# Programming Paradigms 2023: Coursework 2

## 1. Overview

The following files are all provided, they must be downloaded from Moodle:

```
Sample_java.zip      // the sample Java code
Sample_haskell.hs  // the sample Haskell code
```

Chinese Chess (Xiangqi) is a two-player strategy board game which represents a battle between two armies (red and black). Similar to the Western Chess, the aim of this game is to checkmate the opponent's general(king) and win the game.

In this coursework, you will be given some code in both Java and Haskell which partially implements moves in Chinese Chess. For this coursework you will need to complete the code to complete tasks specifiied below for Java and Haskell. Please note that total marks for coursework 2 are given out of 100.

Through this exercise, you will learn differences between coding in Java using OOP and Haskell using Functional Programming. When you have finished this coursework, you will be given an in-lecture Moodle quiz when you will be asked questions contrasting coding styles in Java and Haskell.

The remainder of this coursework sheet is as follows:

- Section 2 below provides a simple description of the game. For more details, please refer to its Wiki page: https://en.wikipedia.org/wiki/Xiangqi.
- Section 3 describes the Java sample code and tasks.
- Section 4 describes the Haskell sample code and tasks.
- Section 5 gives submission instructions along with penalties for late submssion and information about acasdemic misconduct.

**Ensure you read and understand all sections.**

## 2. Chinese Chess Description

### Board

The board of the game consists of 9 vertical and 10 horizontal lines. All the pieces are placed at the intersections, as in the game Go.

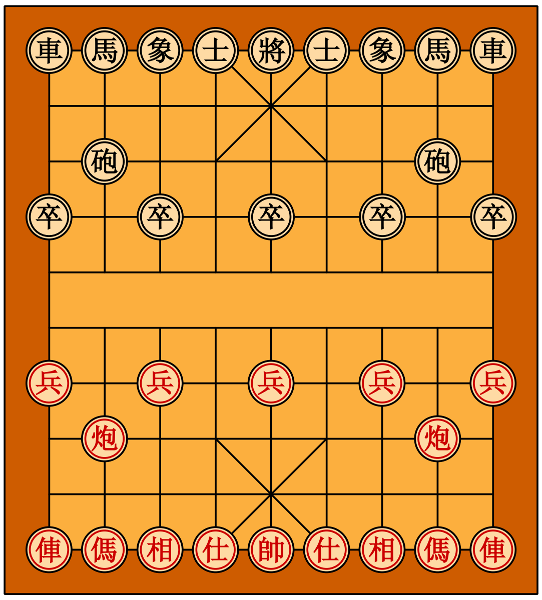


Figure 1: An example of Chinese Chess Board

A river between the 5th and 6th horizontal lines divides the board into two areas, i.e., one area for each player. There is a special zone in each zone called "palace" which is centred at the first to third and eighth to tenth horizontal lines of the board respectively. Each palace consists of three points by three points, demarcated by two diagonal lines connecting opposite corners and intersecting at the centre point (see figure 2).

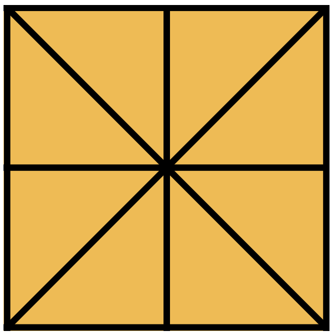


Figure 2: An example of "gong" in Chinese Chess

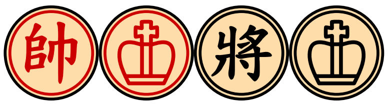
## Rules

The pieces start in the position shown in Figure 1, and typically the red side will move first. Each player in turn moves one piece from the point it occupies, to another point. A piece can only be moved onto an empty point or a point occupied by an enemy piece. In the latter case, the enemy piece will be captured and removed from the board. The game ends when one player checkmates the other's general.

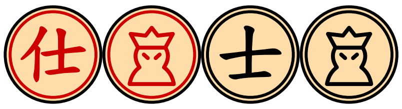
## Pieces

Each player controls 16 pieces of 7 types. Followings are a simple descriptions on how these pieces could be moved on the Chinese Chess board.

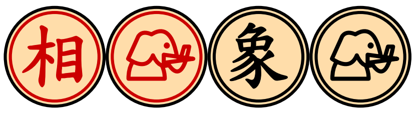
- General: The general (or king) starts the game at the midpoint of the back edge, within the palace. It may move one point orthogonally and cannot leave the palace with only one exception. If the two generals face each other along the same vertical line, i.e., there is no intervening piece in between, the general can move along the vertical line to directly capture the enemy's general.



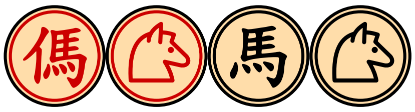
- Advisor: The advisors (or guards) start on either side of the general. They move one point diagonally and are not allowed to leave the palace (same as the general).



- Elephant: The Elephants are located next to the advisors and move two points diagonally. In addition, the elephants cannot jump over intervening pieces. That is, if there is a piece (no matter what color it is) located one point diagonally to the elephant, then the elephant cannot move diagonally towards that direction. Moreover, the elephants are not allowed to cross the river.



- Horse: The horses are located next to the Elephants. The horse moves one point orthogonally and then one point diagonally away from its former position. Unlike the knights in Western Chess, the horse can be blocked by a piece (no matter what color it is) located one point vertically or horizontally adjacent to it. For example, in Figure 3, all the green lines are valid moves for the horse, while the red lines indicate the moves of the horse are blocked by another piece.



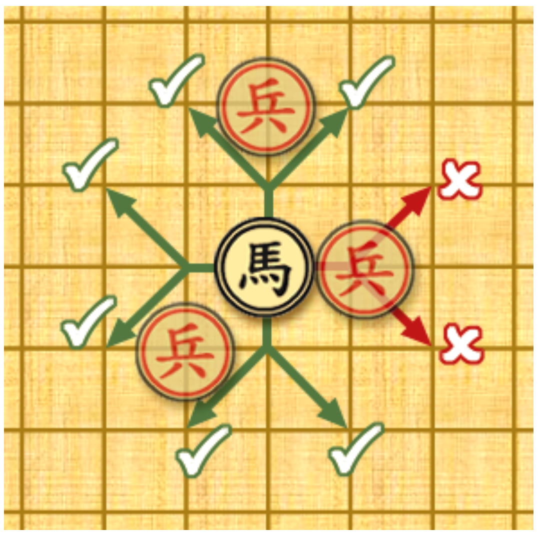
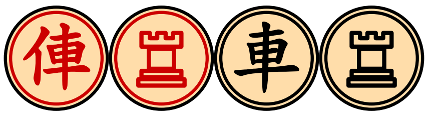
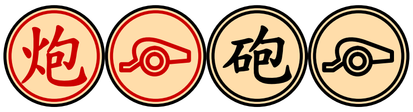


Figure 3: An example of horse move

- Chariot: The chariots are placed at the corners of the board next to the horses. They move any distance orthogonally, but cannot jump over intervening pieces. The behaviour of chariots are identical to the rooks in the Western Chess.



- Cannon: Each player has two cannons, which start on the row behind the soldiers, two points in front of the horses. Cannons move like chariots, i.e., any distance orthogonally without jumping, but can only capture by jumping a single piece of either color along the path of attack. There may be any number of unoccupied spaces between the cannon, the piece over which the cannon jumps and the piece to be captured. An example of how a cannon can capture enemy's pieces can be found in Figure 4.



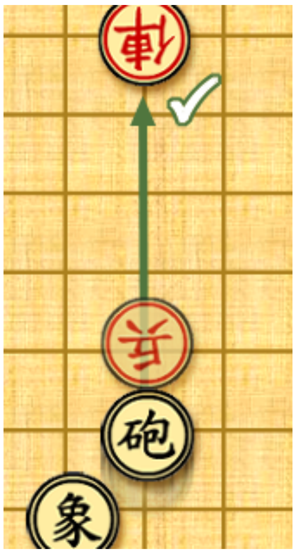
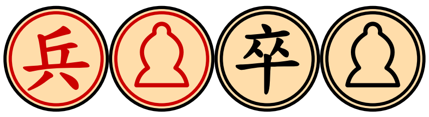


Figure 4: An example of cannon

- Soldier: Each player starts with 5 soldiers, which are located on every other point one row back from the edge of the river. They move by advancing one point forward. Once they have crossed the river, they can also move one point horizontally. However, they can never move backwards.



## 3. Jave Sample Code and Tasks

### Java Sample Code

In the sample Java code, a Chinese Chess board is represented as a 2D array of characters, i.e., `char[9][10]`, where the element with index [0][0] represents the top-left corner of the chess board. Different types of pieces and unoccupied positions are represented as follows (we use uppercase alphabetic letters to represent red pieces and lowercase letters for black pieces):

- G/g : General
- A/a : Advisor
- E/e : Elephant
- H/h : Horse
- R/r : Chariot
- C/c : Cannon
- S/s : Soldier
- . : unoccupied position

The sample Java code consists of three main parts: the chessboard, the pieces and the moves. The chessboard for Chinese Chess is represented as a single Java class called `ChessBoard` which contains all the data and methods that are associated with the game. The followings are a summary of the data and methods declared in `ChessBoard`:

| ***Data*** | ***Description*** |
|---|---|
| `int WIDTH` | An integer constant which represents the width of the board, i.e., the number of vertical lines. |
| `int HEIGHT` | An integer constant which represents the height of the board, i.e., the number of horizontal lines. |
| `char[][] board` | A two-dimensional array for character values which represents the current state of the chess board. |
| `ArrayList<Piece> redPieces` | A list of red pieces. |
| `ArrayList<Piece> blackPieces` | A list of black pieces. |

| ***Method*** | ***Description*** |
|---|---|
| `Piece getPieceAt(int x, int y)` | Given the x-axis and y-axis values, return the piece at the specified position. |

| | |
|---|---|
| `void findAllPieces()` | Find out all the pieces on the current chess board and update the lists of red and black pieces. |
| `boolean validPosition(int x, int y)` | Return true if the given position is within the chessboard, false otherwise. |
| `void printBoard()` | Output the current board to the console window |
| `boolean unoccupied(int x1, int x2, int y1, int y2)` | Given the start position and end position of a vertical or a horizontal line, and check if the line is unoccupied, i.e., there is no piece located on the line. This method works only when x1 == x2, i.e., a vertical line is given, or when y1 == y2, i.e., a horizontal line is given. |
| `boolean checkmate()` | Return true if the black player is checkmated, false otherwise. |

`Piece` in the sample Java code is an abstract class that implements the Moveable interface, and it is the superclass of all the pieces in Chinese Chess. The followings are a summary of the data and methods declared in `Piece`:

| ***Data*** | ***Description*** |
|---|---|
| `int x` | The x-axis value of the current piece |
| `int y` | The y-axis value of the current piece |
| `boolean red` | True, if the piece is red; false, otherwise |

| ***Method*** | ***Description*** |
|---|---|
| `int getX()` | Return the x-axis value of the current piece |
| `Int getY()` | Return the y-axis value of the current piece |
| `boolean isRed()` | Return true if the current piece is red; false, otherwise |
| `boolean riverCrossed()` | Return true, if the piece has already crossed the river; false, otherwise. |
| `boolean atPalace()` | Return true, if the piece is currently at its own palace |
| `boolean sameColor()` | Return true if the given piece has the same color as the current one. |

|  |  |
|---|---|
| `Move[] getMoves(ChessBoard baord)` | As all the pieces implement the `Moveable` interface, the `getMoves` method must be implemented in all the concrete Java classes that represent different types of pieces. This method takes a chess board as its input and returns all possible moves in the given chess board. |

`Move` in the sample Java code is an abstract class which is the superclass of all different types of moves in Chinese Chess. The followings are a summary of the data and methods declared in `Move`:

| ***Data*** | ***Description*** |
|---|---|
| `int x` | The movement distance at x-axis. |
| `int y` | The movement distance at y-axis. |

| ***Method*** | ***Description*** |
|---|---|
| `int getX()` | Return the movement distance at x-axis. |
| `Int getY()` | Return the movement distance at y-axis. |
| `boolean canMove(Piece p, ChessBoard board)` | Given a piece and a chess board, return true if the current movement can be performed by the given piece at the given chess board. |
| `Void printMove()` | Output the current move to the console window. |

## Java Tasks

Given the sample code, you are asked to finish the following tasks in Java:

1. Write class definitions for different types of pieces in Chinese Chess, all of which must be the subclass of `Piece`. Your class definitions need to override the `Move[] getMoves(ChessBoard board)` method which returns a list of potential moves for the current piece in the given chess board. The sample code has already provided class definitions for `Soldier`, `Charoit` and `Guard`, `OMove` for the orthogonal move and `DMove` for the diagonal move. Please complete the class definition for `Cannon`, `Horse`, `Elephant` and `General`. Feel free to modify any of the existing movements or to define new types of moves. However, all the new movements must be inherited from the abstract class `Move`. **[15 Marks]**

2. Complete the pre-defined method `void findAllPieces()` in `ChessBoard`, which searches the current chess board (i.e., board), find out all the pieces exists in the current board, and stored them in two separate Arraylist `redPieces` and `blackPieces`. To compelete this task, you also need to write Java code for the `getPieceAt` method, which return the piece at a specified location. **[10 Marks]**

3. Complete the `boolean riverCrossed()` method in `Piece`, which returns true if the current piece has crossed the river. **[5 Marks]**

4. Complete the `boolean atPalace()` method in `Piece`, which returns true if the piece is currently at its own palace. **[5 Marks]**

5. Complete the pre-defined method `boolean checkmate()` in `ChessBoard`. Assuming it is now the black player's turn, the checkmate method will return true, if the black player is checkmated, i.e., no matter what moves the black player will perform, the red player can surely win the game in its next round. **[15 Marks]**

To test your code, you could rewrite the `AssessedCW2.java` file. There is an example in the sample Java code, where the chessboard is read from a text file. You can also customise your own test cases. Note, you are not allowed to use any additional Java packages other than the ArrayList.

## 4. Haskell Sample Code and Tasks

### Haskell Sample Code

You will be given a Haskell sample file with some code to get you started. The file contains the following data declarations that you need to use to complete the coursework.

| ***Type and data declarations*** | ***Description*** |
|---|---|
| `data PieceType = General \| Guard \| Elephant \| Horse \| Chariot \| Cannon \| Soldier deriving (Eq,Show)` | Represent a particular type of piece. |
| `data PieceColour = Black \| Red deriving (Eq,Show)` | Represent the two colours. |
| `type Piece = (PieceColour, PieceType)` | A particular piece of a colour and type. |
| `data BoardPosition = On Piece \| Empty deriving (Eq,Show)` | A position on the chess board either has a piece on it, or it is empty. |
| `type Board = [[BoardPosition]]` | A chess board is represented as a list of list of board positions. The inner list is one row of 9 positions, and the outer list is the 10 rows that form the board. |
| `type Pos = (Int,Int)` | Represent a position on the chess board as `(x,y)`: `x` indexes rows 0 to 9 from top to bottom, and `y` indexes columns 0 to 8 from left to right. |
| `type Move = (Pos, Pos)` | A move from one position to another. |
| `type Path = [Pos]` | A path as a sequence of positions on the chess board. |

The file contains several useful functions. As part of this coursework, ***it is your responsibility to study these functions*** to determine how they work and how you can use them. However, here are some descriptions of key functions.

| ***Function*** | ***Description*** |
|---|---|
| `emptyBoard :: Board` | An empty chess board with no pieces. |
| `startBoard :: Board` | Initial setting of pieces for Chinese chess. |

| | |
|---|---|
| `strBoard :: Board -> String` | Convert a board into String format for display. Use `putStrLn` to display the board in ghci. It uses the same notation for pieces that is used to represent pieces in the Java section. For example,<br><br>`GHCI> putStrLn (strBoard startBoard)` |
| `baseMoves :: Board -> Piece -> Pos -> [Pos]` | Returns a list of all positions that a given piece can move on a board from a given position, without considering whether the move is valid, or even on the board. For example,<br><br>`baseMoves startBoard (Black, General) (0,4)` |
| `pathMove :: PieceType -> Move -> Path` | Returns a path for a move, excluding the starting and end points. This may be different according to piece type so this is also an argument. For example,<br><br>`pathMove Elephant ( (9,3), (7,5) )` |
| `checkMove :: Board -> PieceType -> Move -> Bool` | Check if a particular move is valid. These are specific validity conditions for each piece; e.g. General must remain in the Palace:<br><br>`checkMove startBoard General ( (0,4), (1,4) )` |
| `validMoves :: Board -> Pos -> [Pos]` | Return a list of valid moves for the piece at a given position on a chess board; e.g.<br><br>`validMoves startBoard (0,0)` |

Note that in the sample file, the functions `baseMoves` and `checkMove` have only been defined for the General, the Soldier and the Chariot. You will be asked to complete for the other pieces as part of your coursework tasks.

## Haskell Tasks

You can use any function from Haskell Prelude, but do not use any functions from any other Haskell library. You may also use functions provided in the Haskell sample file and functions you have written. Please note that to answer some of these questions you may need to write your own auxiliary functions.

When you deliver your solution, please include the data and type declarations and functions from the sample file and any other auxiliary functions you have written, so that your Haskell script compiles when it is loaded into GHCi.

1. Write a function `crossRiver :: Move -> Bool`
   that will return True if and only if a move given as an argument involves crossing the river. For example, `crossRiver ( (4,3), (5,3) )` should return `True`.
   **[5 marks]**

2. Write a function `cannonJump :: Board -> Pos -> (Int,Int) -> [Pos]` to give the cannon jump-and-capture move from the given position in the direction given in the third argument. If such a move exists return the position of the piece taken in a list. If such a move does not exist then return an empty list `[]`. Note that a direction is one of the orthogonal directions (-1,0), (1,0), (0,-1) or (0,1).

   For example, `cannonJump startBoard (2,1) (1,0)` will return `[(9,1)]`, but `cannonJump startBoard (2,1) (0,-1)` will return `[]`. **[5 marks]**

3. Develop the functions `baseMoves` and `checkMove` further so that they work for all possible chess pieces. You can do this by providing new function definitions with patterns for each of the pieces, where required.

   Once you have done this the function `validMoves` should then work for all pieces too, but you should test that it does. **[10 marks]**

4. Write a function `findGeneral :: Board -> PieceColour -> Pos` which finds the position of the General of the given colour. For example,

   ```
   findGeneral startBoard Red
   ```

   will return `(9,4)`. **[5 marks]**

5. Write the function `findAllPieces :: Board -> PieceColour -> [Pos]` which returns a list of the position of all pieces on the chess board of one colour; e.g.

   ```
   findAllPieces startBoard Red
   ```

   will return list of the starting positions of all red pieces. **[10 marks]**

6. Write the function `canCapture :: Board -> PieceColour -> Pos -> [Pos]` which returns the list of positions of any pieces of colour given by the 2nd argument that can capture the piece at the position given by the 3rd argument.

   For example, `canCapture startBoard Red (0,7)` will return `[(7,7)]`.

   **[5 marks]**

7. Write the function `checkmate :: Board -> PieceColour -> Bool` that returns True if and only if the general of the given colour is in checkmate on the given chess board. For example,

   ```
   checkmate startBoard Black
   ```

   will return `False` (the game does not start in checkmate!). **[10 marks]**

The Haskell sample contains several chessboards `b4, b5, b6, b7` that you can use to test your code. However, these are not exhaustive tests and it is strongly recommended that you also construct your own test cases.

## 5. Submission

You should submit two files, one achieve for Java files and one .hs file for your Haskell code. The zip achieves should be named as:

<Your student ID_Java>.zip

<Your student ID_Haskell>.hs

<Your student ID> should be replaced with your student ID number. For example, if your student ID is 20411111, then your zip achieve should be named as 20411111_Java.zip and 20411111_Haskell.hs

For Java submission, you don't need to include the AssessedCW2.java in your submission, as a different file will be used for testing your code.

You should submit your solution via Moodle by the deadline, i.e.,

**6pm, 4th May, 2023.**

### Penalties

The following table illustrate the penalties that apply to this coursework.

| Penalties | Details | Deduction |
|---|---|---|
| Late Submission | If you submit your work after the deadline, you will be penalized according to the standard University penalty | 5% absolute deduction, per day |
| Incorrect Filename | If you submit your work with an incorrect file name. | 10% absolute deduction |
| Incorrect File Format | If you submit your work with an incorrect file format. | 10% absolute deduction |
| Code that does not compile | Your code must compile: in JDK version 8 or higher for Java and ghci version 7 or higher fro haskell. **Code that do not compile could result in 0 marks.** | 0 marks for Java or Haskell code that does not compile |
| Use of other technologies | If you use technologies other than those specified in the assignment brief. | 50% absolute deduction |

## Academic Misconduct

You are reminded that, by submitting your work for assessment, you are declaring that the work is your own. You were already introduced to the University's policy on academic misconduct. You should familiarise yourself with this policy. You should

be aware that the University takes plagiarism very seriously and that your work will be checked for plagiarism. If you are found to have plagiarised, you will be subject to the University's disciplinary procedures.

During the introductory lecture, we provided advice on how to avoid plagiarism, especially in programming-based assignments (such as this one). A summary is provided below:

- You are allowed to discuss the coursework with your classmates. You are not allowed to share your code or your specific approach for solving the task.
- You are allowed to use the Internet to search for information. You should not copy and paste code directly from the Internet. You should include references to any code or resources you have used in completing the assignment.
- You must take action to protect your work from others. Do not share your code or computer with classmates, as this may lead to unexpected problems. If you share a dormitory with other students in the same class, you should take steps to protect your computer. For example, you could use a password to protect your computer. You could also use a USB stick to transfer your work to another computer. You should not use a cloud storage service to store your work.
- You should ask your teacher if you are uncertain as to whether or not you are allowed to do something.
- According to the updated University policies, it is clearly stated that "it is beyond reasonable doubt that AI constitutes academic offence". Therefore, you cannot use ChatGPT or any other AI tool to assist you to complete your own work.