Universidad de Santiago de Chile
Facultad de Ingeniería
Departamento de Ingeniería Informática
Paradigmas de Programación

Paradigmas de Programación

Proyecto semestral de laboratorio

Versión 1.0 - actualizada al 01/11/2024

Laboratorio 1 (Paradigma Lógico - Lenguaje Prolog)

Versión 1.0 1/11/24: Versión final v1

Versión 0.9 20/10/24 : Versión preliminar

(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)

Enunciado General: Procure consultar los aspectos generales del proyecto de laboratorio en el documento general.

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: Juego Conecta4

Profesor responsable: @ Gonzalo Martinez(gonzalo.martinez@usach.cl)

Recomendaciones: El laboratorio está diseñado como un conjunto de ejercicios a abordar bajo cada paradigma. En este sentido, el desarrollo del laboratorio constituye un espacio para practicar y prepararse además para la evaluación de cátedra del correspondiente paradigma. Por tanto, se recomienda incorporar en sus hábitos de estudio/trabajo el desarrollo de las funcionalidades de forma diaria. En total el laboratorio 1 lista N funcionalidades a cubrir. Para alcanzar la nota de aprobación, se deben cubrir las M primeras y para la nota máxima se pueden cubrir Q más. Procure destinar tiempo para analizar y hacer una propuesta de diseño para el laboratorio completo antes de proceder a la implementación de la solución. No es necesario que sus predicados implementen comprobación de tipo, esto es opcional.

Requerimientos No Funcionales (RNF)

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

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. La estructura debe ser igual que la entrega del laboratorio 1. Ver dicho enunciado para mayor información.

2. **(obligatorio) Lenguaje:** La implementación debe ser en el lenguaje de programación Prolog en base a una programación principalmente declarativa.

3. **(obligatorio) Versión:** Usar Swi-prolog versión 8.4 o superior.

4. **(obligatorio) Standard:** Se deben utilizar predicados estándar del lenguaje. No emplear bibliotecas externas.

5. **(1 pts) Documentación:** Todos los predicados deben estar debidamente comentados. Indicando descripción del predicado, tipo de algoritmo/estrategia empleado (ej: fuerza bruta, backtracking, si aplica) argumentos de entrada (dominio) y argumentos de salida (recorrido).

6. **(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 los predicados requeridos en el apartado de requerimientos funcionales. Debe usar el predicado “module” y “use_module”.

7. **(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.

8. **(obligatorio) Script de pruebas (pruebas_RUT_Apellidos.pl): Al igual que la primera entrega ver en detalle las instrucciones al final de este enunciado.**

9. **(obligatorio) Prerrequisitos:** Para cada funcionalidad solicitada se establecen prerrequisitos. Estos deben ser cumplidos para que se proceda con la evaluación del predicado implementado.

Requerimientos Funcionales (RF)

La nota correspondiente al apartado de RF comienza en 1.0 y por cada RF correcto dicho puntaje escrito en el enunciado se suma a la nota base.

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.

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 (el general y el presentado en este documento) 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 y pertinentes para abordar el problema bajo el paradigma lógico utilizando listas.

   Para la implementación debe regirse por la estructura de especificación e implementación de TDA vista en clases: Representación, Constructores, Funciones/predicados de Pertenencia, Selectores, Modificadores y Otros predicados. Procurar hacer un uso adecuado de esta estructura a fin de no afectar la eficiencia de sus predicados. En el resto de predicados 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 los predicados estrictamente necesarios dentro de esta estructura**.

   A modo de ejemplo, si usa una representación basada en listas para implementar un TDA, procure especificar e implementar predicados específicos para selectores (ej: en lugar de usar solo [|] y otras sintaxis propias del TDA lista, realice implementaciones o establezca sinónimos con nombres que resulten apropiados para el TDA. Por ejemplo primerElemento([X|Y], X).

   Dejar claramente documentado con comentarios en el código aquello que corresponde a la estructura base del TDA. Estructura bases que deberá considerar para el resto de los predicados que corresponden a la solución del sistema que va a implementar.

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 y según se requiera, implemente funciones/predicados de pertenencia, selectores, modificadores y otros predicados que pueda requerir para las otras funcionalidades listadas a continuación.

Los predicados especificados e implementados en este apartado son complementarias (de apoyo) a los predicados específicos de los dos TDAs que se señalan a continuación. Su desarrollo puede involucrar otros TDAs y tantos predicados como sean necesarios para abordar los requerimientos.

1. **RF02. (0.1 pts) TDA Player - constructor**. Predicado que permite crear un jugador.

Predicado	**player**
Prerrequisitos para evaluación (req. funcionales)	1
Requisitos de implementación	Usar estructuras basadas en listas y/o pares
Dominio	id (int) X name (string) X color (string) X wins (int) X losses (int) X draws (int) X remaining_pieces (int) X Player donde: ● wins: número de veces que ha ganado ● losses: número de veces que ha ganado ● draws: número de veces que ha estado en empate ● remaining-pieces: cantidad de fichas actuales ○ La regla del juego es que se comienza con 21 fichas, pero ustedes pueden crear juegos más cortos con una cantidad de fichas mayor a 4 y menor a 21. ○ La cantidad de fichas por jugador no afecta el desarrollo del laboratorio ni las jugadas a realizar.

Consulta de ejemplo	`player(1, "Juan", "red", 0, 0, 0, 21, Player)`
	
	`player(2, "Juan", "yellow", 0, 0, 0, 21, Player)`

2. **RF03. (0.1 pts) TDA Piece - constructor**. Predicado que crea una ficha de Conecta4.

Predicado	**piece**
Prerrequisitos para evaluación (req. funcionales)	2
Requisitos de implementación	Usar estructuras basadas en listas y/o pares
Dominio	color (string)

Ejemplo de uso	`piece("red", Piece)`
	
	`piece("yellow", Piece)`

3. **RF04. (0.2 pts) TDA Board - constructor**. Crear un tablero de Conecta4.

Predicado	**board**
Prerrequisitos para evaluación (req. funcionales)	3
Requisitos de implementación	Usar estructuras basadas en listas y/o pares. El tablero debe ser de 7x6 (7 columnas, 6 filas) y debe estar vacío puesto que nadie ha jugado de momento, es un contructor.
Dominio	No recibe parámetros de entrada
Ejemplo de uso	`board(Board)`

4. **RF05. (0.1 pts) TDA Board - otros - sePuedeJugar?**. Predicado que permite verificar si se puede realizar más jugadas en el tablero.

Predicado	**can_play**
Prerrequisitos para evaluación (req. funcionales)	4
Requisitos de implementación	Verifica si en el tablero se puede realizar una jugada. Para realizar una jugada en el tablero debe existir al menos una posición disponible.
Dominio	board (board)

Ejemplo de uso	`can_play(EmptyBoard)`
	
	`can_play(CurrentBoard)`

5. **RF06. (0.7 pts) TDA Board - modificador - jugar ficha**. Jugar una ficha en el tablero

Predicado	**play_piece**
Prerrequisitos para evaluación (req. funcionales)	5
Requisitos de implementación	El predicado debe colocar la ficha en la posición más baja disponible de la columna seleccionada.
Dominio	Board (board) X Column (int) X Piece (piece) X NewBoard (board)
Ejemplo de uso	`play_piece(EmptyBoard, 3, RedPiece, NewBoard)`

6. **RF07. (0.2 pts) TDA Board - otros - verificar victoria vertical**. Predicado que permite verificar el estado actual del tablero y entregar el posible ganador que cumple con la regla de conectar 4 fichas de forma vertical.

Predicado	**check_vertical_win**
Prerrequisitos para evaluación (req. funcionales)	6
Requisitos de implementación	Resolver con recursividad. La función debe verificar si hay 4 fichas consecutivas del mismo color en cualquier columna.
Dominio	board (board) X *int (1 si gana jugador 1, 2 si gana jugador 2, 0 si no hay ganador vertical)*
Ejemplo de uso	`check_vertical_win(CurrentBoard, Winner).`

7. **RF08 (0.1 pts) TDA Board - otros - verificar victoria horizontal**. Predicado que permite verificar el estado actual del tablero y entregar el posible ganador que cumple con la regla de conectar 4 fichas de forma horizontal.

Predicado	**check-horizontal-win**
Prerrequisitos para evaluación (req. funcionales)	7
Requisitos de implementación	Resolver con recursividad natural. La función debe verificar si hay 4 fichas consecutivas del mismo color en cualquier fila.
Dominio	board (board)
Recorrido	*int (1 si gana jugador 1, 2 si gana jugador 2, 0 si no hay ganador vertical)*
Ejemplo de uso	`check_horizontal_win(CurrentBoard, Winner).`

8. **RF09. (0.3 pts) TDA Board - otros - verificar victoria diagonal**. Predicado que permite verificar el estado actual del tablero y entregar el posible ganador que cumple con la regla de conectar 4 fichas de forma diagonal.

Predicado	**check_diagonal_win**
Prerrequisitos para evaluación (req. funcionales)	8
Requisitos de implementación	Resolver con algún tipo de recursión (natural o cola). La función debe verificar si hay 4 fichas consecutivas del mismo color en cualquier diagonal (ascendente o descendente).
Dominio	board (board)
Recorrido	*int (1 si gana jugador 1, 2 si gana jugador 2, 0 si no hay ganador vertical)*
Ejemplo de uso	`check_diagonal_win(CurrentBoard, Winner).`

9. **RF10. (0.1 pts) TDA Board - otros - entregarGanador**. Predicado que permite verificar el estado actual del tablero y entregar el posible ganador que cumple con la regla de conectar 4 fichas de forma diagonal.

Nombre predicado	**who_is_winner**
Prerrequisitos para evaluación (req. funcionales)	7,8,9
Requisitos de implementación	Utilizar las predicados de verificación vertical, horizontal y diagonal para determinar si hay un ganador. Un ganador puede ser vertical, horizontal o diagonal. Realizar de forma declarativa.
Dominio	board (Board)

Recorrido	*int (1 si gana jugador 1, 2 si gana jugador 2, 0 si no hay ganador vertical)*
Ejemplo de uso	`who_is_winner(Board, Winner)`

10. **RF11. (0.1 pts) TDA Game - constructor**. Predicado que permite crear una nueva partida.

Nombre función	**game**
Prerrequisitos para evaluación (req. funcionales)	10
Requisitos de implementación	Usar estructuras basadas en listas y/o pares
Dominio	player1 (player) X player2 (player) X board (board) X current-turn (int)
Ejemplo de uso	`game(Game, Board, Player1, Player2, CurrentTurn, NewGame)`

11. **RF12. (0.1 pts) TDA Game - otros - history**. Predicado que genera un historial de movimientos de la partida.

Nombre función	**game_history**
Prerrequisitos para evaluación (req. funcionales)	11
Dominio	game (game)
Ejemplo de uso	`game_history(Game, CurrentHistory)`

12. **RF13. (0.1 pts) TDA Game - otros - esEmpate?**. Predicado que verifica si el estado actual del juego es empate.

Nombre función	**is_draw**
Prerrequisitos para evaluación (req. funcionales)	12
Requisitos de implementación	Verificar si el tablero está lleno o si ambos jugadores se han quedado sin fichas. Tablero lleno significa que todas las posiciones del mismo se encuentran ocupadas.
Dominio	game (game)
Recorrido	*boolean (#t si es empate, #f si no)*
Ejemplo de uso	`is_draw(Game)`

13. **RF14. (0.1 pts) TDA Player - otros - actualizarEstadisticas**. Predicado que actualiza las estadísticas del jugador, ya sea victoria, derrotas o empates.

Predicado	**update_stats**
Prerrequisitos para evaluación (req. funcionales)	13
Requisitos de implementación	Actualizar las estadísticas del jugador (victorias, derrotas o empates) después de finalizar un juego.
Dominio	Game X EstadisticasPrevias X NuevasEstadisticas
Ejemplo de uso	`update_stats(Game, OldStats, NewStats)`

14. **RF15. (0.1 pts) TDA Game - selector - getCurrentPlayer**. Predicado que obtiene el jugador cuyo turno está en curso.

Nombre función	**get_current_player**
Prerrequisitos para evaluación (req. funcionales)	14
Requisitos de implementación	Obtener el jugador cuyo turno está en curso.
Dominio	game (game)
Recorrido	*player*
Ejemplo de uso	`get_current_player(Game, CurrentPlayer)`

15. **RF16. (0.1 pts) TDA Game - selector - board_get_state**. Predicado que entrega por pantalla el estado actual del tablero en el juego.

Nombre función	**game_get_board**
Prerrequisitos para evaluación (req. funcionales)	15
Requisitos de implementación	Obtener una representación del estado actual del tablero.
Dominio	~~Game X Player~~ Game X Board
Ejemplo de uso	`get_board(Game, CurrentBoard)`

16. **RF17. (0.1 pts) TDA Game - modificador - game-set-end**. Predicado finaliza el juego actualizando las estadísticas de los jugadores según el resultado.

Nombre función	**end_game**
Prerrequisitos para evaluación (req. funcionales)	15
Requisitos de implementación	Finalizar el juego, actualizando las estadísticas de los jugadores según el resultado.
Dominio	Game (game) X Game
Ejemplo de uso	`end_game(Game, EndGame).`

17. **RF18. (3,5 pts) TDA Game - modificador - realizarMovimiento**. Predicado que realiza un movimiento.

Nombre función	**player_play**
Prerrequisitos para evaluación (req. funcionales)	7

Requisitos de implementación	Resolver de manera declarativa.
	- Verificar que sea el turno del jugador correcto.
	- Actualizar el estado del juego después de realizar un movimiento:
	  ● Colocar la pieza en el tablero.
	  ● Cambiar el turno al otro jugador.
	  ● Disminuir la cantidad de fichas del jugador que realizó el movimiento.
	  ● Actualizar el historial de movimientos.
	- Verificar si hay un ganador o si el juego ha terminado en empate después del movimiento.
	- Si el juego ha terminado, actualizar las estadísticas de ambos jugadores.

Dominio	game (game) X player (player) X column (int)

Recorrido	game
Ejemplo de uso	`player_play(Game, Player, Column, NewGame)`

Script de ejecución

Cada script contiene las consultas a realizar al programa cargado en el intérprete de Prolog (swi-prolog). La revisión consiste en que primero cargaremos su programa extensión .pl al interprete y luego ejecutaremos cada script por separado para consultar su entrega.

IMPORTANTE: EN EL PRESENTE NO CAMBIAR, NI BORRAR EL SCRIPT QUE SE PRESENTA A CONTINUACIÓN.

SCRIPT DE EJECUCIÓN (script_base_RUT_NOMBRE_APELLIDOS.pl)

Usted también debe entregar 2 scripts más escritos de la siguiente manera:

SCRIPT DE EJECUCIÓN PROPIO 1 (script1_RUT_NOMBRE_APELLIDOS.pl)

SCRIPT DE EJECUCIÓN PROPIO 1 (script2_RUT_NOMBRE_APELLIDOS.pl)

El script base debe contener lo siguiente: NO COMENTADO. La revisión tomará cada script y lo ejecutará, usted debe asegurar que el script funciona sin que nosotros los revisores tengamos que realizar modificación alguna.

Script de ejecución base (script_base_RUT_NOMBRE_APELLIDOS.pl)

```
% 1. Crear jugadores (10 fichas cada uno para un juego corto)
player(1, "Juan", "red", 0, 0, 0, 10, P1),
player(2, "Mauricio", "yellow", 0, 0, 0, 10, P2),
% 2. Crear fichas
piece("red", RedPiece),
piece("yellow", YellowPiece),
% 3. Crear tablero inicial vacío
board(EmptyBoard),
% 4. Crear nuevo juego
game(P1, P2, EmptyBoard, 1, G0),
% 5. Realizando movimientos para crear una victoria diagonal
player_play(G0, P1, 0, G1),    % Juan juega en columna 0
player_play(G1, P2, 1, G2),    % Mauricio juega en columna 1
player_play(G2, P1, 1, G3),    % Juan juega en columna 1
```

```
player_play(G3, P2, 2, G4),    % Mauricio juega en columna 2
player_play(G4, P1, 2, G5),    % Juan juega en columna 2
player_play(G5, P2, 3, G6),    % Mauricio juega en columna 3
player_play(G6, P1, 2, G7),    % Juan juega en columna 2
player_play(G7, P2, 3, G8),    % Mauricio juega en columna 3
player_play(G8, P1, 3, G9),    % Juan juega en columna 3
player_play(G9, P2, 0, G10),   % Mauricio juega en columna 0
player_play(G10, P1, 3, G11),  % Juan juega en columna 3 (victoria diagonal)
% 6. Verificaciones del estado del juego
write('¿Se puede jugar en el tablero vacío? '),
can_play(EmptyBoard), % Si se puede seguir jugando, el programa continuará
nl,
game_get_board(G11, CurrentBoard),
write('¿Se puede jugar después de 11 movimientos? '),
can_play(CurrentBoard),
nl,
 write('Jugador actual después de 11 movimientos: '),
   get_current_player(G11, CurrentPlayer),
   write(CurrentPlayer),
   nl,

% 7. Verificaciones de victoria
   write('Verificación de victoria vertical: '),
   check_vertical_win(CurrentBoard, VerticalWinner),
   write(VerticalWinner),
   nl,

   write('Verificación de victoria horizontal: '),
   check_horizontal_win(CurrentBoard, HorizontalWinner),
   write(HorizontalWinner),
   nl,
```

```
    write('Verificación de victoria diagonal: '),
    check_diagonal_win(CurrentBoard, DiagonalWinner),
    write(DiagonalWinner),
    nl,

    write('Verificación de ganador: '),
    who_is_winner(CurrentBoard, Winner),
    write(Winner),
    nl,

% 8. Verificación de empate
    write('¿Es empate? '),
    is_draw(G11),
    nl,

% 9. Finalizar juego y actualizar estadísticas
    end_game(G11, EndedGame),

% 10. Mostrar historial de movimientos
    write('Historial de movimientos: '),
    game_history(EndedGame, History),
    write(History),
    nl,

% 11. Mostrar estado final del tablero
    write('Estado final del tablero: '),
    game_get_board(EndedGame, FinalBoard),
    write(FinalBoard).
```