condiciones la función retorna true, de lo contrario false.

Nombre función	dobble?
Prerrequisitos para evaluación	constructor cardsSet
Requisitos de implementación	Hacer uso explícito de alguno de los tipos de recursividad revisados en clases. En esta implementación no debe trabajar de forma declarativa usando funciones provistas por el lenguaje que encapsulan el proceso recursivo.
Dominio	*cardsSet*
Recorrido	*boolean*
Ejemplo de uso	(dobble? (cardsSet (list “A” “B” “C”) 2 -1 randomFn))

4. **(0.1 pts) TDA cardsSet - numCards:** Permite determinar la cantidad de cartas en el set.

Nombre función	numCards
Prerrequisitos para evaluación	constructor cardsSet
Requisitos de implementación	Implementación declarativa - funcional
Dominio	*cardsSet*
Recorrido	*int*
Ejemplo de uso	(numCards (cardsSet (list “A” “B” “C”) 2 -1 randomFn))

5. **(0.1 pts) TDA cardsSet - nthCard:** Obtiene la n-ésima (nth) carta desde el conjunto de cartas partiendo desde 0 hasta (totalCartas-1).

Nombre función	nthCard
Prerrequisitos para evaluación	constructor cardsSet
Requisitos de implementación	Implementación declarativa - funcional
Dominio	*cardsSet*
Recorrido	*card*
Ejemplo de uso	(nthCard (cardsSet (list “A” “B” “C”) 2 -1 randomFn) 1)

6. **(0.1 pts) TDA cardsSet - findTotalCards:** A partir de una carta de muestra, determina la cantidad total de cartas que se deben producir para construir un conjunto válido.

Nombre función	findTotalCards
Prerrequisitos para evaluación	constructor cardsSet, nthCard
Requisitos de implementación	Implementación declarativa - funcional
Dominio	*card*
Recorrido	*int*
Ejemplo de uso	(findTotalCards (nthCard (cardsSet (list “A” “B” “C”) 2 -1 randomFn) 1))

7. **(0.1 pts) TDA cardsSet - requiredElements:** A partir de una carta de muestra, determina la cantidad total de elementos necesarios para poder construir un conjunto válido.

Nombre función	requiredElements
Prerrequisitos para evaluación	constructor cardsSet, nthCard
Requisitos de implementación	Implementación declarativa - funcional
Dominio	*card*
Recorrido	*int*
Ejemplo de uso	(requiredElements (nthCard (cardsSet (list “A” “B” “C”) 2 -1 randomFn) 1))

8. **(0.7 pts) TDA cardsSet - missingCards:** A partir de un conjunto de cartas retorna el conjunto de cartas que hacen falta para que el set sea válido.

Nombre función	missingCards
Prerrequisitos para evaluación	constructor cardsSet, requiredElements, findTotalCards, numCards
Requisitos de implementación	Implementación declarativa - funcional o con uso explícito de recursividad.
Dominio	*cardsSet*
Recorrido	*cardsSet*
Ejemplo de uso	(findTotalCards (nthCard (cardsSet (list “A” “B” “C”) 2 -1 randomFn) 1))

9. **(0.25 pts) TDA cardsSet - cardsSet->string:** convierte un conjunto de cartas a una representación basada en strings que posteriormente pueda visualizarse a través de la función display.

Nombre función	cardsSet->string
Prerrequisitos para evaluación	missingCards
Requisitos de implementación	Implementación con uso explícito de recursividad (de cola o natural).
Dominio	*cardsSet*
Recorrido	*string*
Ejemplo de uso	(cardsSet->string (cardsSet (list “A” “B” “C”)))

10. **(0.3 pts) TDA game - constructor:** Función a partir de la cual se crea e inicializa un tablero a partir de los parámetros de entrada.

Nombre función	game
Prerrequisitos para evaluación	TDA cardsSet completo
Requisitos de implementación	Implementación declarativa - funcional o con uso explícito de recursividad si aplica. Uso efectivo de funciones de orden superior.
Dominio	*numPlayers(int) X cardsSet X mode (fn) X rndFn (fn)* ● numPlayers: Entero que señala la cantidad de jugadores. ● cardsSet: Conjunto válido de cartas ● mode: Función que permite internamente determinar el modo de juego, la forma en que se ejecutan los turnos, repartición de cartas, etc. ● rndFn: Función de aleatorización que debe garantizar transparencia referencial. Se proporciona función de ejemplo al final del enunciado. Es opcional implementar otra. La función se puede usar internamente para aleatorizar aspectos en la generación de cartas tales como orden de los elementos, tamaño de los elementos, etc.
Recorrido	*game* , Corresponde a la estructura que alberga el área de juego, las piezas disponibles, jugadores registrados, sus cartas y el estado del juego, entre otros elementos
Ejemplo de uso	(define game1 (game 4 (cardsSet (list “A” “B” “C”)) stackMode randomFn))

11. **(0.2 pts) TDA game - stackMode:** Función que permite retirar y voltear las dos cartas superiores del stack de cartas en el juego y las dispone en el área de juego.

Nombre función	stackMode
Prerrequisitos para evaluación	constructor game
Requisitos de implementación	Implementación declarativa - funcional o con uso explícito de recursividad si aplica.
Dominio	*cardsSet*
Recorrido	*cardsSet*
Ejemplo de uso	uso interno de la función como parte del constructor game (game 4 (cardsSet (list “A” “B” “C”)) stackMode randomFn)

12. **(0.2 pts) TDA game - register:** Función para registrar a un jugador en un juego. Los jugadores tienen un nombre único y no puede exceder la cantidad de jugadores registrados.

Nombre función	register
Prerrequisitos para evaluación	TDA cardsSet completo, constructor game
Requisitos de implementación	Implementación declarativa - funcional o con uso explícito de recursividad si aplica. Uso efectivo de funciones de orden superior.
Dominio	*user(String) X game* ● user: Representa el nombre del usuario el que debe ser único. ● game:Representa la estructura de un juego donde se pueden registrar jugadores.
Recorrido	*game*
Ejemplo de uso	(register “user1” (game 4 (cardsSet (list “A” “B” “C”)) stackMode randomFn))

13. **(0.1 pts) TDA game - whoseTurnIsIt?:** Función que retorna el usuario a quién le corresponde jugar en el turno.

Nombre función	whoseTurnIsIt?
Prerrequisitos para evaluación	constructor game, register, play, pass, spotIt
Requisitos de implementación	Implementación declarativa - funcional o con uso explícito de recursividad si aplica. Uso efectivo de funciones de orden superior.
Dominio	*game*
Recorrido	*string*
Ejemplo de uso	(whoseTurnIsIt? (game 4 (cardsSet (list “A” “B” “C”)) stackMode randomFn))

14. **(0.5 pts) TDA game - play:** permite realizar una jugada a partir de la acción especificadas por la función currificada action. Las jugadas parten sin una acción especificada, lo que activa internamente el uso de la función stackMode para el trabajo sobre las cartas.

Nombre función	play
Prerrequisitos para evaluación	whoseTurnIsIt?
Requisitos de implementación	Implementación con uso explícito de recursividad (de cola o natural).

Dominio	*game X action (fn)*
	- *game: Corresponde a un juego con jugadores y conjunto de cartas*
	- *action: Función que determina la acción a realizar en el juego. en caso de que action sea*
	  - *null: solo se hace el volteo inicial de cartas según la modalidad de juego activa y no se pasa el turno.*
	  - *spotIt: función currificada para realizar la comparación entre las cartas volteadas a partir del elemento indicado por el usuario. Luego de esta función se contempla cambio de turno*
	  - *pass: función que permite pasar el turno, procurando volver las cartas a su sitio de acuerdo a la modalidad de juego.*
	  - *finish: función que da término a la partida cambiando el estado del juego a Terminado e indicando ganador/perdedor/empate. Luego de finish no se puede continuar la partida.*

Recorrido	*game*
Ejemplo de uso	(define myGame (game 4 (cardsSet (list “A” “B” “C”)) stackMode randomFn)) (play mGame null randomFn) (play mGame pass randomFn) (play mGame (spotIt “A”) randomFn) (play mGame finish randomFn)

15. **(0.1 pts) TDA game - status:** Función que retorna el estado actual del juego.

Nombre función	status
Prerrequisitos para evaluación	play
Requisitos de implementación	Implementación declarativa - funcional o con uso explícito de recursividad si aplica. Uso efectivo de funciones de orden superior.
Dominio	*game*
Recorrido	*string*
Ejemplo de uso	(status (game 4 (cardsSet (list “A” “B” “C”)) stackMode randomFn))

16. **(0.1 pts) TDA game - score:** Función que retorna el puntaje de un jugador a partir de su nombre de usuario.

Nombre función	score
Prerrequisitos para evaluación	play
Requisitos de implementación	Implementación declarativa - funcional o con uso explícito de recursividad si aplica. Uso efectivo de funciones de orden superior.
Dominio	*game*
Recorrido	*int*
Ejemplo de uso	(score (game 4 (cardsSet (list “A” “B” “C”)) stackMode randomFn) “user3”)

17. **(0.25 pts) TDA game - game->string:** convierte un juego/partida a una representación basada en strings que posteriormente pueda visualizarse a través de la función display.

Nombre función	game->string
Prerrequisitos para evaluación	play
Requisitos de implementación	Implementación con uso explícito de recursividad (de cola o natural).
Dominio	*game*
Recorrido	*string*
Ejemplo de uso	(game->string (game 4 (cardsSet (list “A” “B” “C”)) stackMode randomFn))

18. **(0.5 pts) TDA cardsSet - addCard:** Función que permite agregar cartas a un set de manera manual procurando verificar que las cartas incorporadas no violan las restricciones de un set válido, aunque incompleto. Por tanto, no pueden ocurrir situaciones como que dos cartas en el set tienen más de una figura en común. Si una violación ocurre, se devuelve el conjunto de entrada.

Nombre función	addCard
Prerrequisitos para evaluación	Requerimientos 1 al 9 implementados, play
Requisitos de implementación	Implementación declarativa - funcional o con uso explícito de recursividad si aplica.
Dominio	*cardsSet x card*
Recorrido	*cardsSet*

Ejemplo de uso	(addCard
	  (addCard
	  (addCard emptyCardsSet (card “A” “B”))
	  (card “B” “C”))
	  (card “A” “C”)
	)

19. **(1 pto) TDA game - empyHandsStackMode:** Función que permite retirar y voltear la carta en el tope del stack y una de las cartas del usuario seleccionada de manera aleatoria a partir de la función de aleatorización registrada en el juego. Al terminar un turno, si el jugador acierta se ubican en la base del stack, de lo contrario la carta del jugador vuelva a sus cartas y la del stack vuelve a la base de éste.

Nombre función	emptyHandsStackMode
Prerrequisitos para evaluación	Requerimientos 1 al 9 implementados, play
Requisitos de implementación	Implementación declarativa - funcional o con uso explícito de recursividad si aplica.
Dominio	*cardsSet*
Recorrido	*cardsSet*
Ejemplo de uso	uso interno de la función como parte del constructor game (game 4 (cardsSet (list “A” “B” “C”)) emptyHandsStackMode randomFn)

20. **(0.5 pto) TDA game - myMode:** Crea tu propia modalidad de juego sin romper las reglas generales del juego y sin afectar los dominios y recorridos de las funciones base. En concreto, debe se tratada igual que la función stackMode y emptyHandsMode.

Nombre función	myMode
Prerrequisitos para evaluación	Requerimientos 1 al 9 implementados, play
Requisitos de implementación	Implementación declarativa - funcional o con uso explícito de recursividad si aplica.
Dominio	*cardsSet*
Recorrido	*cardsSet*
Ejemplo de uso	uso interno de la función como parte del constructor game (game 4 (cardsSet (list “A” “B” “C”)) myMode randomFn)

Script básico Pruebas

El código presentado a continuación contiene ejemplos que le permitirán probar todas las funciones. No obstante, estas funciones no cubren todos los escenarios posibles. Estos ejemplos le servirán como referencia para su autoevaluación, sin embargo se recomienda que pueda variar los ejemplos y probar distintos escenarios.

En color morado se listan funciones a implementar

```
;función random para la selección aleatoria de elementos desde un conjunto, asignación aleatoria de cartas a ;jugadores, ordenamiento aleatorio de cartas en la pila, etc.
(define randomFn (lambda ……))

;conjunto de elementos con los que se podrían generar ls cartas. Esta lista es solo un ejemplo. En la práctica podría albergar cualquier ;tipo de elemento y de cualquier tipo de dato.
(define elementsSet (list “A” “B” “C” “D” “E” “F” … “Z” ))

;cantidad de jugadores de la partida
(define numPlayers 4)

;cantidad de elementos requeridos para cada carta
(define numElementsPerCard 3)

;máxima cantidad de cartas a generar
(define maxCards 5)  ;para generar la cantidad necesaria de cartas para un set válido

;se genera un conjunto de cartas incompleto
(define dobbleSet0 (cardsSet elementsSet numElementsPerCard maxCards randomFn))

;se consulta si el set generado es un set válido del juego dobble
(dobble? dobbleSet0)

;retorna la cantidad de cartas en el set
(numCards dobbleSet0)

;retorna la máxima cantidad de cartas que se podrían generar a partir de la información recogida de una carta
(findTotalCards (nthCard dobbleSet0 1))

;retorna la cantidad de elementos que se requieren generar a partir de la información recogida de una carta
(requieredElements (nthCard   dobbleSet0 1))

;retorna las cartas que faltan
(missingCards dobbleSet0)

;generar un set completo de cartas lo que queda estipulado con maxCards = -1
(define dobbleSet1 (cardsSet elementsSet numElementsPerCard -1 randomFn))

;crea una partida del juego con el conjunto de cartas, en modo tradicional y con la función aleatoria
;definida para la asignación de cartas y ordenamiento en la pila.
(define game1 (game numPlayers dobbleSet1 stackMode randomFn))

;registra 3 usuarios con nombres de usuario diferentes
(define game2 (register “user1” game1))
(define game3 (register “user2” game2))
(define game4 (register “user3” game3))

;intenta registrar al usuario “user3” que ya fue registrado
(define game5 (register “user3” game4))

;registra al cuarto jugador
(define game6 (register “user4” game5))

;intenta registrar a un quinto jugador
(define game7 (register “user5” game5))

;inicia una partida
(display (cardsSet->string dobbleSet1))

;retorna el nombre de usuario a quien corresponde el turno
(whoseTurnIsIt?  game7)
```

;turno “User1”: al estar jugando en modo Stack (función stackMode), saca las dos cartas que se encuentran en el tope del stack, las voltea y ubica en la mesa. No cambia de turno
(define game8 (play game7 null))

;turno “User1”: pasa el turno al siguiente jugador y las cartas volteadas se vuelven a posicionar en la base del stack
(define game9 (play game8 pass))

;turno “User2”: al estar jugando en modo Stack, saca las dos cartas que se encuentran en el tope del stack, las voltea y ubica en la mesa. No cambia de turno
(define game10 (play game9 null))

;turno “User2”: señala el elemento común en las cartas volteadas y pasa el turno al siguiente jugador.
;Si los elementos indicados corresponden, se traspasan al jugador, de lo contrario vuelven a la base del stack.
(define game11 (play game10 (spotIt “A”)))

;turno “User3”: da por terminado el juego sin cambiar el turno.
(define game12 (play game11 finish))

;retorna el estado actual del juego
(status game12)

;retorna el puntaje del jugador “user1”
(score game12 “user1”)

;despliega toda la información del juego procurando no revelar el contenido de las cartas en el set.
(display (game->string game12))