

Project Report

**Baroque Chess:-A heuristic based Game playing environment using
LISP**

Submitted by:

Sukumar Kamalasan and Wei Yan

Dept of Electrical Engineering and Computer Science

University of Toledo

04th May 2001

Baroque Chess in Lisp

1. What is Baroque Chess?

This game is played with conventional chess pieces and the set up, except the King's side rook is turned upside down. The pieces used in Baroque are defined as followed.

Classic Chess	Baroque Chess
Pawn	Squeezer
Knight	Leaper
Bishop	Imitator
(Right-side-up) Rook	Freezer
(Upside-down) Rook	Coordinator
Queen	Step-back
King	King

1.) Rules

a.) Squeezer : It moves like a rook of ordinary chess, that is vertically or horizontally any number of squares. In order to capture an opponent's piece, the Squeezer must be moved so as to "sandwich" the piece between the Squeezer and another Squeezer.

b.) Freezer: Doesn't capture any other pieces, however, when it is adjacent (in any of the eight neighbouring squares) to an opponent's piece, the opponent may not move that piece.

c.) Coordinator: moves like a queen. It coordinates with the same player's King column determined by the location of the square of the board. If the coordinator is moved so as to make this square one where an opponent's piece stands, the piece is captured.

The game is played on a 3 by 3 board with the usual rule that mandates the player who first occupies a row or a column or diagonals with his/her pieces win the game.

d.) Leaper: moves like a Queen, except the when capturing, it must complete its move by jumping over the piece it captures to the next square in the same line; that square must be vacant to permit the capture.

e.) Step-Back: moves (and looks) like a normal chess King, and it captures like a normal chess King (however, there is no "castling" move in Baroque Chess). The game is finished when a King is captured (there is no checkmating or need to say "check" in the Baroque Chess).

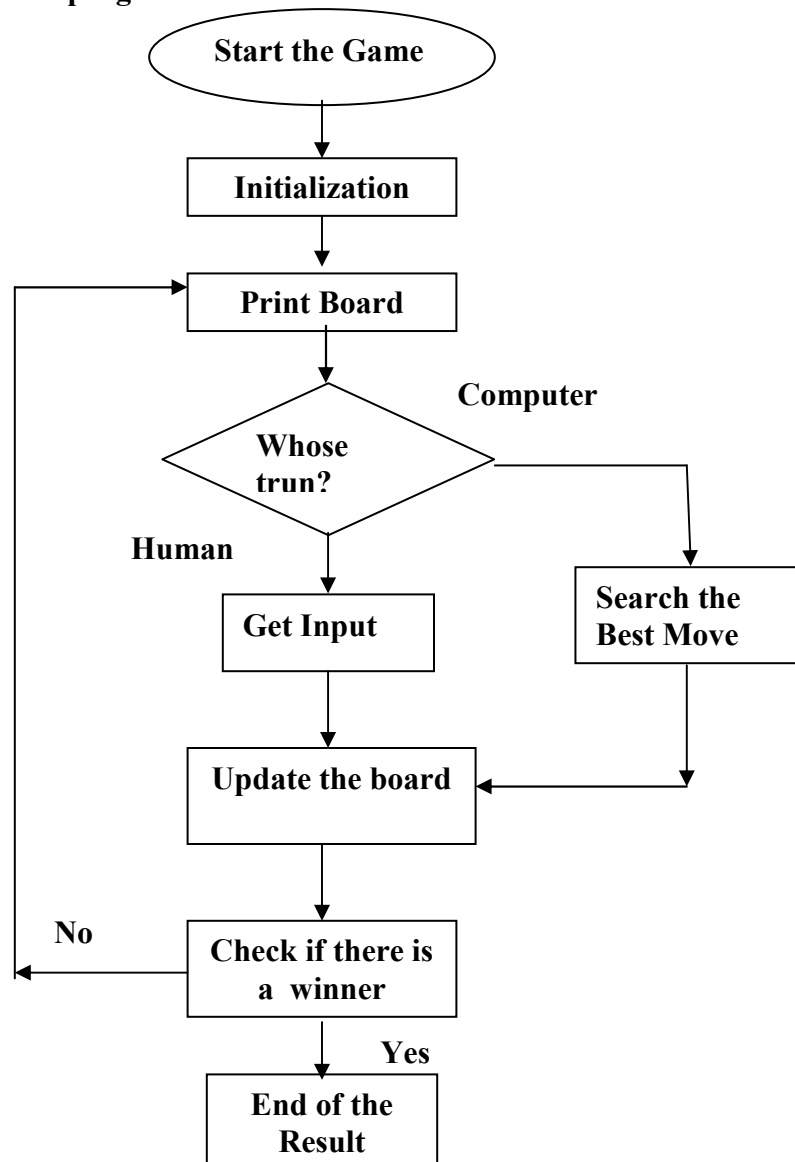
f.) King: Moves (and looks) like a normal chess King, and it captures like a normal chess King (however, there is no "castling" move in Baroque Chess). The game is finished when a King is captured (there is no checkmating or need to say "check" in Baroque Chess).

g.) Imitator: Normally moves like a Queen. However, in order to capture a piece, an Imitator must do as the captured piece would do to capture, In addition, if an Imitator is adjacent to the opponent's Freezer, the Imitator freezes the Freezer, and then neither piece may be moved until one of the two is captured.

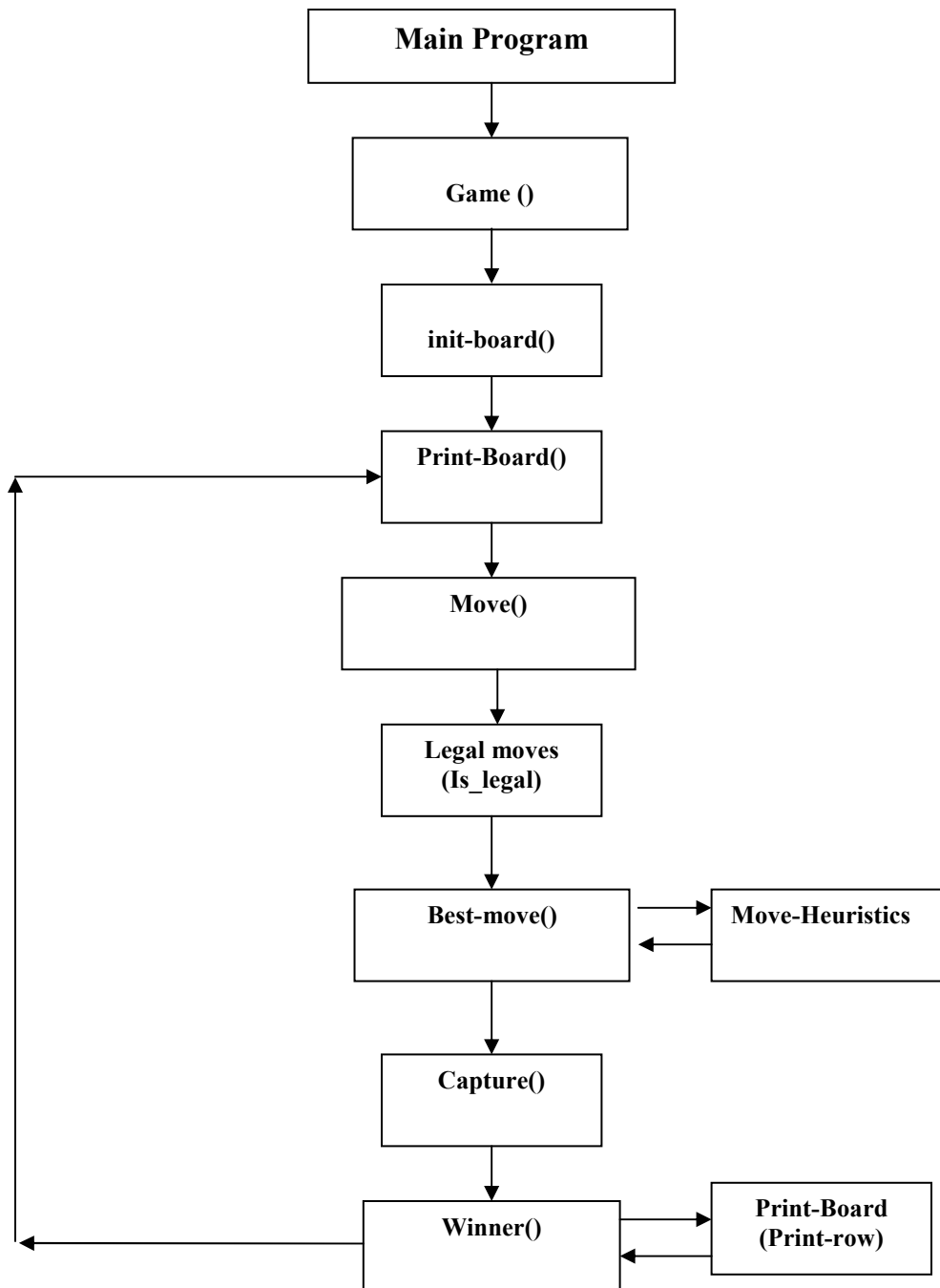
2. Objective of this project

To write a game of Baroque Chess in Lisp which have some sort of intelligent. It is a human to computer game. It should tell if the game is over or not. When it's the human to make a move, it will ask for the input; if it is the computer to make a move, it will perform a searching algorithm to find the next best move. This program should also provide user-friendly interface.

3. Flow chart of the program



4. Corresponding function flow chart



5. Explanation of the functions used in this program

game()

This function is manager of this Baroque Chess game. It is the first function called after you have started a game. This function assumes it is a game between human and computer. It first initializes the board, then it will get the information from the User about whether he wants to move first or not. Then it enters a loop until there is a winner and the game is over. This loop just simply call function **move()** that will be explained later, then it prints the board and switches the player to the other one, then it goes back to the beginning of the loop. When the game is over, it will print the result.

move()

This function takes three arguments: the player, the board and the move-index that counts how many moves have made. If the player is the human, then it will ask for the user input. It will also check if it is legal user input, otherwise it will ask the human player to input again. If the player is computer, then it will call **best-move()** function to find the next best move for computer player. Then it will update the board based on the information it gets. Finally it has to update the information about where the kings are located, which is used to find if the game is over.

best-move()

This is the most important function in this program. First it searches all the possible moves and put them in one list. Then it will perform the alpha beta pruning search. It returns a score that is calculated based on the heuristic function. This function also save the best move in a global variable **square** from which the **move()** function the best move information.

Is Legal()

This function checks whether the move is legal or not. The rules that followed for developing the function is as follows.

1. Imitator, Leaper, Step Back ,Coordinator and freezer can move like a Queen in the chess.
2. The Squeezer can move only in rectangular positions and not diagonal moves.
3. King plays as a chess king

This function calls the sub function for checking empty spaces in all the possible positions. It returns if the space is empty and not blocked by any other pieces. This function calls `emptyspacep` (p stands for Pawn movement), `emptyspace_plu` and `emptyspace_prd`, `emptyspace_q`, `emptyspace_d`, `emptyspace_dd`, `emptyspace_du` where d stands for diagonal, du stands for diagonoal up, dd diagonal down, plu means pawn left

and up and prd means pawn right and down. These separately check all the legal moves by this versatility and modular in nature.

For the legality of movement of the King it calls a function called **legalking** which in turn calculates whether the proposed position is within the adjacent 8 spaces.

capture()

After each move is made, this function checks if any pieces of the opponent are captured. If yes, it will replace that square with blank.

Winner()

This function checks if there is a winner, that is, if the game is over, by just checking the position of the white king and black king. If either of them is a blank, then it returns the winner, otherwise it returns nil. The information of the kings are stored in global variables that are updated after each king's move.

6. Search Algorithm

The search algorithm Considered here is Alpha-beta pruning. It is a technique, which always gives the same answer as brute-force searching without looking at so many nodes of the tree. Intuitively, alpha-beta pruning works by ignoring sub trees, which it knows cannot be reached by best play (on the part of both sides).

Below mentioned search tree shows the general alpha beta pruning.

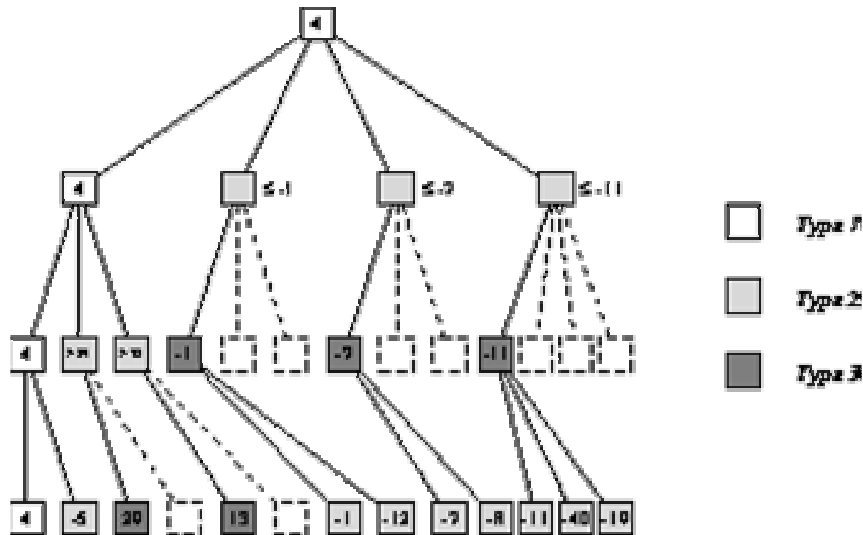


Figure: Pruning of a Perfectly Ordered Tree.

As we know the problem with this brute-force approach is that the size of the tree explodes exponentially. The "branching factor" or number of legal moves in a typical

position is about 35. In order to play master-level chess a search of depth eight appears necessary, which would involve a tree of or about leaf nodes.

The game at present searches 3 level down with alpha beta pruning. The array reference number and the type of the piece classify the nodes. The search will search all the children till that level using the function called best. The best move checks the heuristics function and returns the best possible move in that level. The assigning the score for each of the pieces and getting the value performed does this by the evaluation function.

7. Heuristic Function

The heuristic Function is the total number of pieces in the board at current position. Based on that there are basically two-evaluation function developed.

Evaluation Functions

1. First each number of pieces have been given weightage for any positions in the board. That means the pieces have given same weightage for the positions except for the KING, The KING have got more weightage and infact given a higher number. We call it as '**scores**' for the particular piece.
2. The second heuristics have been the sum of the total number of positional scores of white and black pieces. This determines the **board authority**.

Scores

Each piece on the board is assigned a score that is used in the heuristic function.

King: 10000
Leaper: 200
Imitator: 150
Freezer: 150
Step-back: 150
Squeezer: 100
Coordinator: 100

These scores are just given as this number by playing the game and checking the performance

8. Design and Development phase

Design and Development phase

Phase 1

In the phase one the modules and the possible structures are being developed. In this state initially the functions which we need to structure and the total flowchart of the problem is

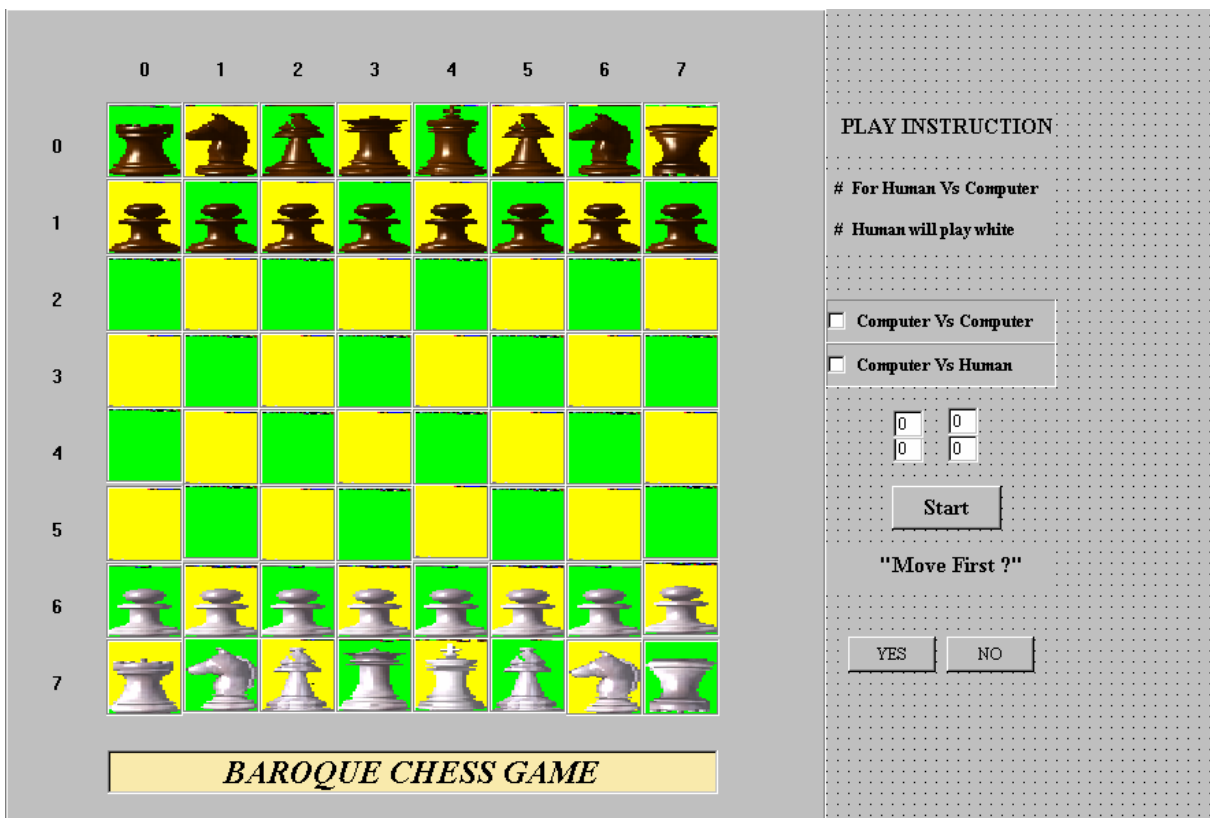
developed. The main functions such as legal move and capture was developed in this phase. Initially the definition of 8*8 board array and the board initialization has developed. Further the print board position function and the supporting function for legal check and the capture move is finalized.

Phase 2

In this phase the heuristics function has been coded. The search algorithm was finalized and the alpha-beta pruning search code is developed. The main top level function for the Computer to human game is coded at this stage and testing for these functions was done.

9. Interface

The current interface we use is a simple text interface. The user has to use the keyboard to make the input. After each move, it will print the board and ask the human player to input if it is his turn. In future work, we would like to change this text interface to graphic interface. The pattern of the Graphic User Interface, which we created, is shown below. It doesn't interact now since further changes in passing the parameters need to be done.



10. Testing and Debugging

Testing is done for basically two levels. First the human to human game without intelligence and then in the second level with intelligence. The board positions of both the levels are shown as a sample here

Certain Moves and Positions

Level 1 Case 1

```
; Loading U:\projmod.lisp
```

```
> (ttt)
```

```
"You play WHITE and computer plays BLACK"
```

```
"Do you want to move first? (y/n) " y
```

```

      0         1         2         3         4         5         6         7
-----+-----+-----+-----+-----+-----+-----+-----
0  B-COOR | B-LEAP | B-IMIT | B-STEP | B-KING | B-IMIT | B-LEAP | B-FREZ
-----+-----+-----+-----+-----+-----+-----+-----
1  B-PAWN | B-PAWN | B-PAWN | B-PAWN | B-PAWN | B-PAWN | B-PAWN | B-PAWN
-----+-----+-----+-----+-----+-----+-----+-----
2  |       |       |       |       |       |       |       |
-----+-----+-----+-----+-----+-----+-----+-----
3  |       |       |       |       |       |       |       |
-----+-----+-----+-----+-----+-----+-----+-----
4  |       |       |       |       |       |       |       |
-----+-----+-----+-----+-----+-----+-----+-----
5  |       |       |       |       |       |       |       |
-----+-----+-----+-----+-----+-----+-----+-----
6  W-PAWN | W-PAWN | W-PAWN | W-PAWN | W-PAWN | W-PAWN | W-PAWN | W-PAWN
-----+-----+-----+-----+-----+-----+-----+-----
7  W-FREZ | W-LEAP | W-IMIT | W-KING | W-STEP | W-IMIT | W-LEAP | W-COOR
-----+-----+-----+-----+-----+-----+-----+-----

```

Please make a move.

```
Input the coordinates of the piece you want to move
and the coordinates of the square you want to move to: 6 0 2 0
```

```

      0         1         2         3         4         5         6         7
-----+-----+-----+-----+-----+-----+-----+-----
0  B-COOR | B-LEAP | B-IMIT | B-STEP | B-KING | B-IMIT | B-LEAP | B-FREZ
-----+-----+-----+-----+-----+-----+-----+-----
1  B-PAWN | B-PAWN | B-PAWN | B-PAWN | B-PAWN | B-PAWN | B-PAWN | B-PAWN
-----+-----+-----+-----+-----+-----+-----+-----
2  W-PAWN |       |       |       |       |       |       |
-----+-----+-----+-----+-----+-----+-----+-----
3  |       |       |       |       |       |       |       |
-----+-----+-----+-----+-----+-----+-----+-----
4  |       |       |       |       |       |       |       |
-----+-----+-----+-----+-----+-----+-----+-----
5  |       |       |       |       |       |       |       |
-----+-----+-----+-----+-----+-----+-----+-----
6  |       | W-PAWN | W-PAWN | W-PAWN | W-PAWN | W-PAWN | W-PAWN | W-PAWN
-----+-----+-----+-----+-----+-----+-----+-----

```

```
7  W-FREZ | W-LEAP | W-IMIT | W-KING | W-STEP | W-IMIT | W-LEAP | W-COOR
-----+
```

```

      0      1      2      3      4      5      6      7
-----+-----+-----+-----+-----+-----+-----+-----+
0  B-COOR | B-LEAP | B-IMIT | B-STEP | B-KING | B-IMIT | B-LEAP | B-FREZ
-----+-----+-----+-----+-----+-----+-----+-----+
1  B-PAWN | B-PAWN | B-PAWN | B-PAWN | B-PAWN |         | B-PAWN | B-PAWN
-----+-----+-----+-----+-----+-----+-----+-----+
2  W-PAWN |         |         |         |         |         |         |         |
-----+-----+-----+-----+-----+-----+-----+-----+
3         |         |         |         |         |         | B-PAWN |         |
-----+-----+-----+-----+-----+-----+-----+-----+
4         |         |         |         |         |         |         |         |
-----+-----+-----+-----+-----+-----+-----+-----+
5         |         |         |         |         |         |         |         |
-----+-----+-----+-----+-----+-----+-----+-----+
6         | W-PAWN | W-PAWN | W-PAWN | W-PAWN | W-PAWN | W-PAWN | W-PAWN
-----+-----+-----+-----+-----+-----+-----+-----+
7  W-FREZ | W-LEAP | W-IMIT | W-KING | W-STEP | W-IMIT | W-LEAP | W-COOR
-----+-----+-----+-----+-----+-----+-----+-----+

```

Please make a move.

Input the coordinates of the piece you want to move
and the coordinates of the square you want to move to: 7 0 3 0

```

      0      1      2      3      4      5      6      7
-----+-----+-----+-----+-----+-----+-----+-----+
0  B-COOR | B-LEAP | B-IMIT | B-STEP | B-KING | B-IMIT | B-LEAP | B-FREZ
-----+-----+-----+-----+-----+-----+-----+-----+
1  B-PAWN | B-PAWN | B-PAWN | B-PAWN | B-PAWN |         | B-PAWN | B-PAWN
-----+-----+-----+-----+-----+-----+-----+-----+
2  W-PAWN |         |         |         |         |         |         |         |
-----+-----+-----+-----+-----+-----+-----+-----+
3  W-FREZ |         |         |         |         |         | B-PAWN |         |
-----+-----+-----+-----+-----+-----+-----+-----+
4         |         |         |         |         |         |         |         |
-----+-----+-----+-----+-----+-----+-----+-----+
5         |         |         |         |         |         |         |         |
-----+-----+-----+-----+-----+-----+-----+-----+
6         | W-PAWN | W-PAWN | W-PAWN | W-PAWN | W-PAWN | W-PAWN | W-PAWN
-----+-----+-----+-----+-----+-----+-----+-----+
7         | W-LEAP | W-IMIT | W-KING | W-STEP | W-IMIT | W-LEAP | W-COOR
-----+-----+-----+-----+-----+-----+-----+-----+

```

"capture!"

```

      0      1      2      3      4      5      6      7
-----+-----+-----+-----+-----+-----+-----+-----+
0  B-COOR | B-LEAP | B-IMIT | B-STEP | B-KING | B-IMIT | B-LEAP | B-FREZ
-----+-----+-----+-----+-----+-----+-----+-----+
1  B-PAWN | B-PAWN | B-PAWN | B-PAWN | B-PAWN |         | B-PAWN | B-PAWN

```

2	W-PAWN							
3	W-FREZ							
4								
5						B-PAWN		
6		W-PAWN		W-PAWN		W-PAWN		W-PAWN
7		W-LEAP		W-IMIT		W-KING		W-STEP
		W-IMIT		W-LEAP		W-COOR		

Please make a move.

Input the coordinates of the piece you want to move

and the coordinates of the square you want to move to: 6 6 4 6

	0	1	2	3	4	5	6	7							
0	B-COOR		B-LEAP		B-IMIT		B-STEP		B-KING		B-IMIT		B-LEAP		B-FREZ
1	B-PAWN		B-PAWN		B-PAWN		B-PAWN		B-PAWN		B-PAWN		B-PAWN		B-PAWN
2	W-PAWN														
3	W-FREZ														
4													W-PAWN		
5										B-PAWN					
6			W-PAWN		W-PAWN		W-PAWN		W-PAWN		W-PAWN				W-PAWN
7			W-LEAP		W-IMIT		W-KING		W-STEP		W-IMIT		W-LEAP		W-COOR

"capture!"

"capture!"

"capture!"

	0	1	2	3	4	5	6	7							
0	B-COOR		B-LEAP		B-IMIT		B-STEP		B-KING		B-IMIT		B-LEAP		B-FREZ
1	B-PAWN		B-PAWN		B-PAWN		B-PAWN		B-PAWN		B-PAWN		B-PAWN		B-PAWN
2	W-PAWN														
3	W-FREZ														
4													W-PAWN		
5													B-PAWN		
6			W-PAWN		W-PAWN		W-PAWN		W-PAWN		W-PAWN				W-PAWN
7			W-LEAP		W-IMIT		W-KING		W-STEP		W-IMIT		W-LEAP		W-COOR

Please make a move.

Input the coordinates of the piece you want to move
and the coordinates of the square you want to move to: 6 5 6 6

"capture!"

	0	1	2	3	4	5	6	7
0	B-COOR	B-LEAP	B-IMIT	B-STEP	B-KING	B-IMIT	B-LEAP	B-FREZ
1	B-PAWN	B-PAWN	B-PAWN	B-PAWN	B-PAWN		B-PAWN	B-PAWN
2	W-PAWN							
3	W-FREZ							
4							W-PAWN	
5								
6		W-PAWN	W-PAWN	W-PAWN	W-PAWN		W-PAWN	W-PAWN
7		W-LEAP	W-IMIT	W-KING	W-STEP	W-IMIT	W-LEAP	W-COOR

	0	1	2	3	4	5	6	7
0	B-COOR	B-LEAP	B-IMIT	B-STEP	B-KING	B-IMIT	B-LEAP	B-FREZ
1	B-PAWN	B-PAWN	B-PAWN	B-PAWN	B-PAWN			B-PAWN
2	W-PAWN							
3	W-FREZ						B-PAWN	
4							W-PAWN	
5								
6		W-PAWN	W-PAWN	W-PAWN	W-PAWN		W-PAWN	W-PAWN
7		W-LEAP	W-IMIT	W-KING	W-STEP	W-IMIT	W-LEAP	W-COOR

Please make a move.

Input the coordinates of the piece you want to move
and the coordinates of the square you want to move to: 3 0 3 5

	0	1	2	3	4	5	6	7
0	B-COOR	B-LEAP	B-IMIT	B-STEP	B-KING	B-IMIT	B-LEAP	B-FREZ
1	B-PAWN	B-PAWN	B-PAWN	B-PAWN	B-PAWN			B-PAWN
2	W-PAWN							

3						W-FREZ	B-PAWN	
4							W-PAWN	
5								
6		W-PAWN	W-PAWN	W-PAWN	W-PAWN		W-PAWN	W-PAWN
7		W-LEAP	W-IMIT	W-KING	W-STEP	W-IMIT	W-LEAP	W-COOR

	0	1	2	3	4	5	6	7
0	B-COOR		B-IMIT	B-STEP	B-KING	B-IMIT	B-LEAP	B-FREZ
1	B-PAWN	B-PAWN	B-PAWN	B-PAWN	B-PAWN			B-PAWN
2	W-PAWN	B-LEAP						
3						W-FREZ	B-PAWN	
4							W-PAWN	
5								
6		W-PAWN	W-PAWN	W-PAWN	W-PAWN		W-PAWN	W-PAWN
7		W-LEAP	W-IMIT	W-KING	W-STEP	W-IMIT	W-LEAP	W-COOR

Please make a move.

Input the coordinates of the piece you want to move

and the coordinates of the square you want to move to: 3 5 1 5

	0	1	2	3	4	5	6	7
0	B-COOR		B-IMIT	B-STEP	B-KING	B-IMIT	B-LEAP	B-FREZ
1	B-PAWN	B-PAWN	B-PAWN	B-PAWN	B-PAWN	W-FREZ		B-PAWN
2	W-PAWN	B-LEAP						
3							B-PAWN	
4							W-PAWN	
5								
6		W-PAWN	W-PAWN	W-PAWN	W-PAWN		W-PAWN	W-PAWN
7		W-LEAP	W-IMIT	W-KING	W-STEP	W-IMIT	W-LEAP	W-COOR

Case 2

(ttt)

"You play WHITE and computer plays BLACK"
 "Do you want to move first? (y/n) " n

	0	1	2	3	4	5	6	7
0	B-COOR	B-LEAP	B-IMIT	B-STEP	B-KING	B-IMIT	B-LEAP	B-FREZ
1	B-PAWN	B-PAWN	B-PAWN	B-PAWN	B-PAWN	B-PAWN	B-PAWN	B-PAWN
2								
3								
4								
5								
6	W-PAWN	W-PAWN	W-PAWN	W-PAWN	W-PAWN	W-PAWN	W-PAWN	W-PAWN
7	W-FREZ	W-LEAP	W-IMIT	W-KING	W-STEP	W-IMIT	W-LEAP	W-COOR

	0	1	2	3	4	5	6	7
0	B-COOR	B-LEAP	B-IMIT	B-STEP	B-KING	B-IMIT	B-LEAP	B-FREZ
1	B-PAWN	B-PAWN	B-PAWN	B-PAWN	B-PAWN		B-PAWN	B-PAWN
2								
3						B-PAWN		
4								
5								
6	W-PAWN	W-PAWN	W-PAWN	W-PAWN	W-PAWN	W-PAWN	W-PAWN	W-PAWN
7	W-FREZ	W-LEAP	W-IMIT	W-KING	W-STEP	W-IMIT	W-LEAP	W-COOR

Please make a move.

Input the coordinates of the piece you want to move

and the coordinates of the square you want to move to: 6 6 3 6

	0	1	2	3	4	5	6	7
0	B-COOR	B-LEAP	B-IMIT	B-STEP	B-KING	B-IMIT	B-LEAP	B-FREZ
1	B-PAWN	B-PAWN	B-PAWN	B-PAWN	B-PAWN		B-PAWN	B-PAWN
2								
3						B-PAWN	W-PAWN	
4								

```

-----+-----+-----+-----+-----+-----+-----+-----
5      |      |      |      |      |      |      |      |
-----+-----+-----+-----+-----+-----+-----+-----
6  W-PAWN | W-PAWN | W-PAWN | W-PAWN | W-PAWN | W-PAWN |      | W-PAWN
-----+-----+-----+-----+-----+-----+-----+-----
7  W-FREZ | W-LEAP | W-IMIT | W-KING | W-STEP | W-IMIT | W-LEAP | W-COOR
-----+-----+-----+-----+-----+-----+-----+-----

```

"capture!"

```

      0      1      2      3      4      5      6      7
-----+-----+-----+-----+-----+-----+-----+-----
0  B-COOR |      | B-IMIT | B-STEP | B-KING | B-IMIT | B-LEAP | B-FREZ
-----+-----+-----+-----+-----+-----+-----+-----
1  B-PAWN | B-PAWN | B-PAWN | B-PAWN | B-PAWN |      | B-PAWN | B-PAWN
-----+-----+-----+-----+-----+-----+-----+-----
2      | B-LEAP |      |      |      |      |      |      |
-----+-----+-----+-----+-----+-----+-----+-----
3      |      |      |      |      |      | B-PAWN | W-PAWN |
-----+-----+-----+-----+-----+-----+-----+-----
4      |      |      |      |      |      |      |      |
-----+-----+-----+-----+-----+-----+-----+-----
5      |      |      |      |      |      |      |      |
-----+-----+-----+-----+-----+-----+-----+-----
6  W-PAWN | W-PAWN | W-PAWN | W-PAWN | W-PAWN | W-PAWN |      | W-PAWN
-----+-----+-----+-----+-----+-----+-----+-----
7  W-FREZ | W-LEAP | W-IMIT | W-KING | W-STEP | W-IMIT | W-LEAP | W-COOR
-----+-----+-----+-----+-----+-----+-----+-----

```

Please make a move.

Input the coordinates of the piece you want to move
and the coordinates of the square you want to move to: 6 4 3 4

"capture!"

```

      0      1      2      3      4      5      6      7
-----+-----+-----+-----+-----+-----+-----+-----
0  B-COOR |      | B-IMIT | B-STEP | B-KING | B-IMIT | B-LEAP | B-FREZ
-----+-----+-----+-----+-----+-----+-----+-----
1  B-PAWN | B-PAWN | B-PAWN | B-PAWN | B-PAWN |      | B-PAWN | B-PAWN
-----+-----+-----+-----+-----+-----+-----+-----
2      | B-LEAP |      |      |      |      |      |      |
-----+-----+-----+-----+-----+-----+-----+-----
3      |      |      |      |      | W-PAWN |      | W-PAWN |
-----+-----+-----+-----+-----+-----+-----+-----
4      |      |      |      |      |      |      |      |
-----+-----+-----+-----+-----+-----+-----+-----
5      |      |      |      |      |      |      |      |
-----+-----+-----+-----+-----+-----+-----+-----
6  W-PAWN | W-PAWN | W-PAWN | W-PAWN |      | W-PAWN |      | W-PAWN
-----+-----+-----+-----+-----+-----+-----+-----
7  W-FREZ | W-LEAP | W-IMIT | W-KING | W-STEP | W-IMIT | W-LEAP | W-COOR
-----+-----+-----+-----+-----+-----+-----+-----

```

"capture!"

"capture!"

```

      0      1      2      3      4      5      6      7
-----+-----+-----+-----+-----+-----+-----+-----
0  B-COOR |      | B-IMIT | B-STEP | B-KING | B-IMIT | B-LEAP | B-FREZ

```

1	B-PAWN	B-PAWN	B-PAWN	B-PAWN	B-PAWN		B-PAWN	B-PAWN
2								
3					W-PAWN		W-PAWN	
4								
5					B-LEAP			
6	W-PAWN	W-PAWN	W-PAWN	W-PAWN		W-PAWN		W-PAWN
7	W-FREZ	W-LEAP	W-IMIT	W-KING	W-STEP	W-IMIT	W-LEAP	W-COOR

Please make a move.

Input the coordinates of the piece you want to move
and the coordinates of the square you want to move to: 6 5 6 4

	0	1	2	3	4	5	6	7
0	B-COOR		B-IMIT	B-STEP	B-KING	B-IMIT	B-LEAP	B-FREZ
1	B-PAWN	B-PAWN	B-PAWN	B-PAWN	B-PAWN		B-PAWN	B-PAWN
2								
3					W-PAWN		W-PAWN	
4								
5					B-LEAP			
6	W-PAWN	W-PAWN	W-PAWN	W-PAWN	W-PAWN			W-PAWN
7	W-FREZ	W-LEAP	W-IMIT	W-KING	W-STEP	W-IMIT	W-LEAP	W-COOR

"capture!"

	0	1	2	3	4	5	6	7
0	B-COOR		B-IMIT	B-STEP	B-KING	B-IMIT	B-LEAP	B-FREZ
1	B-PAWN	B-PAWN	B-PAWN	B-PAWN	B-PAWN		B-PAWN	B-PAWN
2								
3					W-PAWN		W-PAWN	
4								
5								
6	W-PAWN	W-PAWN	W-PAWN	W-PAWN	W-PAWN	B-LEAP		W-PAWN
7	W-FREZ	W-LEAP	W-IMIT	W-KING	W-STEP	W-IMIT	W-LEAP	W-COOR

Level 2 Case 1

(bchess)

"You play WHITE and computer plays BLACK"

"Do you want to move first? (y/n) " y

	0	1	2	3	4	5	6	7
0	B-COOR	B-LEAP	B-IMIT	B-STEP	B-KING	B-IMIT	B-LEAP	B-FREZ
1	B-PAWN	B-PAWN	B-PAWN	B-PAWN	B-PAWN	B-PAWN	B-PAWN	B-PAWN
2								
3								
4								
5								
6	W-PAWN	W-PAWN	W-PAWN	W-PAWN	W-PAWN	W-PAWN	W-PAWN	W-PAWN
7	W-FREZ	W-LEAP	W-IMIT	W-KING	W-STEP	W-IMIT	W-LEAP	W-COOR

Please make a move.

Input the coordinates of the piece you want to move

and the coordinates of the square you want to move to: 6 2 2 2

	0	1	2	3	4	5	6	7
0	B-COOR	B-LEAP	B-IMIT	B-STEP	B-KING	B-IMIT	B-LEAP	B-FREZ
1	B-PAWN	B-PAWN	B-PAWN	B-PAWN	B-PAWN	B-PAWN	B-PAWN	B-PAWN
2			W-PAWN					
3								
4								
5								
6	W-PAWN	W-PAWN		W-PAWN	W-PAWN	W-PAWN	W-PAWN	W-PAWN
7	W-FREZ	W-LEAP	W-IMIT	W-KING	W-STEP	W-IMIT	W-LEAP	W-COOR

	0	1	2	3	4	5	6	7
0	B-COOR	B-LEAP	B-IMIT	B-STEP	B-KING	B-IMIT	B-LEAP	B-FREZ
1	B-PAWN	B-PAWN	B-PAWN	B-PAWN	B-PAWN		B-PAWN	B-PAWN
2			W-PAWN					

3						B-PAWN		
4								
5								
6	W-PAWN	W-PAWN		W-PAWN	W-PAWN	W-PAWN	W-PAWN	W-PAWN
7	W-FREZ	W-LEAP	W-IMIT	W-KING	W-STEP	W-IMIT	W-LEAP	W-COOR

Please make a move.

Input the coordinates of the piece you want to move
and the coordinates of the square you want to move to: 6 3 3 3

	0	1	2	3	4	5	6	7
0	B-COOR	B-LEAP	B-IMIT	B-STEP	B-KING	B-IMIT	B-LEAP	B-FREZ
1	B-PAWN	B-PAWN	B-PAWN	B-PAWN	B-PAWN		B-PAWN	B-PAWN
2			W-PAWN					
3				W-PAWN		B-PAWN		
4								
5								
6	W-PAWN	W-PAWN			W-PAWN	W-PAWN	W-PAWN	W-PAWN
7	W-FREZ	W-LEAP	W-IMIT	W-KING	W-STEP	W-IMIT	W-LEAP	W-COOR

"capture!"
"capture!"

	0	1	2	3	4	5	6	7
0	B-COOR	B-LEAP	B-IMIT	B-STEP	B-KING	B-IMIT	B-LEAP	B-FREZ
1	B-PAWN		B-PAWN	B-PAWN	B-PAWN		B-PAWN	B-PAWN
2		B-PAWN	W-PAWN					
3				W-PAWN		B-PAWN		
4								
5								
6	W-PAWN	W-PAWN			W-PAWN	W-PAWN	W-PAWN	W-PAWN
7	W-FREZ	W-LEAP	W-IMIT	W-KING	W-STEP	W-IMIT	W-LEAP	W-COOR

11. Testing Results and Analysis

The game has developed intelligence and was able to perform. It was found that in certain position it was not performing as desired. This we think may be because of the heuristic which have been adopted. The results shows that certain game positions as can be seen from the dribble section that its not good moves. In general the game has achieved good heuristics.

12. Critiques and Comments

It was found that the Intelligence level is not as expected for the game somehow. It seems the whole game ability is dependent on the intelligence level when compared with the other part of the code. The heuristics function compactness needs to be assessed. Several bugs were fixed and in many place the code is optimized.

13. Achievements

1. Implementation of the Baroque Chess using text interface.
2. Implementation of the Heuristics and making the game more intelligent.
3. Developing a text based interface format for the game
4. Development of the GUI skeleton.
5. Performance of the Search Algorithm

14. Conclusion

1. The implementation of the LISP program for baroque chess has been performed.
2. The testing and the debugging phase was really challenging due to the complexity of the code and the time.
3. The game was able to perform to an acceptable level though we can more room for improving the intelligence.

15. Further Improvements and Challenges

1. The development of Graphical User Interface to make it fully functional gives user-friendlier interaction for the game. This can include even driven programming and the features for mouse
2. Development of different type of heuristics and coordinating them will definitely give optimal and more intelligent Game.
3. Optimization of the code is necessary requirement, which needs to be done especially for search algorithm.
4. The speed of the execution and the interaction will be another factor which needs to be considered

Pledge

“ Submitted work as documented is my original contribution except those sections which are properly cited in the report”

Sukumar Kamalasan

Wei Yan

Date:

Date:

Signature:

Signature:

Contribution

Project Conceptualisation

- Sukumar and Wei

Design of the framework and functions

- Sukumar Wei

Code Development

Functions

1. init-board print-board

- Sukumar and Wei

2. isLegal

- Sukumar

3. capture

- Wei

4. Best move

- Wei and Sukumar

5. Move

- Wei

6. Game

- Sukumar

Testing and debugging

- Respective functions

Main

- Wei and Sukumar

Presentation Slides

- Sukumar and Wei

Report

- Sukumar and Wei

Appendix I

LISP Code with Comments

:: The following code is written by Sukumar and Wei Yan

```
(defun bchess ()
```

```
;;-----
```

```
:: define a 8 by 8 array to represent the board that is a 8 by 8 square board
```

```
(setq *b-king-col* 4)
(setq *b-king-row* 0)
(setq *w-king-col* 3)
(setq *w-king-row* 7)
(setq *square* nil)
```

```
(setq *board* (make-array '(8 8)))
```

```
(defun set-board (x y piece board)
```

```
(setf (aref board x y) piece))
```

```
(defun get-board (x y board)
```

```
(aref board x y))
```

```
(defun copy-board (board)
```

```
(setq new-board (make-array '(8 8)))
(dotimes (x 8)
  (dotimes (y 8)
    (set-board x y (get-board x y board) new-board)))
new-board)
```

```
;;--- initialize a board to start a game -----
```

```

(defun init-board (board)

  (set-board 0 0 'b-coor board)(set-board 0 1 'b-leap board)(set-board 0 2 'b-imit board)(set-board 0 3 'b-
step board)

  (set-board 0 4 'b-king board)(set-board 0 5 'b-imit board)(set-board 0 6 'b-leap board)(set-board 0 7 'b-
frez board)

  (set-board 1 0 'b-pawn board)(set-board 1 1 'b-pawn board)(set-board 1 2 'b-pawn board)(set-board 1 3
'b-pawn board)

  (set-board 1 4 'b-pawn board)(set-board 1 5 'b-pawn board)(set-board 1 6 'b-pawn board)(set-board 1 7
'b-pawn board)

  (set-board 2 0 #\Space board)(set-board 2 1 #\Space board)(set-board 2 2 #\Space board)(set-board 2 3
#\Space board)

  (set-board 2 4 #\Space board)(set-board 2 5 #\Space board)(set-board 2 6 #\Space board)(set-board 2 7
#\Space board)

  (set-board 3 0 #\Space board)(set-board 3 1 #\Space board)(set-board 3 2 #\Space board)(set-board 3 3
#\Space board)

  (set-board 3 4 #\Space board)(set-board 3 5 #\Space board)(set-board 3 6 #\Space board)(set-board 3 7
#\Space board)

  (set-board 4 0 #\Space board)(set-board 4 1 #\Space board)(set-board 4 2 #\Space board)(set-board 4 3
#\Space board)

  (set-board 4 4 #\Space board)(set-board 4 5 #\Space board)(set-board 4 6 #\Space board)(set-board 4 7
#\Space board)
  (set-board 5 0 #\Space board)(set-board 5 1 #\Space board)(set-board 5 2 #\Space board)(set-board 5 3
#\Space board)

  (set-board 5 4 #\Space board)(set-board 5 5 #\Space board)(set-board 5 6 #\Space board)(set-board 5 7
#\Space board)

  (set-board 6 0 'w-pawn board)(set-board 6 1 'w-pawn board)(set-board 6 2 'w-pawn board)(set-board 6 3
'w-pawn board)

  (set-board 6 4 'w-pawn board)(set-board 6 5 'w-pawn board)(set-board 6 6 'w-pawn board)(set-board 6 7
'w-pawn board)

  (set-board 7 0 'w-frez board)(set-board 7 1 'w-leap board)(set-board 7 2 'w-imit board)(set-board 7 3 'w-
king board)

  (set-board 7 4 'w-step board)(set-board 7 5 'w-imit board)(set-board 7 6 'w-leap board)(set-board 7 7 'w-
coor board))

;;---get whatever from one square on the board, if there is a piece, then return ----
;;---that piece, otherwise returns whitespace string ---

(defun get-piece (x)

```

```

(if (not (equal #\space x)) x (string " ")))

;---define a function to print one row of the chess board in text style ----

(defun print-row (index board)

  (format t "~S ~D | ~D | ~D | ~D | ~D | ~D | ~D ~%" index

    (get-piece (get-board index 0 board)) (get-piece (get-board index 1 board))
    (get-piece (get-board index 2 board)) (get-piece (get-board index 3 board))
    (get-piece (get-board index 4 board)) (get-piece (get-board index 5 board))

    (get-piece (get-board index 6 board)) (get-piece (get-board index 7 board)))

  (format t " -----+-----+-----+-----+-----+-----+-----+-----~%"))

;--- print the whole chess board in text style -----

(defun print-board (board)
  (format t "~%~%  0    1    2    3    4    5    6    7  ~%~%"

    (format t " -----+-----+-----+-----+-----+-----+-----+-----~%"

      (dotimes (i 8)

        (print-row i board))
      (format t "~%"))

    (format t " ~%~%"))

;---defining whether the move is legal or not depends on the legal----

;---move procedure for each piece----

(defun isLegal (c1 c2 c3 c4 piece board)

  ;--- checks the empty space in between the current and future position----
  ;--- for rectangular movement----

  (defun emptyspacep ()
    (cond
      ((or (> yc yf) (> xc xf))
       (emptyspace_plu))
      ((or (> yf yc) (> xf xc))
       (emptyspace_prd))))

  ;--- checks the empty space in between the current and future position---
  ;--- in the right and down direction and rectangular

  (defun emptyspace_prd ()
    (cond
      ((= xc xf)
       (setq diff (abs (- yc yf)))
       (setq greater yf)

```

```
(setq lesser yc)
(setq emptys '())
(dotimes (i diff)
  (setq emptys1 (get-board xc (+ lesser (+ i 1)) board))
  (if (not (equal emptys1 #\Space))
      (setq emptys (append emptys (list emptys1))))))
(= yc yf)
(setq diff (abs (- xc xf)))
(setq greater xf)
(setq lesser xc)
(setq emptys '())
(dotimes (i diff)
  (setq emptys1 (get-board (+ lesser (+ i 1)) yc board))
  (if (not (equal emptys1 #\Space))
      (setq emptys (append emptys (list emptys1))))))
```

;;---- checks the empty space in between the current and future position---
 ;;---- in the up and left direction and rectangular

```
(defun emptyspace_plu ()
  (cond
    ((= xc xf)
     (setq diff (abs (- yc yf)))
     (setq greater yc)
     (setq lesser yf)
     (setq emptys '())
     (dotimes (i (- diff 1))
       (setq emptys1 (get-board xc (+ lesser i) board))
       (if (not (equal emptys1 #\Space))
           (setq emptys (append emptys (list emptys1))))))
     (= yc yf)
     (setq diff (abs (- xc xf)))
     (setq greater xc)
     (setq lesser xf)
     (setq emptys '())
     (dotimes (i (- diff 1))
       (setq emptys1 (get-board (+ lesser i) yc board))
       (if (not (equal emptys1 #\Space))
           (setq emptys (append emptys (list emptys1))))))
```

;;---- checks the empty space in between the current and future position----
 ;;---- queen movement----

```
(defun emptyspaceq ()
  (cond
    ((or (= xc xf) (= yc yf))
     (emptyspacep))
    ((= (abs (- xc xf)) (abs (- yc yc)))
     (emptyspaced))))
```

;;---- checks the empty space in between the current and future position----
 ;;---- for diagonal movement----

```
(defun emptyspaced ()
  (cond
    ((and (> xc xf) (or (> yf yc) (> yc yf))))
```

```
(emptyspace_du)
((and (> xf xc) (or (> yf yc) (> yc yf)))
(emptyspace_dd)))
```

;;---- checks the empty space in between the current and future position---

;;---- in the up direction and diagonal

```
(defun emptyspace_du ()
  (if (and (not (equal xc xf)) (not (equal yc yf)))
      (setq diff (abs (- xc xf))))
      (cond
        ((and (> xc xf) (> yc yf))
         (setq xl xc)
         (setq xs xf)
         (setq yl yc)
         (setq ys yf)
         (setq emptys ())
         (dotimes (i (- diff 1))
           (setq emptys1 (get-board (+ xs i) (+ ys i) board))
           (if (not (equal emptys1 #\Space))
               (setq emptys (append emptys (list emptys1))))))
         ((and (> xc xf) (< yc yf))
          (setq xl xc)
          (setq xs xf)
          (setq yl yf)
          (setq ys yc)
          (setq emptys '())
          (dotimes (i (- diff 1))
            (setq emptys1 (get-board (+ xs i) (- yl i) board))
            (if (not (equal emptys1 #\Space))
                (setq emptys (append emptys (list emptys1)))))))))
```

;;---- checks the empty space in between the current and future position---

;;---- in the down direction and diagonal

```
(defun emptyspace_dd ()
  (if (and (not (equal xc xf)) (not (equal yc yf)))
      (setq diff (abs (- xc xf))))
      (cond
        ((and (> xf xc) (> yf yc))
         (setq xl xf)
         (setq xs xc)
         (setq yl yf)
         (setq ys yc)
         (setq emptys '())
         (dotimes (i diff)
           (setq emptys1 (get-board (+ xs (+ i 1)) (+ ys (+ i 1))) board))
           (if (not (equal emptys1 #\Space))
               (setq emptys (append emptys (list emptys1))))))
        ((and (< xc xf) (> yc yf))
         (setq xl xc)
         (setq xs xf)
         (setq yl yf)
         (setq ys yc)
         (setq emptys '())
         (dotimes (i diff)
           (setq emptys1 (get-board (+ xs (+ i 1)) (- yl (+ i 1))) board))
           (if (not (equal emptys1 #\Space))
```

```

(setq emptys (append emptys (list emptys1))))))

;;----checks the legal move position for king----
(defun legalking ()
  (cond ((and (equal xc xf) (not (equal yf yc)))
    (if (or (equal yf (+ yc 1)) (equal yf (- yc 1))) t nil)
    )
    ((and (equal yc yf) (not (equal xf xc)))
    (if (or (equal xf (+ xc 1)) (equal xf (- xc 1))) t nil)
    )
    ((and (not (equal xc xf)) (not (equal yc yf)))
    (if (or (and (equal xf (+ xc 1)) (equal yf (- yc 1)))
      (and (equal xf (- xc 1)) (equal yf (+ yc 1)))
      (and (equal xf (- xc 1)) (equal yf (- yc 1)))
      (and (equal xf (+ yc 1)) (equal yf (+ yc 1))) ) t nil)
    )
  ) )

(setq xc c1) (setq yc c2) (setq xf c3) (setq yf c4)(setq emptys1 #\Space)
(cond
  ((or (equal piece 'b-pawn) (equal piece 'w-pawn))
    (emptyspacep)
    (if (and (or (= xc xf) (= yc yf))
      (equal emptys '() ) t nil))
    ((or (equal piece 'b-frez) (equal piece 'w-frez)
      (equal piece 'b-imit) (equal piece 'w-imit)
      (equal piece 'b-step) (equal piece 'w-step)
      (equal piece 'b-coor) (equal piece 'w-coor))
    (emptyspaceq)
    (if (and (or (= xc xf) (= yc yf) (= (abs (- xc xf)) (abs (- yc yf)))) (equal (get-board xf yf board)
#\Space)
      (equal emptys '() ) t nil))
    ((or (equal piece 'b-leap) (equal piece 'w-leap))
    (setq emptys '())
    (if (and (or (= xc xf) (= yc yf) (= (abs (- xc xf)) (abs (- yc yf)))) (equal (get-board xf yf board)
#\Space)
      (equal emptys '() ) t nil))
    ((and (or (equal piece 'b-king) (equal piece 'w-king)) (equal (get-board xf yf board) #\space))
    (legalking)
    ))
  )
)

;;-----
;; return true if a piece is the other player's. And the 2 parameters are string type
(defun isEnemy (myPiece hisPiece)
  (cond ((and (equal #\B (char myPiece 0)) (equal #\W (char hisPiece 0))) t)
    ((and (equal #\W (char myPiece 0)) (equal #\B (char hisPiece 0))) t)
    (t nil)))

;;-----
;; after each move, check if any opponent's piece is captured, if yes, erase it -----
(defun capture (x_1 y_1 x_2 y_2 piece board)

;;----- squeezer-capture -----
(defun squeezer-capture (x1 y1 x2 y2 s-piece)
  (when (> x2 1)
    (let ((str1 (string (get-piece (get-board (- x2 1) y2 board))))
      (str2 (string (get-piece (get-board (- x2 2) y2 board))))
    )
  )

```

```

(when (and (equal #P (char str2 2)) (not (isEnemy s-piece str2)))
  (if (equal #I (char s-piece 2))
    (when (and (equal #P (char str1 2)) (isEnemy str1 s-piece))
      (print "capture!")
      (set-board (- x2 1) y2 #\Space board)
      str1)
    (when (isEnemy str1 s-piece)
      (print "capture!")
      (set-board (- x2 1) y2 #\Space board)
      str1))))))
(when (< x2 6)
  (let ((str1 (string (get-piece (get-board (+ x2 1) y2 board))))
        (str2 (string (get-piece (get-board (+ x2 2) y2 board))))))
    (when (and (equal #P (char str2 2)) (not (isEnemy s-piece str2)))
      (if (equal #I (char s-piece 2))
        (when (and (equal #P (char str1 2)) (isEnemy str1 s-piece))
          (print "capture!")
          (set-board (+ x2 1) y2 #\Space board)
          str1)
        (when (isEnemy str1 s-piece)
          (print "capture!")
          (set-board (+ x2 1) y2 #\Space board)
          str1))))))
(when (> y2 1)
  (let ((str1 (string (get-piece (get-board x2 (- y2 1) board))))
        (str2 (string (get-piece (get-board x2 (- y2 2) board))))))
    (when (and (equal #P (char str2 2)) (not (isEnemy s-piece str2)))
      (if (equal #I (char s-piece 2))
        (when (and (equal #P (char str1 2)) (isEnemy str1 s-piece))
          (print "capture!")
          (set-board x2 (- y2 1) #\Space board)
          str1)
        (when (isEnemy str1 s-piece)
          (print "capture!")
          (set-board x2 (- y2 1) #\Space board)
          str1))))))
(when (< y2 6)
  (let ((str1 (string (get-piece (get-board x2 (+ y2 1) board))))
        (str2 (string (get-piece (get-board x2 (+ y2 2) board))))))
    (when (and (equal #P (char str2 2)) (not (isEnemy s-piece str2)))
      (if (equal #I (char s-piece 2))
        (when (and (equal #P (char str1 2)) (isEnemy str1 s-piece))
          (print "capture!")
          (set-board x2 (+ y2 1) #\Space board)
          str1)
        (when (isEnemy str1 s-piece)
          (print "capture!")
          (set-board x2 (+ y2 1) #\Space board)
          str1))))))
;----- king-capture -----
(defun king-capture (x1 y1 x2 y2 s-piece)
  (let ((str (string (get-piece (get-board x2 y2 board))))
        (if (equal #I (char s-piece 2))
          (when (and (equal #K (char str 2)) (isEnemy s-piece str))
            (print "capture!"))
          (print "capture!"))))

```

```

    (set-board x2 y2 #\Space board)
    str)
  (when (isEnemy s-piece str)
    (print "capture!")
    (set-board x2 y2 #\Space board)
    str))))

;;----- leaper-capture -----
(defun leaper-capture (x1 y1 x2 y2 s-piece)
  (let ((x3 (if (= x2 x1) x2 (- x2 (/ (- x2 x1) (abs (- x2 x1))))))
        (y3 (if (= y2 y1) y2 (- y2 (/ (- y2 y1) (abs (- y2 y1))))))
        (str ((str (string (get-piece (get-board x3 y3 board))))
                (if (equal #I (char s-piece 2))
                    (when (and (equal #L (char str 2)) (isEnemy s-piece str))
                      (print "capture!")
                      (set-board x3 y3 #\Space board)
                      str)
                    (when (isEnemy s-piece str)
                      (print "capture!")
                      (set-board x3 y3 #\Space board)
                      str)))))))

;;----- step-back-capture -----
(defun step-back-capture (x1 y1 x2 y2 s-piece)
  (when (and (> x1 0) (> y1 0) (< x1 7) (< y1 7))
    (let ((x3 (if (= x2 x1) x2 (- x1 (/ (- x2 x1) (abs (- x2 x1))))))
          (y3 (if (= y2 y1) y2 (- y1 (/ (- y2 y1) (abs (- y2 y1))))))
          (str ((str (string (get-piece (get-board x3 y3 board))))
                  (if (equal #I (char s-piece 2))
                      (when (and (equal #S (char str 2)) (isEnemy s-piece str))
                        (print "capture!")
                        (set-board x3 y3 #\Space board)
                        str)
                      (when (isEnemy s-piece str)
                        (print "capture!")
                        (set-board x3 y3 #\Space board)
                        str)))))))

;;----- coordinator-capture -----
(defun coordinator-capture (x1 y1 x2 y2 s-piece)
  (let ((y3 (if (equal #B (char s-piece 0)) *b-king-col* *w-king-col*)))
    (let ((str (string (get-piece (get-board x2 y3 board))))
          (if (equal #I (char s-piece 2))
              (when (and (equal #C (char str 2)) (isEnemy s-piece str))
                (print "capture!")
                (set-board x2 y3 #\Space board)
                str)
              (when (isEnemy str s-piece)
                (print "capture!")
                (set-board x2 y3 #\Space board)
                str))))))

;;--- check if any piece is captured by calling the functions above ----

(let ((str-piece (string piece)))
  (cond ((equal #P (char str-piece 2))

```



```

      (equal (char victim-piece 2) 'F))
      (setq score (+ score 250)))
      ((or (equal (char victim-piece 2) 'S)
           (equal (char victim-piece 2) 'C))
          (setq score (+ score 100))))
      (cond ((equal (char victim-piece 2) 'L)
             (setq score (- score 400)))
            ((or (equal (char victim-piece 2) 'I)
                 (equal (char victim-piece 2) 'P)
                 (equal (char victim-piece 2) 'F))
             (setq score (- score 250)))
            ((or (equal (char victim-piece 2) 'S)
                 (equal (char victim-piece 2) 'C))
             (setq score (- score 100))))))

(t (if (equal victim-piece nil)
      (setq score (best-move board (+ move-index 1) (other player) a b
                                (+ search-level 1) score))
      (if (equal (char victim-piece 0) 'W)
          (setq score (best-move board (+ move-index 1) (other player) a b
                                    (+ search-level 1) (+ score 100)))
          (setq score (best-move board (+ move-index 1) (other player) a b
                                    (+ search-level 1) (- score 100))))))))

;;--- perform alpha-beta pruning -----

(if (equal 'b player)
    (cond ((>= score b)
           (setq *square* (list x1 y1 x2 y2))
           (setq return-flag t)
           (> score a)
           (setq a score)
           (setq best-square (list x1 y1 x2 y2))))
      (cond ((<= score a)
             (setq *square* (list x1 y1 x2 y2))
             (setq return-flag t)
             (< score b)
             (setq b score)
             (setq best-square (list x1 y1 x2 y2))))))

;;----if return-flag is set, then return score -----
;;----otherwise, alpha or beta will be returned -----
(cond (return-flag score)
      (t (setq *square* best-square)
         (if (equal player 'b) a b))))

;;----- move a piece -----
(defun move (player move-index board)

  (cond ((and (equal player 'b) (or (= move-index 1) (= move-index 2)))

        (set-board 3 5 'b-pawn board)
        (set-board 1 5 #\Space board))

        ((equal player 'w)

```

```

(loop
  (format t "Please make a move. ~%" )
  (format t " Input the coordinates of the piece you want to move ~%" )
  (format t " and the coordinates of the square you want to move to: ")
  (let ((x1 (Read)) (y1 (read)) (x2 (read)) (y2 (Read)))
    (let ((piece (string (get-piece (get-board x1 y1 board)))))
      (if (equal #B (char piece 0)) "you got a wrong piece"
          (cond ((isLegal x1 y1 x2 y2 (get-board x1 y1 board) board)
                  (capture x1 y1 x2 y2 piece board)
                  (set-board x2 y2 (get-board x1 y1 board) board)
                  (set-board x1 y1 #\Space board)
                  (when (equal "W-KING" piece)
                    (setq *w-king-row* x2)
                    (setq *w-king-col* y2))
                  (return))
                (t "Illegal Move!"))))))))

((equal player 'b)
 (let ((new-board (copy-board board)))
   (best-move new-board move-index 'b -100000 100000 0 0)
   (let ((piece (get-board (first *square*) (second *square*) board))
         (x1 (first *square*))
         (y1 (second *square*))
         (x2 (third *square*))
         (y2 (fourth *square*)))
     (when (equal piece 'b-king)
       (setq *b-king-row* x2)
       (setq *b-king-col* y2))
     (capture x1 y1 x2 y2 (string piece) board)
     (set-board x2 y2 (get-board x1 y1 board) board)
     (set-board x1 y1 #\Space board))))))

;;-----
;; return the other player
(defun other (player)
  (if (equal player 'b) 'w 'b))

;;-----
;; return the winner, if game is not over yet, return nil

(defun winner ()
  (cond ((equal #\Space (get-board *b-king-row* *b-king-col* *board*)) 'w)
        ((equal #\Space (get-board *w-king-row* *w-king-col* *board*)) 'b)
        (t nil)))

;;----- start a game -----
(defun game()

  (init-board *board*)
  (print "You play WHITE and computer plays BLACK")

  (print "Do you want to move first? (y/n) ")
  (let ((answer (read)))
    (cond ((equal answer 'y) (setq player 'w))
          ((equal answer 'n) (setq player 'b))
  ))

```

```
(t (print "That's not an correct answer. Computer will play first.") (setq player 'b)))

(let ((move-index 1))
  (loop (print-board *board*)
        (move player move-index *board*)
        (setq player (other player))
        (setq move-index (+ move-index 1))
        (unless (equal nil (winner)) (return))))

(let ((who (winner)))
  (print-board *board*)
  (cond ((equal who 'b)(print "Black has won!"))
        ((equal who 'w)(print "White has won"))
        (t "No winner!"))))

.....
(game))
```