Parallel Haskell Chess Engine

Nikolaus Holzer

December 18, 2024

1 Introduction

This project implements a fully fledged (almost) chess engine using the popular and well-known minimax algorithm in Haskell. Throughout this report, I will provide details on the implementation of my chess engine, subsequently, I will explain the minimax algorithm as well as the optimizations that I included to make it run faster. My key goals for this project were to create a UCI (Universal Chess Interface) compatible chess Engine, that I could use with chess GUIs like cutechess, write the data structures and evaluation metrics myself and implement a minimax algorithm that takes into account the limitations of my data structures and helper functions.

2 Overview of the engine

I implemented my chess engine without the help of chess-specific Haskell packages. This has the distinct advantage, that I was able better to study the performance characteristics of the chess engine.

2.1 UCI

UCI is a commonly adopted yet very poorly documented communication protocol for chess engines and GUIs to communicate with each other. Since one of my aims for this project was to use my engine in my GUI of choice, I implemented a working UCI Haskell engine. This poses one main challenge, namely that UCI expects the engine to listen in an async way, which I did not implement to keep my project simple. Thus, I have been able to identify edge cases where the GUI throws errors due to my engine behaving in unexpected (to the GUI) ways. Nonetheless, most of the time, my engine can communicate with the GUI without a problem.

	a	b	c	d	e	f	g	h
8	56	57	58	59	60	61	62	63
7	48	49	50	51	52	53	54	55
6	40	41	42	43	44	45	46	47
5	32	33	34	35	36	37	38	39
4	24	25	26	27	28	29	30	31
3	16	17	18	19	20	21	22	23
2	8	9	10	11	12	13	14	15
1	0	1	2	3	4	5	6	7

Figure 1: Chess board squares with the corresponding number

2.2 Minimax algorithm

The minimax algorithm is a popular way to find the optimal move for each player in a two-player game. At each step, the algorithm computes the best move that a player can take given that the other player also plays the most optimal move. Since the minimax algorithm computes each possible move it has exponential space and time complexity. To address these shortcomings of the traditional minimax algorithm, pruning is commonly used to prevent the algorithm from exploring branches that are dominated by another branch. For example in Figure 3 the maximizing player goes first. The maximizer knows that the minimizer will always choose 2, which is less than 3 if he were to pick the right branch, meaning that we can prune it and only compute the left branch.

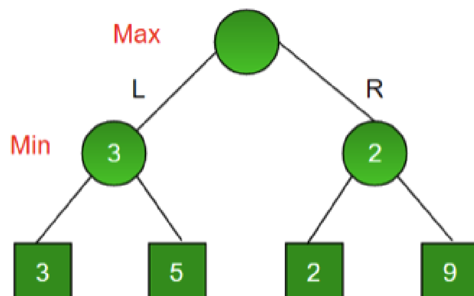

Figure 2: Sample minimax tree. https://www.geeksforgeeks.org/minimax-algorithm-in-game-theory-set-1-introduction/

2.3 Bitboard data structures

Instead of using a package that implements the chess game logic, I decided to build the bitboard data structures that my engine needed. By implementing these data structures, I was able to highlight the bottlenecks making optimizations in my minimax algorithm all the more important.

```haskell
startpos :: Board
startpos =
  Board
    { pawnsWhite = 0x000000000000FF00 -- [a,h]2 entire row
    , pawnsBlack = 0x00FF000000000000 -- [a,h]7 entire row
    , knightsWhite = 0x0000000000000042 -- b1 (bit 1) and g1 (bit 6)
    , knightsBlack = 0x4200000000000000 -- b8 (bit 57) and g8 (bit 62)
    , bishopsWhite = 0x0000000000000024 -- c1 (bit 2) and f1 (bit 5)
    , bishopsBlack = 0x2400000000000000 -- c8 (bit 58) and f8 (bit 61)
    , rooksWhite = 0x0000000000000081 -- a1 (bit 0) and h1 (bit 7)
    , rooksBlack = 0x8100000000000000 -- a8 (bit 56) and h8 (bit 63)
    , queensWhite = 0x0000000000000008 -- d1 (bit 3)
    , queensBlack = 0x0800000000000000 -- d8 (bit 59)
    , kingsWhite = 0x0000000000000010 -- e1 (bit 4)
    , kingsBlack = 0x1000000000000000 -- e8 (bit 60)
    }
```

Listing 1: Board datatype in Haskell

Bitboards are a way of representing the chess game board as a 64-bit word. This allows us to store the entire game state in a very efficient manner and allows for fast manipulations on the chessboard. In a bitboard game representation, each piece type is saved as a 64-bit word. As seen in 1, the numbering of the chessboard squares starts from the bottom left at 0 and then moves from left to right across the board. Listing 1 shows the board that encodes the chess game board with the appropriate bits set. In hexadecimal representation, each number corresponds to 4 bits, meaning that each row of the chess board is encoded by two hex bits as shown in 3.

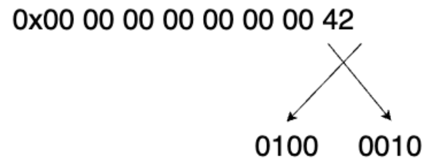

Figure 3: Hex to bitboard mapping

2.4 Move generation

Bitboards make move generation for pieces, computationally simple but programmatically very hard. Listing **??** shows the process for generating bitmasks that encode knight moves from each possible square. When possible my algorithm precomputes these move masks for pieces with fixed patterns like the King, Pawns, and Knights. Sliding pieces are more complex to implement since the algorithm needs to check for collisions, which if implemented inefficiently is costly. Listing 3 also shows the implementation of sliding piece generation with collision checking.

```
knightMasks :: [Bitboard]
knightMasks = [knightMoves (1 `shiftL` square) | square <- [0 .. 63]]

-- fileX is used to kill moves that will throw the knight off the board
knightMoves :: Bitboard -> Bitboard
knightMoves knightBitboard =
  let move1 = (knightBitboard `shiftL` 15) .&. complement fileH
      move2 = (knightBitboard `shiftL` 17) .&. complement fileA
      move3 = (knightBitboard `shiftL` 10) .&. complement fileAB
      move4 = (knightBitboard `shiftL` 6) .&. complement fileGH
      move5 = (knightBitboard `shiftR` 15) .&. complement fileA
      move6 = (knightBitboard `shiftR` 17) .&. complement fileH
      move7 = (knightBitboard `shiftR` 10) .&. complement fileGH
      move8 = (knightBitboard `shiftR` 6) .&. complement fileAB
   in move1 .|. move2 .|. move3 .|. move4 .|. move5 .|. move6 .|. move7 .|. move8

generateKnightMoves :: Int -> Bitboard -> Bitboard -> Bitboard
generateKnightMoves square occupancy captureable =
  let attacks = knightMasks !! square
   in attacks .&. complement occupancy

generateLine :: Bitboard -> Bitboard -> Bitboard -> Int -> Int -> Bitboard
generateLine position occupancy captureable step limit =
  let next =
        if step > 0
          then position `shiftL` step
          else position `shiftR` (-step) -- shiftL only trakes positive
   in if next == 0
           || invalidShift position step
           || canCapture position captureable
           || isBlocked position occupancy step
           || limit == 0
        then 0
        else next .|. generateLine next occupancy captureable step (limit - 1)

canCapture :: Bitboard -> Bitboard -> Bool
canCapture position captureable = (position .&. captureable) /= 0

isBlocked :: Bitboard -> Bitboard -> Int -> Bool
isBlocked position occupancy step =
  (step > 0 && (position `shiftL` step) .&. occupancy /= 0)
    || (step < 0 && (position `shiftR` (-step)) .&. occupancy /= 0)
```

Listing 2: Sample move generation for knights and sliding pieces

Figure 4 shows a visualization of bitboards of generated queen boards. Occupancy and captureable are simply the bitwise and of the respective colors at each turn.

```
0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0
0 0 1 0 1 0 0 0
0 0 0 0 0 0 0 0
0 0 0 1 0 0 0 0
```

(a) Occupancy (stop before)

```
0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0
0 0 0 0 1 0 0 0
1 0 0 0 0 0 0 0
0 0 0 0 0 0 0 1
0 0 0 0 0 0 0 0
0 0 0 0 1 0 0 0
0 0 0 0 0 0 0 0
```

(b) Captureable (stop on)

```
1 0 0 0 0 0 0 0
0 1 0 0 0 0 0 0
0 0 1 0 0 0 0 0
1 0 0 1 0 0 0 0
1 0 0 0 1 0 0 1
1 0 0 0 0 1 0 1
1 1 0 0 0 0 1 1
0 1 1 0 1 1 1 0
```

(c) Queen moves from a1, h1

Figure 4: Visualizing chess bitboards: occupancy, captureable, and queen moves.

3 Optimizations and results

My engine uses Bitboards, which is very memory efficient. For example, computing all possible bitboards for a search tree of depth 5 should only take $2^5 * 12 * 8 = 3072$ bits of memory without overheads. I implement two sequential optimizations on top of using bitboards as described above, as well as a simple parallelization strategy. Table 1 shows the runtimes for three different versions of the minimax algorithm. One plain, one with caching, one with caching and pruning, and finally caching pruning and parallelism.

Depth	Base Runtime (s)	Cache Runtime(s)	Cache AB Runtime(s)	Par 4 Runtime(s)	Par 8 Runtime(s)
1	0.246	0.023	0.024	0.015	0.020
2	0.021	0.016	0.017	0.016	0.019
4	0.305	0.127	0.127	0.039	0.048
6	128.602	3.060	2.601	0.823	0.734
8	-	61.664	51.687	17.686	14.349
10	-	3081.200	1015.601	447.699	978.045

Table 1: Runtime Results comparisons with Varying Depth

3.1 Intermediate result caching

As shown in Table 1, the simple minimax is very slow. Caching intermediary results significantly improve the runtimes, reducing the time for depth 6 by a factor of 40. This method caches subtrees using a string representation of the board and the current depth as a key, with the best score as a value. This lets us cache entire subtrees, which saves recomputation times. Particularly in the early stages of the game, this can have tremendous speedups since multiple different move combinations can lead to identical subtrees.

3.2 Alpha-beta pruning

Adding alpha-beta pruning to the sequential implementation further optimizes the performance over caching, the sequential implementation being 3 times faster for a depth of 10 as seen in Table 1. Alpha-beta pruning works by ignoring branches from which the current player could never receive a better reward based on the optimal strategy of the opposing player. We can save computation time by pruning these branches.

3.3 Parallelization

Despite my optimizations, sequential runtimes remain high. This indicates good potential for parallelization since a lot of work is being done for each piece, and we are not IO or memory-bound. My strategy to parallelize the minimax implementation is simple. I use a thread to evaluate each piece in parallel, and then aggregate the results in a top-level

helper which does the final move selection. This lets me maximize the work that each thread is doing whilst also maintaining simplicity in my program.

We can see in Table 1 that going to 8 cores, improves the runtimes for less than depth 10, but it seems like computational constraints double the runtime for a depth of 10 compared to using 4 cores.

```
evaluateMovesInParallel :: Int -> Board -> Int -> [Move] -> State Cache [Int]
evaluateMovesInParallel color board depth moves = do
  let evaluateMove :: Move -> Int
      evaluateMove move =
        let newBoard = applyMove2 move board
            (score, _) = simpleMinimaxAB (-color) newBoard (depth -
                        1) (minBound :: Int) (maxBound :: Int) Map.empty
        in score
  let moveEvaluations = parMap rpar evaluateMove moves `using` parList rseq
  return moveEvaluations
```

Listing 3: Parallel evaluation of top-level moves leaves significant work for each core.

4 Analysis of parallelization

Using a Macbook Pro M2 (8 cores) makes running with up to 8 threads possible. Comparing the runtime performance using different numbers of threads for different depths shows that the speedup scales logarithmically with added threads for depths 6 and 8 as seen in Figure 5. Here the effect of diminishing returns is best seen, as performance gains shrink with each extra thread. At a depth of 10, the system becomes resource-constrained, which can be seen by the stark decline in performance when using more than 4 threads. After 4 cores, adding more does not improve runtimes significantly or it harms them.

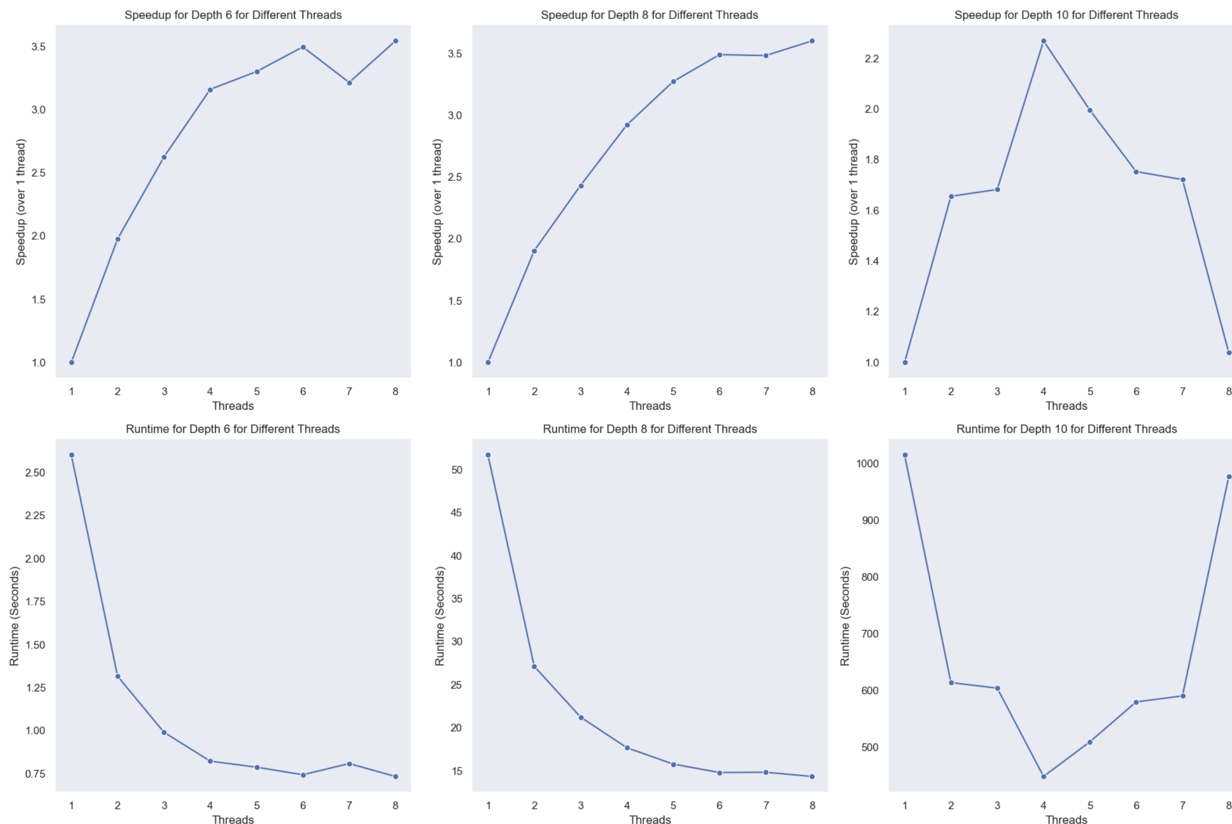

Figure 5: Speedup and runtime comparisons for depth=6, 8, 10

4.1 Threadscope analysis

Analyzing the effect of increasing the number of threads with Threadscope confirms the above findings. When comparing between 4 and 8 cores as seen in Figure 6 and Figure 7 the execution with 4 threads has much less downtime than on 8 threads. Analyzing the spark conversion shows that both 4 and 8 threads sparked 60 sparks with 45 being converted. Since my program doesn't spark many threads, this is expected behavior and indicates that instead of a spark pool overflow the bottleneck is somewhere else.

Analyzing the heap shows that executing on 4 cores yields a heap size of 29.4 GB whereas 8 cores has a heap size of 45.7 GB. My computer has 24 GB of RAM, indicating that my program is memory-constrained. This explains why after exceeding 4 threads for a depth of 10, performance takes such a hit since the code is overflowing more and more from memory.

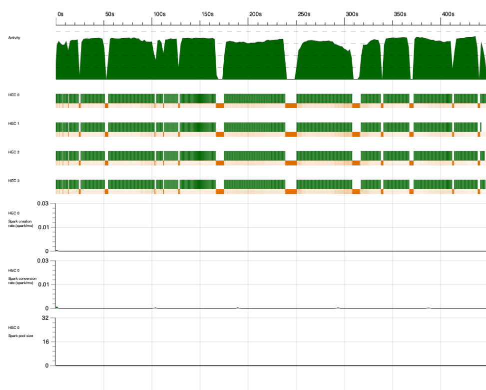

Figure 6: Threadscope for 4 threads and depth=10

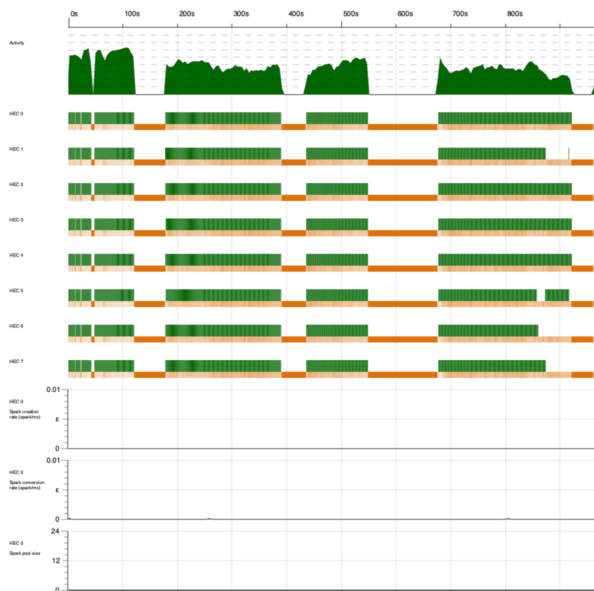

Figure 7: Threadscope for 8 threads and depth=10

Figure 8: Overall caption for both figures (optional).

5 Conclusion

In this project, I implemented a very efficient sequential minimax algorithm using Bitboard data structures, caching, and alpha-beta pruning. I then parallelize it to increase the performance, eventually becoming memory-bound despite using efficient data types. My results show that parallelizing brings an impressive boost to the performance of the minimax algorithm.

6 Appendix

6.1 BestMove.hs

[htbp]

```
positionalScore :: Board -> Int
positionalScore board =
  sum (map (\sq -> pawnTableWhite !! sq) (bitboardSquares (pawnsWhite board)))
    + sum (map (\sq -> knightTableWhite !! sq) (bitboardSquares (knightsWhite board)))
    + sum (map (\sq -> bishopTableWhite !! sq) (bitboardSquares (bishopsWhite board)))
    + sum (map (\sq -> rookTableWhite !! sq) (bitboardSquares (rooksWhite board)))
    + sum (map (\sq -> queenTableWhite !! sq) (bitboardSquares (queensWhite board)))
    + sum (map (\sq -> kingTableWhite !! sq) (bitboardSquares (kingsWhite board)))
    - sum (map (\sq -> pawnTableWhite !! (indexForBlack sq)) (bitboardSquares (pawnsBlack board)))
    - sum (map (\sq -> knightTableWhite !! (indexForBlack sq)) (bitboardSquares (knightsBlack board)))
    - sum (map (\sq -> bishopTableWhite !! (indexForBlack sq)) (bitboardSquares (bishopsBlack board)))
    - sum (map (\sq -> rookTableWhite !! (indexForBlack sq)) (bitboardSquares (rooksBlack board)))
    - sum (map (\sq -> queenTableWhite !! (indexForBlack sq)) (bitboardSquares (queensBlack board)))
    - sum (map (\sq -> kingTableWhite !! (indexForBlack sq)) (bitboardSquares (kingsBlack board)))

-- Updated evaluateBoard with positional considerations
evaluateBoard :: Board -> Int -> Int
evaluateBoard board color=
  let materialScore =
        (popCount (pawnsWhite board) * 10)
          + (popCount (knightsWhite board) * 30)
          + (popCount (bishopsWhite board) * 30)
          + (popCount (rooksWhite board) * 50)
          + (popCount (queensWhite board) * 90)
          + (popCount (kingsWhite board) * 500)
          - (popCount (pawnsBlack board) * 10)
          - (popCount (knightsBlack board) * 30)
          - (popCount (bishopsBlack board) * 30)
          - (popCount (rooksBlack board) * 50)
          - (popCount (queensBlack board) * 90)
          - (popCount (kingsBlack board) * 500)
      posScore = positionalScore board
      -- posScore = 0
   in color * (materialScore + posScore)

applyMove2 :: Move -> Board -> Board
applyMove2 move board =
  let fromSq = moveFrom move
      toSq = moveTo move
      -- Identify the piece that is moving
      mbPiece = findPiece board fromSq
      -- Clear the from-square from all boards
      clearedBoard = clearSquare board fromSq
      -- Also clear the to-square in case of a capture
      capturedClearedBoard = clearSquare clearedBoard toSq
      pieceName =
        case mbPiece of
          Just (pn, _) -> pn
          Nothing -> error "No piece found on from-square (invalid move)"
      movedBoard = placePiece pieceName toSq capturedClearedBoard
   in movedBoard

-- Place a piece at a given square on the board
placePiece :: String -> Int -> Board -> Board
placePiece pieceName sq board =
  let bit = 1 `shiftL` sq
   in case pieceName of
        "pawnsWhite" -> board {pawnsWhite = pawnsWhite board .|. bit}
        "pawnsBlack" -> board {pawnsBlack = pawnsBlack board .|. bit}
```

```
        "knightsWhite" -> board {knightsWhite = knightsWhite board .|. bit}
        "knightsBlack" -> board {knightsBlack = knightsBlack board .|. bit}
        "bishopsWhite" -> board {bishopsWhite = bishopsWhite board .|. bit}
        "bishopsBlack" -> board {bishopsBlack = bishopsBlack board .|. bit}
        "rooksWhite" -> board {rooksWhite = rooksWhite board .|. bit}
        "rooksBlack" -> board {rooksBlack = rooksBlack board .|. bit}
        "queensWhite" -> board {queensWhite = queensWhite board .|. bit}
        "queensBlack" -> board {queensBlack = queensBlack board .|. bit}
        "kingsWhite" -> board {kingsWhite = kingsWhite board .|. bit}
        "kingsBlack" -> board {kingsBlack = kingsBlack board .|. bit}
        _ -> error "Invalid piece type"

-- ########################################################################
-- Helper to determine which piece is at a given square (0-63).
-- This is how I encode the current position to compare with other positions (at the same depth)
getPieceChar :: Board -> Int -> Char
getPieceChar board sq
  | testBit (pawnsWhite board) sq = 'P'
  | testBit (pawnsBlack board) sq = 'p'
  | testBit (knightsWhite board) sq = 'N'
  | testBit (knightsBlack board) sq = 'n'
  | testBit (bishopsWhite board) sq = 'B'
  | testBit (bishopsBlack board) sq = 'b'
  | testBit (rooksWhite board) sq = 'R'
  | testBit (rooksBlack board) sq = 'r'
  | testBit (queensWhite board) sq = 'Q'
  | testBit (queensBlack board) sq = 'q'
  | testBit (kingsWhite board) sq = 'K'
  | testBit (kingsBlack board) sq = 'k'
  | otherwise = '.' -- empty square

-- Convert the board to a simplified FEN-like string.
boardToFEN :: Board -> String
boardToFEN board =
  let ranks = [7,6 .. 0] -- top rank = 7, down to 0
      fenRows = map (rankToFen board) ranks
   in foldr1 (\r acc -> r ++ "/" ++ acc) fenRows

rankToFen :: Board -> Int -> String
rankToFen board r =
  let start = r * 8
      end = start + 7
      chars = [getPieceChar board sq | sq <- [start .. end]]
   in compressEmpty chars

-- Compress consecutive empty squares ('.') into digits as per FEN
compressEmpty :: [Char] -> String
compressEmpty = go 0
  where
    go :: Int -> String -> String
    go count [] =
      if count > 0
        then show count
        else ""
    go count (c:cs)
      | c == '.' = go (count + 1) cs
      | otherwise =
        (if count > 0
           then show count
           else "")
          ++ [c]
          ++ go 0 cs


-- I define a cache Map datatype which I use to keep track of previous executied trees, which is much easier than r
```

```haskell
-- Key: (FEN_string, color, depth) -> value: score
type Cache = Map.Map (String, Int, Int) Int

-- some helper funcs
mergeCaches :: Cache -> Cache -> Cache
mergeCaches = Map.union

getCachedScore :: Cache -> (String, Int, Int) -> Maybe Int
getCachedScore cache key = Map.lookup key cache

insertCacheScore :: Cache -> (String, Int, Int) -> Int -> Cache
insertCacheScore cache key score = Map.insert key score cache
-- ########################################################################
-- use generateAllMoves for each player at each turn to do the minimax 1 = white -1 = black


debugTrace :: String -> a -> a
debugTrace msg x =
  if debug
    then trace msg x
    else x

simpleMinimax :: Int -> Board -> Int -> Int
simpleMinimax color board depth = debugTrace msg result
  where
    moves =
      if color == 1
        then (generateAllMovesWhite board)
        else (generateAllMovesBlack board)
    msg = "Minimax called at depth: " ++ show depth ++ ", color: " ++ show color ++ ", moves: " ++ show (length mov
    result
      | depth == 0 || null moves =
        let score = evaluateBoard board color
         in debugTrace ("Terminal node reached with score: " ++ show score) score
      | color == 1 =
        let scores = [simpleMinimax (-color) (applyMove2 m board) (depth - 1) | m <- moves]
            best = maximum scores
         in debugTrace ("Maximizer choosing best score: " ++ show best) best
      | otherwise =
        let scores = [simpleMinimax (-color) (applyMove2 m board) (depth - 1) | m <- moves]
            best = minimum scores
         in debugTrace ("Minimizer choosing best score: " ++ show best) best

chooseBestMove :: Int -> Board -> Int -> Move
chooseBestMove color board depth =
  let moves =
        if color == 1
          then generateAllMovesWhite board
          else generateAllMovesBlack board
      scores = [(simpleMinimax (-color) (applyMove2 m board) (depth - 1), m) | m <- moves]
   in if null scores
        then error "No moves available"
        else if color == 1
               then snd (maximumBy (compare `on` fst) scores)
               else snd (minimumBy (compare `on` fst) scores)


simpleMinimaxCache :: Int -> Board -> Int -> Int -> Int -> State Cache Int
simpleMinimaxCache color board depth alpha beta = do
  let fen = boardToFEN board
      key = (fen, color, depth)
  cache <- get
  case Map.lookup key cache of
    Just score -> return score
    Nothing -> do
```

```
      let moves =
            if color == 1
              then generateAllMovesWhite board
              else generateAllMovesBlack board
      if depth == 0 || null moves
        then do
          let score = evaluateBoard board color -- I need to flip this because black should have - and white +.
          debugTrace
            ("Terminal node: "
               ++ show key
               ++ ", Score: "
               ++ show score
               ++ ", Cache: "
               ++ show (Map.lookup key cache))
            $ return ()
          modify (Map.insert key score)
          return score
        else do
          let helper =
                if color == 1
                  then simpleHelperMaxC
                  else simpleHelperMinC
          score <- helper moves board depth alpha beta
          modify (Map.insert key score)
          return score

-- Helper for Maximizing Player
simpleHelperMaxC ::
     [Move] -> Board -> Int -> Int -> Int -> State Cache Int
simpleHelperMaxC [] _ _ alpha _ = return alpha
simpleHelperMaxC (m:ms) board depth alpha beta = do
  let newBoard = applyMove2 m board
  score <- simpleMinimaxCache (-1) newBoard (depth - 1) alpha beta
  let newAlpha = max alpha score
  if newAlpha >= beta
    then return newAlpha  -- Prune
    else do
      restScore <- simpleHelperMaxC ms board depth newAlpha beta
      return (max score restScore)

-- Helper for Minimizing Player
simpleHelperMinC ::
     [Move] -> Board -> Int -> Int -> Int -> State Cache Int
simpleHelperMinC [] _ _ _ beta = return beta
simpleHelperMinC (m:ms) board depth alpha beta = do
  let newBoard = applyMove2 m board
  score <- simpleMinimaxCache 1 newBoard (depth - 1) alpha beta
  let newBeta = min beta score
  if newBeta <= alpha
    then return newBeta  -- Prune
    else do
      restScore <- simpleHelperMinC ms board depth alpha newBeta
      return (min score restScore)


chooseBestMoveCache :: Int -> Board -> Int -> State Cache (Maybe Move)
chooseBestMoveCache color board depth = do
  let moves =
        if color == 1
          then generateAllMovesWhite board
          else generateAllMovesBlack board
  if null moves
    then return Nothing
    else do
      evaluatedMoves <- mapM (\m -> do
```

```
                                let newBoard = applyMove2 m board
                                score <- simpleMinimaxCache (-color) newBoard (depth - 1) (-1000000) 1000000
                                return (m, score)
                            ) moves

      let bestMove = if color == 1
                      then Just $ fst $ maximumBy (comparing snd) evaluatedMoves
                      else Just $ fst $ minimumBy (comparing snd) evaluatedMoves
      return bestMove


-- Minimax with Alpha-Beta Pruning and Caching
simpleMinimaxAB :: Int -> Board -> Int -> Int -> Int -> Cache -> (Int, Cache)
simpleMinimaxAB color board depth alpha beta cache =
  let fen = boardToFEN board
      key = (fen, color, depth)
  in case getCachedScore cache key of
       Just score -> (score, cache)
       Nothing ->
         let moves =
               if color == 1
                 then generateAllMovesWhite board
                 else generateAllMovesBlack board
         in if depth == 0 || null moves
              then
                let score = evaluateBoard board color -- when the bottom depth is reached evaluate the value of the
                    newCache = insertCacheScore cache key score
                in (score, newCache)
                -- in debugTrace
                --       ("Terminal node: "
                --         ++ show key
                --         ++ ", Score: "
                --         ++ show score
                --         ++ ", Cache: "
                --         ++ show (Map.lookup key cache)) (score, newCache)
              else
                let helper =
                      if color == 1
                        then simpleHelperMax
                        else simpleHelperMin
                    (score, updatedCache) = helper moves board depth alpha beta cache
                in (score, insertCacheScore updatedCache key score)


simpleHelperMax ::
     [Move] -> Board -> Int -> Int -> Int -> Cache -> (Int, Cache)
simpleHelperMax [] _ _ alpha _ cache = (alpha, cache)
simpleHelperMax (m:ms) board depth alpha beta cache =
  let newBoard = applyMove2 m board
      (score, newCache) = simpleMinimaxAB (-1) newBoard (depth - 1) alpha beta cache
      newAlpha = max alpha score
  in if newAlpha >= beta
       then (newAlpha, newCache)  -- Prune
       else
         let (restScore, restCache) = simpleHelperMax ms board depth newAlpha beta newCache
         in (max score restScore, restCache)

simpleHelperMin ::
     [Move] -> Board -> Int -> Int -> Int -> Cache -> (Int, Cache)
simpleHelperMin [] _ _ _ beta cache = (beta, cache)
simpleHelperMin (m:ms) board depth alpha beta cache =
  let newBoard = applyMove2 m board
      (score, newCache) = simpleMinimaxAB 1 newBoard (depth - 1) alpha beta cache
      newBeta = min beta score
  in if newBeta <= alpha
```

```
        then (newBeta, newCache)  -- Prune
        else
          let (restScore, restCache) = simpleHelperMin ms board depth alpha newBeta newCache
          in (min score restScore, restCache)


-- Choose the best move using minimax with caching and parallel processing
-- does the first layer of the minimax here
chooseBestMovePar :: Int -> Board -> Int -> State Cache (Maybe Move)
chooseBestMovePar color board depth = do
  let moves =
        if color == 1
          then generateAllMovesWhite board
          else generateAllMovesBlack board
  if null moves
    then do
      trace "No moves available for the current player." $
        return Nothing
    else do
      evaluatedScores <- evaluateMovesInParallel color board depth moves

      let bestMove =
            if color == 1
              then Just $ snd $ maximumBy (comparing fst) (zip evaluatedScores moves)
              else Just $ snd $ minimumBy (comparing fst) (zip evaluatedScores moves)

      return bestMove

evaluateMovesInParallel :: Int -> Board -> Int -> [Move] -> State Cache [Int]
evaluateMovesInParallel color board depth moves = do
  let evaluateMove :: Move -> Int
      evaluateMove move =
        let newBoard = applyMove2 move board
            (score, _) = simpleMinimaxAB (-color) newBoard (depth - 1) (minBound :: Int) (maxBound :: Int) Map.empt
        in score
  let moveEvaluations = parMap rpar evaluateMove moves `using` parList rseq
  return moveEvaluations


-- Define how to show a Move
showMove2 :: Maybe Move -> Int -> String
showMove2 Nothing _ = "No move"
showMove2 (Just m) color =
  "Move from " ++ showSquare (moveFrom m) ++ " to " ++ showSquare (moveTo m) ++ " for color " ++ show color

extractMove :: Maybe Move -> Move
extractMove Nothing = Move {moveFrom = 0, moveTo = 0, promotion = "", isCapture = 0, isCastling = 0, isEnPassant =
extractMove (Just move) = move

showSquare :: Int -> String
showSquare sq =
  let file = ['a' .. 'h'] !! (sq `mod` 8)
      rank = show ((sq `div` 8) + 1)
   in [file] ++ rank
```

6.2 Engine.hs

[htbp]

```
        module Main where


import Control.Monad (when)
```

```
import Control.Monad.State
import Data.Bits (shiftL)
import qualified Data.Map.Strict as Map
import System.Exit (exitSuccess)
import System.IO

import Bestmove
import Bitboard
import Fen

--  https://backscattering.de/chess/uci/
-- Chess engine using fen notation and bitboards demo
-- ################################################################
uciPrint :: String
uciPrint =
  unlines
    $ [ "id name HaskellChess 1.0"
      , "id author nik"
      , ""
      , "option name Debug Log File type string default"
      , "option name Threads type spin default 1 min 1 max 16" -- can set to more maybe --> threads are at compile
      , "uciok" -- pass this at the end for uci
      ]

data EngineConfig = EngineConfig
  { name :: String
  , author :: String
  , debugMode :: Int
  , threads :: Int
  , fen :: FEN
  }

instance Show EngineConfig where
  show config =
    unlines
      $ [ "id name " ++ name config
        , "id author " ++ author config
        , ""
        , "option name Debug " ++ show (debugMode config)
        , "option name Threads " ++ show (threads config)
        , "starting position " ++ show (fen config)
        ]

setConfig :: Int -> Int -> FEN -> EngineConfig
setConfig t d f =
  EngineConfig
    { name = "HaskellChess 1.0"
    , author = "nik"
    , threads = t
    , debugMode = d
    , fen = f
    }

applyMove :: Board -> Int -> Int -> Board
applyMove board fromSquare toSquare =
  let fromBit = 1 `shiftL` (fromSquare)
      toBit = 1 `shiftL` (toSquare)
      clearBoard = clearSquare board toSquare
   in case findPiece board fromSquare of
        Just ("pawnsWhite", bb) ->
          clearBoard {pawnsWhite = updateBitboard bb fromBit toBit}
        Just ("pawnsBlack", bb) ->
          clearBoard {pawnsBlack = updateBitboard bb fromBit toBit}
        Just ("knightsWhite", bb) ->
          clearBoard {knightsWhite = updateBitboard bb fromBit toBit}
```

```
        Just ("knightsBlack", bb) ->
          clearBoard {knightsBlack = updateBitboard bb fromBit toBit}
        Just ("bishopsWhite", bb) ->
          clearBoard {bishopsWhite = updateBitboard bb fromBit toBit}
        Just ("bishopsBlack", bb) ->
          clearBoard {bishopsBlack = updateBitboard bb fromBit toBit}
        Just ("rooksWhite", bb) ->
          clearBoard {rooksWhite = updateBitboard bb fromBit toBit}
        Just ("rooksBlack", bb) ->
          clearBoard {rooksBlack = updateBitboard bb fromBit toBit}
        Just ("queensWhite", bb) ->
          clearBoard {queensWhite = updateBitboard bb fromBit toBit}
        Just ("queensBlack", bb) ->
          clearBoard {queensBlack = updateBitboard bb fromBit toBit}
        Just ("kingsWhite", bb) ->
          clearBoard {kingsWhite = updateBitboard bb fromBit toBit}
        Just ("kingsBlack", bb) ->
          clearBoard {kingsBlack = updateBitboard bb fromBit toBit}
        Nothing -> error "No piece found"
        _ -> error "No piece found"

  -- sum each layer of the board with corresponding piece value
  -- use popcount on each Bitboard

-- minimax best move solver
-- board position, color to move,
-- this should be in the
runEngine :: Int -> Int -> FEN -> Board -> IO ()
runEngine threads dMode fenConfig board = do
  -- let config = setConfig threads dMode fenConfig
  guiIn <- getLine
  case words guiIn of
    -- set up the configurations of the engine
    ["quit"] -> exitSuccess
    ["uci"] -> putStr uciPrint >> runEngine threads dMode fenConfig board
    -- option setting
    ["debug", val] ->
      case val of
        "on" -> runEngine threads 1 fenConfig board
        "off" -> runEngine threads 0 fenConfig board
        _ -> runEngine threads 0 fenConfig board
    -- set starting positions
    ["position", "startpos"] -> do
      when (dMode == 1) (putStrLn "Recieved starting position without moves")
      runEngine threads dMode setDefaultFen board
    -- need to change this to handle list of moves
    "position":"startpos":"moves":moves -> do
      when
        (dMode == 1)
        (putStrLn
           ("Recieved starting position with moves "
              ++ unwords moves
              ++ ". Adding the last move "
              ++ last moves
              ++ " to the state."))
      let (fromSquare, toSquare) = splitAt 2 (last moves) -- is monadic, doesn't need an in
          newBoard =
            applyMove board (squareLookup fromSquare) (squareLookup toSquare)
      -- when (dMode == 1) (putStrLn (show newBoard))
      runEngine threads dMode setDefaultFen newBoard
    -- start a new game
    ["ucinewgame"] -> runEngine 1 1 setDefaultFen board
    -- ask the engine if it is available
    ["isready"] -> putStrLn "readyok" >> runEngine threads dMode fenConfig board
    -- start calculating best move based on fen position
```

```haskell
    ["go", "infinite"] -> do
      let color =
            if (activeColor fenConfig) == 'b'
              then 1
              else -1
      -- let color = -1
      let dep = 4
      putStrLn
        ("Starting minimax for cpu on color: "
           ++ show color
           ++ ". DEPTH: "
           ++ show dep)
      let (bestMove, _) = runState (chooseBestMovePar color board dep) Map.empty
      let moveStr = showMove (extractMove bestMove) color
      -- let bestMove1 = chooseBestMove color board 3
      -- let moveStr = showMove (bestMove1) color

      putStrLn ("move: " ++ moveStr)
      let (fromSquare, toSquare) = splitAt 2 moveStr
          newBoard =
            applyMove board (squareLookup fromSquare) (squareLookup toSquare)
      putStrLn fromSquare
      putStrLn toSquare
      -- when (dMode == 1) (putStrLn (show newBoard))
      putStrLn ("bestmove " ++ moveStr)
      runEngine threads dMode fenConfig newBoard
    ["stop"] -> exitSuccess -- todo exit whatever calculation is currently ongong kill current calculation --> alwa
    -- not 100% sure what this does
    ["ponderhit"] -> runEngine threads dMode fenConfig board -- can ignore this
    _ -> runEngine threads dMode fenConfig board

main :: IO ()
main = do
  hSetBuffering stdout NoBuffering
  hSetBuffering stdin NoBuffering
  runEngine 1 1 setDefaultFen startpos
```

6.3 Bitboard.hs

[htbp]

```haskell
        {-# LANGUAGE DeriveGeneric #-}
{-# LANGUAGE DeriveAnyClass #-}

module Bitboard where

import Control.DeepSeq (NFData)
import Data.Bits
  ( (.&.)
  , (.|.)
  , complement
  , shiftL
  , shiftR
  , testBit
  , xor
  )
import Data.Char (intToDigit)
import Data.Word (Word64)
import GHC.Generics (Generic)
import Numeric (showIntAtBase)

-- Move type that encodes each move
-- Doesn't compute the capture here, does that in evaluate board
```

```
-- ###############################################################
data Move = Move
  { moveFrom :: Int -- save as a bitboard
  , moveTo :: Int -- also save as a bitboard
  , promotion :: String
  , isCapture :: Int
  , isCastling :: Int
  , isEnPassant :: Int
  } deriving (Show, Eq, Generic, NFData)

-- instance NFData Move where
--   rnf (Move f t p c e) = rnf f `seq` rnf t `seq` rnf p `seq` rnf c `seq` rnf e
movesGrid :: [Move] -> String
movesGrid moves = showGrid combinedBitboard
  where
    combinedBitboard = foldr combineMove 0 moves
    combineMove move acc = acc .|. 1 `shiftL` moveTo move

showMove :: Move -> Int -> String
showMove move color =
  squareToAlgebraic (moveFrom move) color
    ++ squareToAlgebraic (moveTo move) color

squareToAlgebraic :: Int -> Int -> String
squareToAlgebraic square color =
  let file =
        if color == 1
          then ['A' .. 'H'] !! (square `mod` 8)
          else ['a' .. 'h'] !! (square `mod` 8)
      rank = show (1 + (square `div` 8))
   in [file] ++ rank

-- ###############################################################
type Bitboard = Word64 -- 8*8 bits

data Board = Board
  { pawnsWhite :: Bitboard
  , pawnsBlack :: Bitboard
  , knightsBlack :: Bitboard
  , knightsWhite :: Bitboard
  , bishopsBlack :: Bitboard
  , bishopsWhite :: Bitboard
  , rooksBlack :: Bitboard
  , rooksWhite :: Bitboard
  , queensWhite :: Bitboard
  , queensBlack :: Bitboard
  , kingsWhite :: Bitboard
  , kingsBlack :: Bitboard
  }

instance Show Board where
  show board =
    unlines
      [ "Pawns (White):   " ++ helper (pawnsWhite board)
      , "Pawns (Black):   " ++ helper (pawnsBlack board)
      , "Knights (White): " ++ helper (knightsWhite board)
      , "Knights (Black): " ++ helper (knightsBlack board)
      , "Bishops (White): " ++ helper (bishopsWhite board)
      , "Bishops (Black): " ++ helper (bishopsBlack board)
      , "Rooks (White):   " ++ helper (rooksWhite board)
      , "Rooks (Black):   " ++ helper (rooksBlack board)
      -- , "Castles (White): " ++ helper (castlesWhite board)
      -- , "Castles (Black): " ++ helper (castlesBlack board)
      , "Queens (White):  " ++ helper (queensWhite board)
      , "Queens (Black):  " ++ helper (queensBlack board)
```

```
      , "Kings (White):   " ++ helper (kingsWhite board)
      , "Kings (Black):   " ++ helper (kingsBlack board)
      ]
    where
      helper :: Word64 -> String
      helper bitboard =
        let x = showIntAtBase 16 intToDigit bitboard ""
         in "0x" ++ replicate (16 - length x) '0' ++ x

showPieceGrid :: Board -> String -> String
showPieceGrid board pieceType =
  case pieceType of
    "PawnsWhite" -> showGrid (pawnsWhite board)
    "PawnsBlack" -> showGrid (pawnsBlack board)
    "KnightsWhite" -> showGrid (knightsWhite board)
    "KnightsBlack" -> showGrid (knightsBlack board)
    "BishopsWhite" -> showGrid (bishopsWhite board)
    "BishopsBlack" -> showGrid (bishopsBlack board)
    "RooksWhite" -> showGrid (rooksWhite board)
    "RooksBlack" -> showGrid (rooksBlack board)
    "QueensWhite" -> showGrid (queensWhite board)
    "QueensBlack" -> showGrid (queensBlack board)
    "KingsWhite" -> showGrid (kingsWhite board)
    "KingsBlack" -> showGrid (kingsBlack board)
    _ -> "Invalid piece type"

showGrid :: Bitboard -> String
showGrid bitboard = unlines $ map showRow [7,6 .. 0]
  where
    -- Create an 8x8 binary grid representation
    showRow row = unwords $ map (showSquare row) [0 .. 7]
    showSquare row col =
      if testBit bitboard (row * 8 + col)
        then "1"
        else "0"

showBin :: Bitboard -> String
showBin bitboard =
  let x = showIntAtBase 2 intToDigit bitboard ""
   in replicate (64 - length x) '0' ++ x

-- hesadecimal instaed of binary. Each row corresponds to two hex values
-- each piece type has it's own board --> can or all together for the entire board (64 bits)
-- in the lookup table below, we need to check for each piece type if the desired square is occupied
startpos :: Board
startpos =
  Board
    { pawnsWhite = 0x000000000000FF00 -- [a,h]2 entire row
    , pawnsBlack = 0x00FF000000000000 -- [a,h]7 entire row
    , knightsWhite = 0x0000000000000042 -- b1 (bit 1) and g1 (bit 6)
    , knightsBlack = 0x4200000000000000 -- b8 (bit 57) and g8 (bit 62)
    , bishopsWhite = 0x0000000000000024 -- c1 (bit 2) and f1 (bit 5)
    , bishopsBlack = 0x2400000000000000 -- c8 (bit 58) and f8 (bit 61)
    , rooksWhite = 0x0000000000000081 -- a1 (bit 0) and h1 (bit 7)
    , rooksBlack = 0x8100000000000000 -- a8 (bit 56) and h8 (bit 63)
    , queensWhite = 0x0000000000000008 -- d1 (bit 3)
    , queensBlack = 0x0800000000000000 -- d8 (bit 59)
    , kingsWhite = 0x0000000000000010 -- e1 (bit 4)
    , kingsBlack = 0x1000000000000000 -- e8 (bit 60)
    }

squareLookup :: String -> Int
squareLookup square =
  let (file, rank) = splitAt 1 square
      x = fromEnum (head file) - fromEnum 'a'
```

```
      y = read rank - 1
   in y * 8 + x

-- and with a negative mask to clear. Safe to do over whole board for all pieces since only one piece can occupy a
clearSquare :: Board -> Int -> Board
clearSquare board square =
  let mask = complement (1 `shiftL` square) -- Mask to clear the square
   in board
        { pawnsWhite = pawnsWhite board .&. mask
        , pawnsBlack = pawnsBlack board .&. mask
        , knightsWhite = knightsWhite board .&. mask
        , knightsBlack = knightsBlack board .&. mask
        , bishopsWhite = bishopsWhite board .&. mask
        , bishopsBlack = bishopsBlack board .&. mask
        , rooksWhite = rooksWhite board .&. mask
        , rooksBlack = rooksBlack board .&. mask
        , queensWhite = queensWhite board .&. mask
        , queensBlack = queensBlack board .&. mask
        , kingsWhite = kingsWhite board .&. mask
        , kingsBlack = kingsBlack board .&. mask
        }

findPiece :: Board -> Int -> Maybe (String, Bitboard)
findPiece board square =
  let bit = 1 `shiftL` square
   in case () of
        _
          | pawnsWhite board .&. bit /= 0 ->
            Just ("pawnsWhite", pawnsWhite board)
          | pawnsBlack board .&. bit /= 0 ->
            Just ("pawnsBlack", pawnsBlack board)
          | knightsWhite board .&. bit /= 0 ->
            Just ("knightsWhite", knightsWhite board)
          | knightsBlack board .&. bit /= 0 ->
            Just ("knightsBlack", knightsBlack board)
          | bishopsWhite board .&. bit /= 0 ->
            Just ("bishopsWhite", bishopsWhite board)
          | bishopsBlack board .&. bit /= 0 ->
            Just ("bishopsBlack", bishopsBlack board)
          | rooksWhite board .&. bit /= 0 ->
            Just ("rooksWhite", rooksWhite board)
          | rooksBlack board .&. bit /= 0 ->
            Just ("rooksBlack", rooksBlack board)
          | queensWhite board .&. bit /= 0 ->
            Just ("queensWhite", queensWhite board)
          | queensBlack board .&. bit /= 0 ->
            Just ("queensBlack", queensBlack board)
          | kingsWhite board .&. bit /= 0 ->
            Just ("kingsWhite", kingsWhite board)
          | kingsBlack board .&. bit /= 0 ->
            Just ("kingsBlack", kingsBlack board)
          | otherwise -> Nothing

-- xor the frombit with the board to get rid of it, or the tobit to write to that spot
updateBitboard :: Bitboard -> Bitboard -> Bitboard -> Bitboard
updateBitboard bitboard fromBit toBit = (bitboard `xor` fromBit) .|. toBit

-- extract active squares from bitboard
-- ################################################################
bitboardSquares :: Bitboard -> [Int]
bitboardSquares bitboard =
  [i | i <- [0 .. 63], bitboard .&. (1 `shiftL` i) /= 0]

-- add a function to uppercase if white else lowercase
-- occupancy is the entire and of each board
```

```
-- add castling to king piece
generateAllMovesWhite :: Board -> [Move]
generateAllMovesWhite board =
  let whites =
        (pawnsWhite board)
          .|. (knightsWhite board)
          .|. (bishopsWhite board)
          .|. (rooksWhite board)
          .|. (queensWhite board)
          .|. (kingsWhite board)
      blacks =
        (pawnsBlack board)
          .|. (knightsBlack board)
          .|. (bishopsBlack board)
          .|. (rooksBlack board)
          .|. (queensBlack board)
          .|. (kingsBlack board)
      occupancy = blacks .|. whites
      captureable = blacks
   in (generateAllPawnMovesW (pawnsWhite board) occupancy captureable)
        ++ (generateAllKnightMoves (knightsWhite board) occupancy captureable)
        ++ (generateAllBishopMoves (bishopsWhite board) occupancy captureable)
        ++ (generateAllRookMoves (rooksWhite board) occupancy captureable)
        ++ (generateAllQueenMoves (queensWhite board) occupancy captureable)
        ++ (generateAllKingMoves (kingsWhite board) occupancy captureable)

generateAllMovesBlack :: Board -> [Move]
generateAllMovesBlack board =
  let whites =
        (knightsWhite board)
          .|. (bishopsWhite board)
          .|. (rooksWhite board)
          .|. (queensWhite board)
          .|. (kingsWhite board)
          .|. (pawnsWhite board)
      blacks =
        (knightsBlack board)
          .|. (bishopsBlack board)
          .|. (rooksBlack board)
          .|. (queensBlack board)
          .|. (kingsBlack board)
          .|. (pawnsBlack board)
      occupancy = blacks .|. whites
      captureable = whites
   in (generateAllPawnMovesB (pawnsBlack board) occupancy captureable)
        ++ (generateAllBishopMoves (bishopsBlack board) occupancy captureable)
        ++ (generateAllRookMoves (rooksBlack board) occupancy captureable)
        ++ (generateAllQueenMoves (queensBlack board) occupancy captureable)
        ++ (generateAllKingMoves (kingsBlack board) occupancy captureable)
        ++ (generateAllKnightMoves (knightsBlack board) occupancy captureable)

-- castle
-- understand en passant
-- do a board eval
{-
Here is the api for moves. I follow largely the same structure for each piece even though under the hood generation
All piece move generators can be called through generateAllXMoves and take the same arguments, making piece generat
-}
-- NOTE : I SWITCHED LEFT AND RIGHT
-- LEFT IS 7 SHIFTL AND RIGHT IS 9 NOT THE OTHER WAY AROUND. IT DOESN'T MAKE A DIFFERENCE THOUGH
fileA, fileH, fileAB, fileGH :: Bitboard
fileA = 0x0101010101010101

fileH = 0x8080808080808080
```

```
fileAB = fileA .|. (fileA `shiftL` 1)

fileGH = fileH .|. (fileH `shiftR` 1)

-- on the fly computed movement masks for pieces with fixed patterns
-- ################################################################
-- Knight
-- movement pattern
knightMasks :: [Bitboard]
knightMasks = [knightMoves (1 `shiftL` square) | square <- [0 .. 63]]

knightMoves :: Bitboard -> Bitboard
knightMoves knightBitboard =
  let move1 = (knightBitboard `shiftL` 15) .&. complement fileH
      move2 = (knightBitboard `shiftL` 17) .&. complement fileA
      move3 = (knightBitboard `shiftL` 10) .&. complement fileAB
      move4 = (knightBitboard `shiftL` 6) .&. complement fileGH
      move5 = (knightBitboard `shiftR` 15) .&. complement fileA
      move6 = (knightBitboard `shiftR` 17) .&. complement fileH
      move7 = (knightBitboard `shiftR` 10) .&. complement fileGH
      move8 = (knightBitboard `shiftR` 6) .&. complement fileAB
   in move1 .|. move2 .|. move3 .|. move4 .|. move5 .|. move6 .|. move7 .|. move8

generateKnightMoves :: Int -> Bitboard -> Bitboard -> Bitboard
generateKnightMoves square occupancy _ =
  let attacks = knightMasks !! square
   in attacks .&. complement occupancy

generateAllKnightMoves :: Bitboard -> Bitboard -> Bitboard -> [Move]
generateAllKnightMoves knightBoard occupancy captureable =
  [ Move
    square
    targetSquare
    ""
    (if targetBit .&. captureable /= 0
       then 1
       else 0)
    0
    0
  | square <- bitboardSquares knightBoard
  , targetSquare <-
      bitboardSquares (generateKnightMoves square occupancy captureable)
  , let targetBit = 1 `shiftL` targetSquare
  ]

-- precomputed move masks for all the sliding pieces
-- ################################################################
-- Rook
-- start at the square then shift by 8 for vertical and 1 for horizontal
generateRookMoves :: Int -> Bitboard -> Bitboard -> Bitboard
generateRookMoves square occupancy captureable =
  let position = 1 `shiftL` (square)
      up = generateLine position occupancy captureable 8 8 -- by 8 because we shift straight up so need to skip to
      down = generateLine position occupancy captureable (-8) 8
      right = generateLine position occupancy captureable 1 8
      left = generateLine position occupancy captureable (-1) 8
   in up .|. down .|. right .|. left

-- get the occupancy
generateAllRookMoves :: Bitboard -> Bitboard -> Bitboard -> [Move]
generateAllRookMoves rookBoard captureable occupancy -- after hitting a capturable block all cells behind it for sl
 =
  [ Move
    square
    targetSquare
```

```
    ""
    (if targetBit .&. captureable /= 0
       then 1
       else 0)
    0
    0
  | square <- bitboardSquares rookBoard
  , targetSquare <-
      bitboardSquares (generateRookMoves square occupancy captureable)
  , let targetBit = 1 `shiftL` targetSquare
  ]

-- bishop - same as rook but shift by 15 squares
-- ################################################################
generateBishopMoves :: Int -> Bitboard -> Bitboard -> Bitboard
generateBishopMoves square occupancy captureable =
  let position = 1 `shiftL` (square)
      upLeft = generateLine position occupancy captureable 9 8 -- by 8 because we shift straight up so need to skip
      upRight = generateLine position occupancy captureable 7 8
      downRight = generateLine position occupancy captureable (-9) 8
      downLeft = generateLine position occupancy captureable (-7) 8
   in upLeft .|. upRight .|. downRight .|. downLeft

-- get the occupancy
generateAllBishopMoves :: Bitboard -> Bitboard -> Bitboard -> [Move]
generateAllBishopMoves bishopBoard captureable occupancy -- after hitting a capturable block all cells behind it fo
 =
  [ Move
    square
    targetSquare
    ""
    (if targetBit .&. captureable /= 0
       then 1
       else 0)
    0
    0
  | square <- bitboardSquares bishopBoard
  , targetSquare <-
      bitboardSquares (generateBishopMoves square occupancy captureable)
  , let targetBit = 1 `shiftL` targetSquare
  ]

-- Queen
-- ################################################################
generateQueenMoves :: Int -> Bitboard -> Bitboard -> Bitboard
generateQueenMoves square occupancy captureable =
  let position = 1 `shiftL` square
      -- bishop moves
      upLeft = generateLine position occupancy captureable 9 8
      upRight = generateLine position occupancy captureable 7 8
      downRight = generateLine position occupancy captureable (-9) 8
      downLeft = generateLine position occupancy captureable (-7) 8
      -- rook moves
      up = generateLine position occupancy captureable 8 8 -- by 8 because we shift straight up so need to skip to
      down = generateLine position occupancy captureable (-8) 8
      right = generateLine position occupancy captureable 1 8
      left = generateLine position occupancy captureable (-1) 8
   in upLeft
        .|. upRight
        .|. downRight
        .|. downLeft
        .|. up
        .|. down
        .|. right
        .|. left
```

```haskell
generateAllQueenMoves :: Bitboard -> Bitboard -> Bitboard -> [Move]
generateAllQueenMoves queenBoard occupancy captureable =
  [ Move
    square
    targetSquare
    ""
    (if targetBit .&. captureable /= 0
       then 1
       else 0)
    0
    0
  | square <- bitboardSquares queenBoard
  , targetSquare <-
      bitboardSquares (generateQueenMoves square occupancy captureable)
  , let targetBit = 1 `shiftL` targetSquare
  ]

generateAllQueenMovesT :: Bitboard -> Bitboard -> Bitboard -> Bitboard
generateAllQueenMovesT queenBoard occupancy captureable =
  foldr (.|.) 0 [generateQueenMoves square occupancy captureable | square <- bitboardSquares queenBoard]


-- King
-- #################################################################
generateKingMoves :: Int -> Bitboard -> Bitboard -> Bitboard
generateKingMoves square occupancy captureable =
  let position = 1 `shiftL` square
      -- bishop moves
      upLeft = generateLine position occupancy captureable 9 1
      upRight = generateLine position occupancy captureable 7 1
      downRight = generateLine position occupancy captureable (-9) 1
      downLeft = generateLine position occupancy captureable (-7) 1
      -- rook moves
      up = generateLine position occupancy captureable 8 1
      down = generateLine position occupancy captureable (-8) 1
      right = generateLine position occupancy captureable 1 1
      left = generateLine position occupancy captureable (-1) 1
   in upLeft
        .|. upRight
        .|. downRight
        .|. downLeft
        .|. up
        .|. down
        .|. right
        .|. left

generateAllKingMoves :: Bitboard -> Bitboard -> Bitboard -> [Move]
generateAllKingMoves kingBoard occupancy captureable =
  [ Move
    square
    targetSquare
    ""
    (if targetBit .&. captureable /= 0
       then 1
       else 0)
    0
    0
  | square <- bitboardSquares kingBoard
  , targetSquare <-
      bitboardSquares (generateKingMoves square occupancy captureable)
  , let targetBit = 1 `shiftL` targetSquare
  ]

-- pawn - capture logic is handled in here
```

```haskell
-- ################################################################
generatePawnMovesW :: Int -> Bitboard -> Bitboard -> Bitboard
generatePawnMovesW square occupancy captureable =
  let singleStepTarget = square + 8
      canMoveOneStep =
        singleStepTarget < 64
          && ((occupancy `shiftR` singleStepTarget) .&. 1) == 0
          && ((captureable `shiftR` singleStepTarget) .&. 1) == 0
      oneStepMask =
        if canMoveOneStep
          then (1 `shiftL` singleStepTarget)
          else 0
      doubleStepTarget = square + 16
      canMoveTwoSteps =
        canMoveOneStep
          && (square `div` 8 == 1)
          && doubleStepTarget < 64
          && ((occupancy `shiftR` doubleStepTarget) .&. 1) == 0
      twoStepMask =
        if canMoveTwoSteps
          then (1 `shiftL` doubleStepTarget)
          else 0
      file = square `mod` 8
      leftCaptureTarget = square + 7
      canCaptureLeft =
        file /= 0
          && leftCaptureTarget < 64
          && ((captureable `shiftR` leftCaptureTarget) .&. 1) == 1
      leftCaptureMask =
        if canCaptureLeft
          then (1 `shiftL` leftCaptureTarget)
          else 0
      rightCaptureTarget = square + 9
      canCaptureRight =
        file /= 7
          && rightCaptureTarget < 64
          && ((captureable `shiftR` rightCaptureTarget) .&. 1) == 1
      rightCaptureMask =
        if canCaptureRight
          then (1 `shiftL` rightCaptureTarget)
          else 0
   in oneStepMask .|. twoStepMask .|. leftCaptureMask .|. rightCaptureMask

generateAllPawnMovesW :: Bitboard -> Bitboard -> Bitboard -> [Move]
generateAllPawnMovesW pawnBoard occupancy captureable =
  [ Move
    square
    targetSquare
    ""
    (if targetBit .&. captureable /= 0
       then 1
       else 0)
    0
    0
  | square <- bitboardSquares pawnBoard
  , targetSquare <-
      bitboardSquares (generatePawnMovesW square occupancy captureable)
  , let targetBit = 1 `shiftL` targetSquare
  ]

-- ################################################################
generatePawnMovesB :: Int -> Bitboard -> Bitboard -> Bitboard
generatePawnMovesB square occupancy captureable =
  let file = square `mod` 8
      singleStepTarget = square - 8
```

```haskell
      canMoveOneStep =
        singleStepTarget >= 0
          && ((occupancy `shiftR` singleStepTarget) .&. 1) == 0
      oneStepMask =
        if canMoveOneStep
          then (1 `shiftL` singleStepTarget)
          else 0
      doubleStepTarget = square - 16
      canMoveTwoSteps =
        canMoveOneStep
          && (square `div` 8 == 6)
          && doubleStepTarget >= 0
          && ((occupancy `shiftR` doubleStepTarget) .&. 1) == 0
      twoStepMask =
        if canMoveTwoSteps
          then (1 `shiftL` doubleStepTarget)
          else 0
      leftCaptureTarget = square - 9
      canCaptureLeft =
        file /= 0
          && leftCaptureTarget >= 0
          && ((captureable `shiftR` leftCaptureTarget) .&. 1) == 1
      leftCaptureMask =
        if canCaptureLeft
          then (1 `shiftL` leftCaptureTarget)
          else 0
      rightCaptureTarget = square - 7
      canCaptureRight =
        file /= 7
          && rightCaptureTarget >= 0
          && ((captureable `shiftR` rightCaptureTarget) .&. 1) == 1
      rightCaptureMask =
        if canCaptureRight
          then (1 `shiftL` rightCaptureTarget)
          else 0
   in oneStepMask .|. twoStepMask .|. leftCaptureMask .|. rightCaptureMask

generateAllPawnMovesB :: Bitboard -> Bitboard -> Bitboard -> [Move]
generateAllPawnMovesB pawnBoard occupancy captureable =
  [ Move
    square
    targetSquare
    ""
    (if targetBit .&. captureable /= 0
       then 1
       else 0)
    0
    0
  | square <- bitboardSquares pawnBoard
  , targetSquare <-
      bitboardSquares (generatePawnMovesB square occupancy captureable)
  , let targetBit = 1 `shiftL` targetSquare
  ]

-- Helper functions for sliding piece
-- #################################################################
generateLine :: Bitboard -> Bitboard -> Bitboard -> Int -> Int -> Bitboard
generateLine position occupancy captureable step limit =
  let next =
        if step > 0
          then position `shiftL` step
          else position `shiftR` (-step) -- shiftL only trakes positive
   in if next == 0
           || invalidShift position step
           || canCapture position captureable
```

```
          || isBlocked position occupancy step
          || limit == 0
        then 0
        else next .|. generateLine next occupancy captureable step (limit - 1)

canCapture :: Bitboard -> Bitboard -> Bool
canCapture position captureable = (position .&. captureable) /= 0

isBlocked :: Bitboard -> Bitboard -> Int -> Bool
isBlocked position occupancy step =
  (step > 0 && (position `shiftL` step) .&. occupancy /= 0)
    || (step < 0 && (position `shiftR` (-step)) .&. occupancy /= 0)

-- check if we are overshooting the columns
invalidShift :: Bitboard -> Int -> Bool
invalidShift position step =
  (step > 0
     && (position `shiftL` step == 0
          || crossesLeftBoundary position step fileA
          || crossesDiagonalBoundary position step fileA fileH))
    || (step < 0
         && (position `shiftR` (-step) == 0
              || crossesRightBoundary position (-step) fileH
              || crossesDiagonalBoundary position step fileA fileH))

-- Helper functions for boundary checks
crossesLeftBoundary :: Bitboard -> Int -> Bitboard -> Bool
crossesLeftBoundary position step _ = step == 1 && (position .&. fileH /= 0)

crossesRightBoundary :: Bitboard -> Int -> Bitboard -> Bool
crossesRightBoundary position step _ =
  step == -1 && (position .&. fileA /= 0)

crossesDiagonalBoundary :: Bitboard -> Int -> Bitboard -> Bitboard -> Bool
crossesDiagonalBoundary position step fileAl fileHl
  | step == 9 || step == -7 = position .&. fileHl /= 0 -- Moving left and wrapping to file H
  | step == 7 || step == -9 = position .&. fileAl /= 0 -- Moving right and wrapping to file A
  | otherwise = False
```

6.4 Fen.hs

[htbp]

```
        module Fen where

-- ##################################################################
data FEN = FEN
  { board_fen :: String
  , activeColor :: Char
  , castling :: String
  , enPassant :: String
  , halfMoveClock :: Int
  , fullMoveNumber :: Int
  }

instance Show FEN where
  show fen =
    board_fen fen
      ++ " "
      ++ [activeColor fen]
      ++ " "
      ++ castling fen
      ++ " "
```

```
      ++ enPassant fen
      ++ " "
      ++ show (halfMoveClock fen)
      ++ " "
      ++ show (fullMoveNumber fen)

-- call to set the default setting of the fen notation of the engine
setDefaultFen :: FEN
setDefaultFen =
  FEN
    { board_fen = "rnbqkbnr/pppppppp/8/8/8/8/PPPPPPPP/RNBQKBNR" -- Standard starting position
    , activeColor = 'w' -- White to move
    , castling = "KQkq" -- Both sides can castle
    , enPassant = "-" -- No en passant square
    , halfMoveClock = 0 -- No halfmoves yet
    , fullMoveNumber = 1 -- First move
    }
```