Logisch Programmeren

Project Schaakcomputer

Pieter De Clercq

25 juni 2018

UNIVERSITEIT GENT

FACULTEIT WETENSCHAPPEN

Inleiding

In dit verslag bespreek ik mijn implementatie van een schaakcomputer in Prolog, de projectopgave voor het vak Logisch Programmeren. Eerst zal ik bespreken hoe een schaakbord intern wordt voorgesteld in de schaakcomputer. Vervolgens overloop ik de belangrijkste facetten van het algoritme dat bepaalt wat de beste volgende zet is. Tot slot bespreek ik het ontwikkelingsproces van deze schaakcomputer, duid ik aan waar de moeilijkheden lagen en waar er nog ruimte is voor verbeteringen, zowel voor snelheid als voor het verhogen van de winstkansen.

Hoofdstuk 1

Bordrepresentatie

Parser

Het verwerken van een FEN-string gebeurt door middel van een DCG. Dit heeft als voordeel dat de parser, zonder verdere aanpassingen, bidirectioneel werkt. Dezelfde code die een FEN-string naar een bord kan omzetten, kan dus ook gebruikt worden om een bordrepresentatie om te zetten naar een FEN-string. Het is eveneens mogelijk om via deze DCG alle mogelijke geldige schaakborden te genereren. Tijdens het opstellen van de parser werd hiervoor gekozen omdat dit nuttig leek tijdens het zoeken naar de beste volgende zet, maar uiteindelijk heb ik dit niet gedaan wegens performantieredenen.

Schaakbord

Een schaakbord wordt voorgesteld als een lijst van 8 rijen. Deze rijen komen overeen met de rijen van een echt schaakbord, van onder naar boven. De rij met index 0 in de lijst komt dus overeen met de onderste rij van het schaakbord. Dit is belangrijk om te noteren, aangezien in FEN-notatie het bord van boven naar onder wordt voorgesteld. Het feit dat de rijen vanaf 0 worden geïndexeerd zorgt niet voor problemen dankzij het *nth1/3* predicaat, dat indexfouten door de programmeur uitsluit.

$$bord = \{rij_1, rij_2, rij_3, rij_4, rij_5, rij_6, rij_7, rij_8\}$$

Elke rij is op zijn beurt nogmaals onderverdeeld in 8 vakjes, deze komen overeen met respectievelijk de kolommen A tot en met G van het schaakbord. De waarde van een vakje is ofwel het schaakstuk dat op die positie staat, ofwel *none* wanneer het vakje leeg is.

$$rij = \{piece_A, none, piece_C, none, piece_E, none, piece_F, none\}$$

Schaakstukken

Schaakstukken worden voorgesteld door middel van een tuple $piece(Type, Kleur)$, waarbij $Type \in \{bishop, king, knight, pawn, queen, rook\}$ en $Kleur \in \{black, white\}$. Op deze manier kan via unificatie snel worden gezocht naar alle stukken van een bepaald type, of van een bepaalde speler.

Voorbeeld

$$piece = piece(king, white)$$

Hoofdstuk 2

Algoritme

De basis van het algoritme van mijn schaakcomputer is gebaseerd op minimax-bomen. Op deze minimax-bomen heb ik alpha-beta snoeien toegepast om de hoeveelheid takken te verkleinen, waardoor de snelheid omhoog gaat en er dieper gezocht kan worden. Vertrekkend van deze basis heb ik nog twee aanpassingen toegepast, welke ik in onderstaande secties zal bespreken.

Scorefunctie

Om de beste zet te kunnen bepalen, is het noodzakelijk dat twee zetten met elkaar kunnen worden vergeleken. Hiervoor wordt een scorefunctie geïmplementeerd. Deze scorefunctie is gebaseerd op het aantal stukken van de speler en het aantal stukken van de tegenstander, waarbij aan elk stuk een andere waarde wordt toegekend, zodanig dat bepaalde stukken belangrijker zijn dan andere. Dankzij deze scorefunctie zal over het algemeen de schaakcomputer steeds kiezen om zo snel mogelijk stukken van de tegenstander te slaan.

Stuk	**Waarde**
bishop	3.000
king	$1.000.000$
knight	12.000
pawn	1.000
queen	25.000
rook	5.000

Tabel 2.1: Waarde per stuk in de scorefunctie

Het valt op dat de koning een zeer hoge waarde heeft. Dit is noodzakelijk omdat tijdens het zoeken naar volgende zetten, niet wordt gecontroleerd of de speler schaak staat, dit zou namelijk een grote impact hebben op de snelheid. In plaats daarvan wordt er vanuit gegaan dat, indien de speler schaak zou staan, in de volgende zet de tegenstander de koning zou slaan. Dit is het geval dankzij de uitzonderlijk hoge waarde van de koning. Door de minimax functie zullen zetten waarbij de koning geslaan wordt niet gedaan worden, tenzij het niet anders kan, namelijk wanneer de speler schaakmat staat.

Algorithm 1 Scorefunctie

```
function SCORE(situatie)
    mijn_score ← SCORE_SUB(situatie, huidige_speler)
    tegenstander_score ← SCORE_SUB(situatie, tegenstander)
    return mijn_score − tegenstander_score
function SCORE_SUB(situatie, speler)
    stukken ← alle stukken van speler
    scores_i ← WAARDE(stukken_i)
    return SOM(scores)
```

Basisalgoritme: Minimax-bomen

Zoals reeds eerder vermeld vormen minimax-bomen de kern van het schaakalgoritme. Vanuit de huidige spelsituatie (het huidige bord, de rokademogelijkheden, …), worden alle mogelijke volgende spelsituaties bepaald. Voor elke volgende spelsituatie wordt dit algoritme nog eens herhaald, tot een vooraf ingestelde diepte. Om de tijdsduur binnen de perken te houden werd ervoor gekozen deze diepte te begrenzen op 3. Hieronder volgt een schets van het minimax-algoritme.

Algorithm 2 Minimax basisalgoritme

```
function MINIMAX(situatie, diepte)
    situaties ← VOLGENDE_SPELSITUATIES(situatie)
    if diepte == 0 of situaties == ∅ then
        return situatie, SCORE(situatie)
    beurt ← state.beurt
    if beurt == mijn_speler then
        beste_score ← −∞
        beste_situatie ← situatie
        for all kandidaat ∈ situaties do
            st, sc ← MINIMAX(kandidaat, diepte − 1)
            if sc > beste_score then
                beste_score ← sc
                beste_situatie ← st
        return beste_situatie, beste_score
    else
        beste_score ← +∞
        beste_situatie ← situatie
        for all kandidaat ∈ states do
            st, sc ← MINIMAX(kandidaat, diepte − 1)
            if sc < beste_score then
                beste_score ← sc
                beste_situatie ← st
        return beste_situatie, beste_score
```

Alpha-beta snoeien

Het basis minimax algoritme werkt perfect om de volgende zet te bepalen, maar heeft als nadeel dat de boom exponentieel groter wordt naarmate er dieper wordt gezocht. Vooral in het midden van het spel kan dit voor problemen zorgen aangezien er dan zeer veel geldige zetten zijn. Een manier om de grootte van de boom te beperken, is alpha-beta snoeien toepassen op de boom. De tijd om een volgende zet te bepalen werd hiermee gemiddeld van 30 seconden verminderd naar 5. Het minimax algoritme dient hiervoor aangepast te worden, zodat niet enkel de score in rekening wordt gebracht, maar ook de factoren α en β.

Algorithm 3 Minimax met alpha-beta snoeien

```
function ALPHABETA(situatie, diepte, α, β)
    situaties ← VOLGENDE_SPELSITUATIES(situatie)
    if diepte == 0 of situaties == ∅ then
        return situatie, SCORE(situatie)
    beurt ← state.beurt
    if beurt == mijn_speler then
        beste_score ← −∞
        beste_situatie ← situatie
        for all kandidaat ∈ situaties do
            st, sc ← ALPHABETA(kandidaat, diepte − 1, α, β)
            if sc > beste_score then
                beste_score ← sc
                beste_situatie ← st
                α ← MAX(α, sc)
                if β ≤ α then
                    return beste_situatie, beste_score
        return beste_situatie, beste_score
    else
        beste_score ← +∞
        beste_situatie ← situatie
        for all kandidaat ∈ situaties do
            st, sc ← ALPHABETA(kandidaat, diepte − 1, α, β)
            if sc < beste_score then
                beste_score ← sc
                beste_situatie ← st
                β ← MIN(β, sc)
                if β ≤ α then
                    return beste_situatie, beste_score
        return beste_situatie, beste_score
```

Uitbreiding 1: Randomisatie bij gelijke scores

Tijdens het testen van het schaakalgoritme viel onmiddellijk op dat de schaakcomputer steeds dezelfde zetten deed. Dit is logisch aangezien het vaak voorkomt dat er veel zetten zijn die dezelfde score opleveren, voornamelijk bij het begin van een partij. Hoewel dit an sich geen probleem vormt, kan het zijn dat de tegenstander zijn schaakcomputer zodanig programmeert om hier rekening mee te houden. Om dit te omzeilen werd het alphabeta algoritme uit de vorige sectie aangepast om een niet-deterministische factor in rekening te brengen, als volgt:

Algorithm 4 Minimax met alpha-beta snoeien en randomisatie

```
function ALPHABETA(situatie, diepte, α, β)
    ⋮
    if beurt == mijn_speler then
        ⋮
        for all kandidaat ∈ situaties do
            st, sc ← ALPHABETA(kandidaat, diepte − 1, α, β)
            pick_equal ← RANDOM(0, 1) < 0.5
            if sc > beste_score of (sc == beste_score en pick_equal) then
                beste_score ← sc
                ⋮
        return beste_situatie, beste_score
    else
        ⋮
        for all kandidaat ∈ situaties do
            st, sc ← ALPHABETA(kandidaat, diepte − 1, α, β)
            pick_equal ← RANDOM(0, 1) < 0.5
            if sc < beste_score of (sc == beste_score en pick_equal) then
                beste_score ← sc
                ⋮
        return beste_situatie, beste_score
```

Uitbreiding 2: Dynamische diepte

Zoals eerder vermeld is de diepte van de zoekbomen beperkt tot 3, wegens performantieredenen. Het kan echter voorkomen, voornamelijk naar het einde van het spel toe, dat het aantal mogelijke zetten sterk vermindert. In dat geval zal de zoekboom ook veel kleiner zijn, waardoor naar een grotere diepte kan worden gezocht om mogelijks de tegenstander sneller mat te zetten. Dit kan eenvoudig worden geïmplementeerd in het vorige minimax algoritme.

Algorithm 5 Minimax met alpha-beta snoeien, randomisatie en dynamische diepte

```
function ALPHABETA(situatie, diepte, α, β)
    ⋮
    if beurt == mijn_speler then
        ⋮
        for all kandidaat ∈ situaties do
            nieuwediepte ← DYNAMISCHE_DIEPTE(situaties, diepte)
            st, sc ← ALPHABETA(kandidaat, nieuwediepte, α, β)
            pick_equal ← RANDOM(0, 1) < 0.5
            if sc > beste_score of (sc == beste_score en pick_equal) then
                beste_score ← sc
                ⋮
        return beste_situatie, beste_score
    else
        ⋮
        for all kandidaat ∈ situaties do
            nieuwediepte ← DYNAMISCHE_DIEPTE(situaties, diepte)
            st, sc ← ALPHABETA(kandidaat, nieuwediepte, α, β)
            pick_equal ← RANDOM(0, 1) < 0.5
            if sc < beste_score of (sc == beste_score en pick_equal) then
                beste_score ← sc
                ⋮
        return beste_situatie, beste_score
function DYNAMISCHE_DIEPTE(situaties, diepte)
    if situaties.length < 4 then
        return diepte + 1
    return diepte
```

Hoofdstuk 3

Conclusie

Tijdens dit project werd een schaakcomputer geïmplementeerd in Prolog. Het algoritme om de beste volgende zet te bepalen is gebaseerd op een variant van minimax-bomen, aangevuld met twee uitbreidingen. Deze uitbreidingen bestaan uit zowel randomisatie, als dynamische diepte. Het ontwikkelingsproces ging relatief vlot, alhoewel debuggen niet altijd even gemakkelijk verliep. Verdere uitbreidingen aan het schaakalgoritme zijn zeker mogelijk, rekeninghoudend met het feit dat professionele schaakcomputers veel dieper gaan dan diepte 3. Deze dieptes kunnen voornamelijk worden bereikt door code optimalisaties alsook betere score functies. Zelf had ik echter geen schaakervaring, dus de huidige scorefunctie leek de meest voordehandliggende. Het werken aan dit project heeft mij veel zinvolle inzichten gegeven in logisch programmeren en in schaken, in de toekomst ben ik zeker van plan Prolog nog te gebruiken voor toepassingen rond artificiële intelligentie.

Appendix: Broncode

src/board.pl

```
:- module(board, []).

:- use_module(state).

% A list of all the possible castling squares in this order:
% (castling_type, KingInitial, RookInitial, KingDestination, RookDestination).
castling_squares(castling(kingside, black), 8/5, 8/8, 8/7, 8/6).
castling_squares(castling(kingside, white), 1/5, 1/8, 1/7, 1/6).
castling_squares(castling(queenside, black), 8/5, 8/1, 8/3, 8/6).
castling_squares(castling(queenside, white), 1/5, 1/1, 1/3, 1/4).

%% check(+Board: board, +Player: turn).
%
%  Validates the given player is in check.
%
%  @param Board the board
%  @param Player the player to verify
check(Board, Player) :-
  % Get the location of the player's king.
  piece_at(Board, KingSquare, piece(king, Player)),
  % Get the enemy of the given player.
  state:enemy(Player, Enemy),
  % Verify that the enemy is attacking the king of the given player.
  state:attacking(Board, Enemy, KingSquare), !.

%% clear(+Board: board, +R/C: square, -After: board).
%
%  Removes a piece from the board.
%
%  @param Before the initial board
%  @param R/C the square at which the piece should be removed
%  @param After the resulting board
clear(Before, R/C, After) :-
  nth1_row(Before, R, RowBefore),
  piece_replace(RowBefore, C, none, RowAfter),
  row_replace(Before, R, RowAfter, After).
```

```prolog

%% enemy(+Board: board, +Square: square, +turn).
%
%  Validates the given square contains any piece that belongs to the given
%  player's enemy.
%
%  @param Board the board
%  @param Square the square to check
%  @param black-white the player
enemy(Board, Square, black) :- piece_at(Board, Square, piece(_, white)).
enemy(Board, Square, white) :- piece_at(Board, Square, piece(_, black)).

%% free(+Board: board, +R/C: square).
%
%  Succeeds if the given square does not contain any piece.
%
%  @param Board the board
%  @param R/C the square
free(Board, R/C) :-
  % Extract the row from the board.
  nth1_row(Board, R, Row),
  % Verify the piece at the given square is none.
  nth1_piece(Row, C, none).

%% mine(+Board: board, +Square: square, +Player: turn).
%
%  Validates the given square contains any piece that belongs to the given
%  player.
%
%  @param Board the board
%  @param Square the square to check
%  @param Player the player to match
mine(Board, Square, Player) :- piece_at(Board, Square, piece(_, Player)).

%% move_piece(+Before: board, +From: square, +To: square, -After: board).
%
%  Moves a piece on the board.
%
%  @param Before the initial board
%  @param From the current square
%  @param To the destination square
%  @param After the resulting board
```

```
move_piece(Before, From, To, After) :-
  % Get the piece to replace.
  piece_at(Before, From, Piece),
  % Remove the piece from the square.
  set_piece(Before, From, none, Removed),
  % Put the piece at the destination square.
  set_piece(Removed, To, Piece, After).

%% nth1_piece(+board:board, +integer, +R: -R: row).
%
%  Gets the row at the given row number on the board.
%
%  @param board the Board
%  @param integer the row number
%  @param R the row at that row number
nth1_row(board(R, _, _, _, _, _, _, _), 1, R).
nth1_row(board(_, R, _, _, _, _, _, _), 2, R).
nth1_row(board(_, _, R, _, _, _, _, _), 3, R).
nth1_row(board(_, _, _, R, _, _, _, _), 4, R).
nth1_row(board(_, _, _, _, R, _, _, _), 5, R).
nth1_row(board(_, _, _, _, _, R, _, _), 6, R).
nth1_row(board(_, _, _, _, _, _, R, _), 7, R).
nth1_row(board(_, _, _, _, _, _, _, R), 8, R).

%% nth1_piece(+row:row, +integer, +R: -P: piece).
%
%  Gets the piece at the given column in the row.
%
%  @param row the row
%  @param integer the column number
%  @param P the piece at that position
nth1_piece(row(P, _, _, _, _, _, _, _), 1, P).
nth1_piece(row(_, P, _, _, _, _, _, _), 2, P).
nth1_piece(row(_, _, P, _, _, _, _, _), 3, P).
nth1_piece(row(_, _, _, P, _, _, _, _), 4, P).
nth1_piece(row(_, _, _, _, P, _, _, _), 5, P).
nth1_piece(row(_, _, _, _, _, P, _, _), 6, P).
nth1_piece(row(_, _, _, _, _, _, P, _), 7, P).
nth1_piece(row(_, _, _, _, _, _, _, P), 8, P).

%% piece_at(+Board:board, +R/C: square, -Piece: piece).
%
%  Gets the piece at a given square;
%
%  @param Board the board
%  @param R/C the square
%  @param Piece the piece at that square.
piece_at(Board, R/C, Piece) :-
```

```
  % Extract the row of the piece from the board.
  nth1_row(Board, R, Row),
  % Extract the column of the piece from the row.
  nth1_piece(Row, C, Piece).

%% piece_replace(+row:row, +integer, +P: Piece, -row:row).
%
%  Replaces a piece in a row.
%
%  @param row the initial row
%  @param integer the column number to replace
%  @param P the new piece to put on the given column
%  @param row the resulting row
piece_replace(row(_, P2, P3, P4, P5, P6, P7, P8), 1, P,
   row(P, P2, P3, P4, P5, P6, P7, P8)).
piece_replace(row(P1, _, P3, P4, P5, P6, P7, P8), 2, P,
   row(P1, P, P3, P4, P5, P6, P7, P8)).
piece_replace(row(P1, P2, _, P4, P5, P6, P7, P8), 3, P,
   row(P1, P2, P, P4, P5, P6, P7, P8)).
piece_replace(row(P1, P2, P3, _, P5, P6, P7, P8), 4, P,
   row(P1, P2, P3, P, P5, P6, P7, P8)).
piece_replace(row(P1, P2, P3, P4, _, P6, P7, P8), 5, P,
   row(P1, P2, P3, P4, P, P6, P7, P8)).
piece_replace(row(P1, P2, P3, P4, P5, _, P7, P8), 6, P,
   row(P1, P2, P3, P4, P5, P, P7, P8)).
piece_replace(row(P1, P2, P3, P4, P5, P6, _, P8), 7, P,
   row(P1, P2, P3, P4, P5, P6, P, P8)).
piece_replace(row(P1, P2, P3, P4, P5, P6, P7, _), 8, P,
   row(P1, P2, P3, P4, P5, P6, P7, P)).

%% row_replace(+board:board, +integer, +R: row,
   -board:board).
%
%  Replaces a row on the board.
%
%  @param board the initial board
%  @param integer the row number to replace
%  @param R the new row to put on the given row number
%  @param board the resulting board
row_replace(board(_, R2, R3, R4, R5, R6, R7, R8), 1, R,
   board(R, R2, R3, R4, R5, R6, R7, R8)).
row_replace(board(R1, _, R3, R4, R5, R6, R7, R8), 2, R,
   board(R1, R, R3, R4, R5, R6, R7, R8)).
row_replace(board(R1, R2, _, R4, R5, R6, R7, R8), 3, R,
   board(R1, R2, R, R4, R5, R6, R7, R8)).
row_replace(board(R1, R2, R3, _, R5, R6, R7, R8), 4, R,
   board(R1, R2, R3, R, R5, R6, R7, R8)).
```

```
row_replace(board(R1, R2, R3, R4, _, R6, R7, R8), 5, R, board(R1, R2, R3, R4, R, R6, R7, R8)).
row_replace(board(R1, R2, R3, R4, R5, _, R7, R8), 6, R, board(R1, R2, R3, R4, R5, R, R7, R8)).
row_replace(board(R1, R2, R3, R4, R5, R6, _, R8), 7, R, board(R1, R2, R3, R4, R5, R6, R, R8)).
row_replace(board(R1, R2, R3, R4, R5, R6, R7, _), 8, R, board(R1, R2, R3, R4, R5, R6, R7, R)).

%% set_piece(+Before:board, +R/C: square, +Piece:piece, -After:board).
%
%  Sets the given piece on the board at square R/C.
%
%  @param Before the initial board
%  @param R/C the square to put the piece on
%  @param Piece the piece to put
%  @param After the resulting board
set_piece(Before, R/C, Piece, After) :-
  % Fetch the row to put the piece on.
  nth1_row(Before, R, RowBefore),
  % Put the piece on the correct row.
  piece_replace(RowBefore, C, Piece, RowAfter),
  % Put the modified row in the resulting board.
  row_replace(Before, R, RowAfter, After), !.

%% square(R/C).
%
%  Validates or generates a square on the playfield.
%
%  @param R/C the row and column of the square
square(R/C) --> between(1, 8, R), between(1, 8, C).
square(R/C) :- between(1, 8, R), between(1, 8, C).
```

src/fen.pl

```
:- module(fen, []).

:- use_module(library(dcg/basics)).

%% parse(FenString:string, State:state).
%
%  Parses a given FEN string into a state. Can also perform the reverse
%  conversion operation.
%
%  @param FenString the FEN string
```

```
%  @param State the corresponding state
parse(FenString, State) :-
  % Apply the DCG to the string to obtain the state, or reverse.
  phrase(state(State), FenString).

% DCG parser for a state, based on the FEN specifications.
state(state(Board, Turn, Castling, EnPassant, HalfCount, FullCount)) -->
  board(Board), "␣", turn(Turn), "␣", castlings(Castling),
  "␣", en_passant(EnPassant), "␣", half_count(HalfCount),
  "␣", full_count(FullCount).

% DCG parser for a board.
board(board(R1, R2, R3, R4, R5, R6, R7, R8)) -->
  row(R8),"/",row(R7),"/",row(R6),"/",row(R5),
  "/",row(R4),"/",row(R3),"/",row(R2),"/",row(R1).

% DCG parser for castling.
castlings([]) --> "-", !.
castlings(Cs) --> castling_possibilities(Cs).
% castlings[] kan niet opnieuw gebruikt worden want dan zou K-kq ook geldig zijn
castling_possibilities([]) --> [].
castling_possibilities([C | Cs]) --> castling(C), castling_possibilities(Cs).

% DCG parser for the castling possibilities.
castling(castling(kingside, black)) --> "k".
castling(castling(kingside, white)) --> "K".
castling(castling(queenside, black)) --> "q".
castling(castling(queenside, white)) --> "Q".

% DCG parser for the en passant squares.
en_passant(none) --> "-".
en_passant(3/1) --> "a3".
en_passant(3/2) --> "b3".
en_passant(3/3) --> "c3".
en_passant(3/4) --> "d3".
en_passant(3/5) --> "e3".
en_passant(3/6) --> "f3".
en_passant(3/7) --> "g3".
en_passant(3/8) --> "h3".
en_passant(6/1) --> "a6".
en_passant(6/2) --> "b6".
en_passant(6/3) --> "c6".
en_passant(6/4) --> "d6".
en_passant(6/5) --> "e6".
```

```
en_passant(6/6) --> "f6".
en_passant(6/7) --> "g6".
en_passant(6/8) --> "h6".

% DCG parser for the full-move counter.
full_count(N) --> integer(N), {N > 0}.

% DCG parser for the half-move counter.
half_count(N) --> integer(N), {N >= 0}.

% DCG parser for pieces.
piece(piece(bishop, black)) --> "b".
piece(piece(bishop, white)) --> "B".
piece(piece(king, black)) --> "k".
piece(piece(king, white)) --> "K".
piece(piece(knight, black)) --> "n".
piece(piece(knight, white)) --> "N".
piece(piece(pawn, black)) --> "p".
piece(piece(pawn, white)) --> "P".
piece(piece(queen, black)) --> "q".
piece(piece(queen, white)) --> "Q".
piece(piece(rook, black)) --> "r".
piece(piece(rook, white)) --> "R".

% DCG parser for pieces on a row.
piece(Piece, Left, Left1) --> piece(Piece), {Left1 is
    Left-1, Left1 >= 0}.

% DCG parser for pieces on a row.
pieces([], 0) --> [].

% DCG parser for empty squares on a row.
pieces([none, none, none, none, none, none, none, none], 8)
    --> "8".
pieces([none, none, none, none, none, none, none | X], I)
    --> "7", !,
  {I1 is I-7, I1 >= 0},
  pieces(X, I1).
pieces([none, none, none, none, none, none | X], I) --> "6",
    !,
  {I1 is I-6, I1 >= 0},
  pieces(X, I1).
pieces([none, none, none, none, none | X], I) --> "5", !,
  {I1 is I-5, I1 >= 0},
  pieces(X, I1).
pieces([none, none, none, none | X], I) --> "4", !,
  {I1 is I-4, I1 >= 0},
  pieces(X, I1).
```

```
pieces([none, none, none | X], I) --> "3", !,
  {I1 is I-3, I1 >= 0},
  pieces(X, I1).
pieces([none, none | X], I) --> "2", !,
  {I1 is I-2, I1 >= 0},
  pieces(X, I1).
pieces([none | X], I) --> "1", !,
  {I1 is I-1, I1 >= 0},
  pieces(X, I1).

% DCG parser for pieces on a row.
pieces([H|R], Left) --> piece(H, Left, Left1), pieces(R,
   Left1).

% DCG parser (FEN to state) for a row.
row(row(A, B, C, D, E, F, G, H)) --> {var(A), !},
  pieces(Row, 8),
  {
    flatten(Row, [A, B, C, D, E, F, G, H])
  }.

% DCG parser (state to FEN) for a row.
row(row(A, B, C, D, E, F, G, H)) --> {nonvar(A), !},
  pieces([A, B, C, D, E, F, G, H], 8).

% DCG parser for turns.
turn(black) --> "b".
turn(white) --> "w".
```

src/main.pl

```
#!/usr/bin/env swipl

:- use_module(fen).
:- use_module(minimax).
:- use_module(movement).
:- use_module(state).

% Set the correct string mode.
:- set_prolog_flag(double_quotes, chars).
% Increase the stack limit to allow deeper searches.
:- set_prolog_flag(stack_limit, 2 000 000 000).
% Set the initialization goal.
:- initialization(main, main).

%% parse(+Argv:list, -State:state).
%
```

```
%  Parses the given FEN input arguments into a state.
%
%  @param Argv the arguments to parse
%  @param State the resulting state
parse(Argv, State) :-
  % Concatenate all the separate arguments into a FEN string.
  atomic_list_concat(Argv, ' ', FenRaw),
  % Convert the string to the correct representation.
  atom_codes(FenRaw, FenString),
  % Parse the FEN string into a state.
  fen:parse(FenString, State).

%% validate(+Player:turn, +State:state).
%
%  Validates a given state. A state is valid if the given player is not in
%  check.
%
%  @param Player the player's king that may not be attacked
%  @param State the state to validate
validate(Player, State) :-
  % Validate the player is not in check.
  \+ state:check(State, Player).

%% write_draw().
%
%  Writes out the "DRAW" message.
write_draw() :-
  % Convert the string "DRAW" to a list of atoms that can be printed.
  atom_codes(Draw, "DRAW"),
  % Print the string.
  write(Draw), nl.

%% write_fen(+State:state).
%
%  Writes out a given state in FEN notation.
%
%  @param State the state to print
write_fen(State) :-
  % Parse the state to its corresponding FEN notation.
  fen:parse(ResultFen, State),
  % Convert the FEN string to atoms that can be printed.
  atom_codes(ResultRaw, ResultFen),
  % Write out the result.
  write(ResultRaw), nl.

%% main(+Argv:list).
```

```
%
%  Regular main function that prints out the best move. This function will, if
%  successful, halt the program.
%
%  @param Argv the input arguments
main(Argv) :-
  % Precondition for this function: FEN has 6 parts.
  length(Argv, 6),
  % Parse the arguments into a state.
  parse(Argv, State),
  % Disable the garbage collector during calculations.
  set_prolog_flag(gc, false),
  % Perform minimax and alpha-beta pruning on the state to get the next best
  % state for the current player.
  minimax:alphabeta(State, 3, NextState),
  % Re-instantiate the garbage collector.
  set_prolog_flag(gc, false), garbage_collect,
  % Write out the resulting move.
  write_fen(NextState),
  % Halt execution.
  halt(0).

%% main(+Argv:list).
%
%  Testing main function that prints out all next moves. This function will, if
%  successful, halt the program.
%
%  @param Argv the input arguments
main(Argv) :-
  % Precondition: there should be 6 FEN arguments and 1 TEST argument.
  length(Argv, 7),
  % Remove the TEST argument from the input.
  exclude(=('TEST'), Argv, ArgvClean),
  % Parse the arguments into a state.
  parse(ArgvClean, State),
  % Get the current player.
  state:turn(State, Player),
  % Generate all possible moves.
  movement:all_moves(State, Moves),
  % Apply all possible moves to obtain the resulting states.
  maplist(state:apply_move(State), Moves, ResultStates),
  % Remove the states where the player is in check.
  include(validate(Player), ResultStates, ValidStates),
  % Verify at least one legal move is available.
```

```
  length(ValidStates, AmountStates), AmountStates > 0,
  % Write the FEN representation for every valid state.
  maplist(write_fen(), ValidStates), !,
  % Halt execution.
  halt(0).

%% main(+Argv:list).
%
%  Main function that prints out a draw. A draw is assumed if the regular and
%  testing main functions have not found any valid moves.
%
%  @param Argv the input arguments
main(Argv) :-
  % The amount of input arguments must be 6 or 7 (testing).
  (length(Argv, 6) ; length(Argv, 7)),
  % Write out a draw.
  write_draw(),
  % Halt execution.
  halt(0).
```

src/minimax.pl

```
:- module(minimax, []).

:- use_module(movement).
:- use_module(state).

%% adjust_depth(+Moves: list, +Depth: integer, -NewDepth: integer).
%
%  Adjusts the depth based on the amount of moves. If the amount of moves is
%  very small, the tree is allowed to go deeper one more level to possibly
%  choose a better move.
%
%  @param Moves the moves
%  @param Depth the current depth
%  @param the adjusted depth
adjust_depth(Moves, Depth, NewDepth) :-
  % Get the amount of moves.
  length(Moves, I),
  I < 4,
  NewDepth is Depth + 1, !.
adjust_depth(_, Depth, Depth).

```

```prolog
%% alphabeta(+InitialState: state, +MaxDepth: integer, -BestState: state).
%
% Entry point for the alpha-beta pruning minimax function.
%
% @param InitialState the current state
% @param MaxDepth the maximum depth to search
% @param BestState the best move
alphabeta(InitialState, MaxDepth, BestState) :-
  % Get the current turn frmo the game state.
  state:turn(InitialState, Player),
  % Start the alpha-beta pruning.
  alphabeta(InitialState, Player, MaxDepth, -999 999 999, +999 999 999, BestState, _).

%% alphabeta(+State: state, +Player: player, +Depth: integer, +Alpha: integer,
%%          +Beta: integer, -BestNextState: state, -BestScore: integer).
%
% Alpha-beta pruning function.
%
% @param State the current state
% @param Player the current player
% @param Depth the maximum remaining depth to search
% @param Alpha the alpha bound
% @param Beta the beta bound
% @param BestNextState the best move
% @param BestScore the best move's score
alphabeta(State, Player, 0, _, _, _, Score) :-
  % Base case, don't search any further.
  score(State, Player, Score), !.
alphabeta(Current, Player, Depth, Alpha, Beta, BestNextState, BestScore) :-
  % Generate all next moves.
  movement:all_moves(Current, NextMoves),
  % Get the adjusted depth.
  adjust_depth(NextMoves, Depth, NewDepth),
  % Apply all next moves.
  maplist(state:apply_move(Current), NextMoves, NextStates),
  % Find the best move.
  best(NextStates, Player, NewDepth, Alpha, Beta, BestNextState, BestScore), !.
alphabeta(State, Player, _, _, _, _, Score) :-
  % No more moves are available, return the score of the current state.
  score(State, Player, Score).
```

```
%% best(+States: list, +Player: player, +Depth: integer, +Alpha: integer,
%%      +Beta: integer, -BestState: state, -BestScore: integer).
%
%  Evaluate all states and get the best state (recursive step).
%
%  @param States the states to evaluate
%  @param Player the current player
%  @param Depth the current depth
%  @param Alpha the alpha bound
%  @param Beta the beta bound
%  @param BestState the best move
%  @param BestScore the best score
best([State1 | States], Player, Depth, Alpha, Beta, BestState, BestScore) :-
  % Decrease the depth.
  Deeper is Depth - 1,
  % Perform alpha-beta pruning on the next depth.
  alphabeta(State1, Player, Deeper, Alpha, Beta, _, Score1),
  % Evaluate to find the best move.
  evaluate(State1, Score1, States, Player, Depth, Alpha, Beta, BestState, BestScore).

%% better_of(+State1: state, +Score1: integer, +State2: state, +Score2: integer,
%%           -BestState: state, -BestScore: integer).
%
%  Get the best state between two states.
%
%  @param State1 the first state
%  @param Score1 the first score
%  @param State2 the second state
%  @param Score2 the second score
%  @param BestState the best move
%  @param BestScore the best score
better_of(State1, Score, _, Score, _, State1, Score) :-
  % Choose either one if both scores are equal.
  random(0, 2, 1), !.
better_of(_, Score, State2, Score, _, State2, Score).
better_of(State1, Score1, _, Score2, Player, State1, Score1) :-
  % Current player's turn -> Maximize.
  turn(State1, Player), Score1 > Score2, !.
better_of(State1, Score1, _, Score2, Player, State1, Score1) :-
  % Enemy's turn -> Minmize.
```

```
  \+ turn(State1, Player), Score1 < Score2, !.
better_of(_, _, State2, Score2, _, State2, Score2).

%% bounds(+Alpha: integer, +Beta: integer, +State: state, +Score: integer,
%%        +Player: turn, -NewAlpha: integer, -NewBeta: integer).
%
%  Update the alpha or beta bound.
%
%  @param Alpha the current alpha
%  @param Beta the current beta
%  @param State the State
%  @param Score the score for the given state
%  @param Player the player
%  @param NewAlpha the updated alpha value
%  @param NewBeta the updated beta value
bounds(Alpha, Beta, State, Score, Player, Score, Beta) :-
  % Verify if the alpha should be updated.
  turn(State, Player), Score > Alpha, !.
bounds(Alpha, Beta, State, Score, Player, Alpha, Score) :-
  % Verify if the alpha should be updated.
  \+ turn(State, Player), Score < Beta, !.
bounds(Alpha, Beta, _, _, _, Alpha, Beta).

%% evaluate(+State: state, +Score: integer, +States: list, +Player: turn,
%%          +Depth: integer, +Alpha: integer, +Beta: integer, -BestState: state,
%%          -BestScore: integer).
%
%  Evaluates the current state and determines the best one.
%
%  @param State the State
%  @param Score the score for the given state
%  @param State other states to evaluate
%  @param Player the player
%  @param Depth the current depth
%  @param Alpha the current alpha
%  @param Beta the current beta
%  @param BestState the best move
%  @param BestScore the best score
evaluate(State, Score, [], _, _, _, _, State, Score) :- !.
evaluate(State, Score, States, Player, Depth, Alpha, Beta, BestState, BestScore) :-
  % Calculate the new alpha and beta bounds.
  bounds(Alpha, Beta, State, Score, Player, NewAlpha, NewBeta),
```

```
  % Determine the next best move.
  best(States, Player, Depth, NewAlpha, NewBeta, State1,
   Score1),
  % Determine the best move between the lower depth and the
   current best.
  better_of(State, Score, State1, Score1, Player, BestState,
   BestScore).

%% piece_score(+Piece: piece, -Score: integer).
%
%  Get the minmax score for a given piece.
%
%  @param Piece the piece
%  @param Score the score
piece_score(bishop, 3 000).
piece_score(king, 1 000 000).
piece_score(knight, 12 000).
piece_score(pawn, 1 000).
piece_score(queen, 25 000).
piece_score(rook, 5 000).

%% score(+State: state, +Player: turn, -Score: integer).
%
%  Evaluate a given state and player.
%
%  @param State the State
%  @param Player the player
%  @param Score the score
score(State, Player, Score) :-
  % Get the enemy of the player.
  state:enemy(Player, Enemy),
  % Get this player's score.
  score_sub(State, Player, MyScore),
  % Get the enemy's score.
  score_sub(State, Enemy, EnemyScore),
  % Calculate the final score.
  Score is MyScore - EnemyScore.

%% score_sub(+State: state, +Player: turn, -Score: integer).
%
%  Evaluate a given state and player, subroutine.
%
%  @param State the State
%  @param Player the player
%  @param Score the score
score_sub(State, Player, Score) :-
  % Find all pieces of the player.
```

```
  findall(Type, state:piece_at(State, _, piece(Type,
   Player)), Pieces),
  % Map each piece to its score.
  maplist(piece_score, Pieces, PieceScores),
  % Calculate the sum of all the scores.
  sum_list(PieceScores, Score).

%% turn(+State: state, -Turn: turn).
%
%  Get the current player, this is the enemy of the player
   in the state, since
%  since the state has the next player (the move is already
   applied).
%
%  @param State the State
%  @param Turn the turn
turn(State, Turn) :-
  % Get the opposite of the turn in the state.
  \+ state:turn(State, Turn), !.
turn(State, Turn) :-
  % Get the turn from the state.
  state:turn(State, Player),
  % Get the enemy of the turn from the state.
  state:enemy(Player, Turn).
```

src/movement.pl

```
:- module(movement, []).

:- use_module(board).
:- use_module('movement/bishop', [moves/4 as bishop_moves]).
:- use_module('movement/king', [moves/4 as king_moves]).
:- use_module('movement/knight', [moves/4 as knight_moves]).
:- use_module('movement/pawn', [moves/4 as pawn_move]).
:- use_module('movement/positions').
:- use_module('movement/queen', [moves/4 as queen_moves]).
:- use_module('movement/rook', [moves/4 as rook_moves]).
:- use_module(state).

%% all_moves(+State: state, +Square: square, +Piece: piece,
   -Moves: list).
%
%  Gets all available moves for the given
%
%  @param From the starting square
%  @param Direction the direction of the move
%  @param To the destination square
```

```prolog
all_moves(State, Moves) :-
  % Find all moves on the board.
  findall(X, board:square(X), AllSquares),
  % Generate all moves for every square.
  maplist(all_moves(State), AllSquares, AllMoves),
  % Flatten the found moves.
  flatten(AllMoves, Moves).

%% all_moves(+State: state, +Square: square, -Moves: list).
%
%  Gets all available moves for the given location.
%
%  @param State the current state
%  @param Square the square to generate moves for
%  @param Moves the moves found
all_moves(State, Square, Moves) :-
  % Get the current player.
  state:turn(State, Turn),
  % Get the piece at the given square, only generate moves for the current
  % player.
  state:piece_at(State, Square, piece(Type, Turn)), !,
  % Get all moves.
  all_moves(State, Square, piece(Type, Turn), Moves).
all_moves(_, _, []).

%% all_moves(+State: state, +Square: square, +Piece: piece, -Moves: list).
%
%  Gets all available moves for the given piece at the given location.
%
%  @param State the current state
%  @param Square the square to generate moves for
%  @param Piece the piece at the given square
%  @param Moves the moves found
all_moves(State, Square, piece(bishop, Turn), Moves) :-
  bishop_moves(State, Square, Turn, Moves), !.
all_moves(State, Square, piece(knight, Turn), Moves) :-
  knight_moves(State, Square, Turn, Moves), !.
all_moves(State, Square, piece(king, Turn), Moves) :-
  king_moves(State, Square, Turn, Moves), !.
all_moves(State, Square, piece(pawn, Turn), Moves) :-
  pawn_move(State, Square, Turn, Moves), !.
all_moves(State, Square, piece(queen, Turn), Moves) :-
  queen_moves(State, Square, Turn, Moves), !.
all_moves(State, Square, piece(rook, Turn), Moves) :-
  rook_moves(State, Square, Turn, Moves), !.
```

```
all_moves(_, _, _, []).

%% attacking(+Board: board, +Piece: piece, +Current: square, -Target: square).
%
%  Gets whether a square can attack another square.
%
%  @param Board the board
%  @param Piece the piece at the given Current square
%  @param Current the starting piece
%  @param Target the target to attack
attacking(Board, piece(bishop, _), Current, Target) :-
  positions:bishop_attacks(Current, Direction, Target),
  path_clear(Board, move(Current, Direction, Target)).
attacking(_, piece(king, _), Current, Target) :-
  positions:king_attacks(Current, Target).
attacking(_, piece(knight, _), Current, Target) :-
  positions:knight_attacks(Current, Target).
attacking(_, piece(pawn, Color), Current, Target) :-
  positions:pawn_attacks(Current, Color, Target).
attacking(Board, piece(queen, _), Current, Target) :-
  positions:queen_attacks(Current, Direction, Target),
  path_clear(Board, move(Current, Direction, Target)).
attacking(Board, piece(rook, _), Current, Target) :-
  positions:rook_attacks(Current, Direction, Target),
  path_clear(Board, move(Current, Direction, Target)).

%% path_clear(+Board: board, +Path: move).
%
%  Verifies that a path contains only empty squares.
%
%  @param Board the board
%  @param Path the path containing the path to follow
path_clear(_, move(Square, _, Square)) :- !.
path_clear(Board, move(From, Direction, To)) :-
  % Get the next square on the path.
  path_next(From, Direction, Next),
  % Verify this square is empty using the subroutine.
  path_clear_sub(Board, move(Next, Direction, To)).

%% path_clear_sub(+Board: board, +Path: move).
%
%  Verifies that a path contains only empty squares, subroutine. This routine is
%  required, because else the starting square that contains the piece would also
%  need to be empty, which is obviously always false.
%
```

```prolog
%  @param Board the board
%  @param Path the path containing the path to follow
path_clear_sub(_, move(Square, _, Square)) :- !.
path_clear_sub(Board, move(From, Direction, To)) :-
  % Verify the square is empty.
  board:free(Board, From),
  % Get the next square on the path.
  path_next(From, Direction, Next),
  % Verify this square is empty using the subroutine.
  path_clear_sub(Board, move(Next, Direction, To)).

%% path_next(+R/C: square, +Direction: direction, -R1/C1:
   square).
%
%  Get the next square on a path in a given direction.
%
%  @param R/C the current square
%  @param Direction the direction of the path
%  @param R1/C1 the next square
path_next(R/C, down, R1/C) :- R1 is R - 1.
path_next(R/C, down_left, R1/C1) :- R1 is R - 1, C1 is C - 1.
path_next(R/C, down_right, R1/C1) :- R1 is R - 1, C1 is C +
   1.
path_next(R/C, left, R/C1) :- C1 is C - 1.
path_next(R/C, right, R/C1) :- C1 is C + 1.
path_next(R/C, up, R1/C) :- R1 is R + 1.
path_next(R/C, up_left, R1/C1) :- R1 is R + 1, C1 is C - 1.
path_next(R/C, up_right, R1/C1) :- R1 is R + 1, C1 is C + 1.
```

src/state.pl

```prolog
:- module(state, []).

:- use_module(board).
:- use_module(movement).

%% apply_move(+Before: state, +Move: move, -After: state).
%
%  Applies the given castling move to the state.
%
%  @param Before the current state
%  @param Move the move to apply
%  @param After the resulting state, after applying the move
apply_move(Before, move(castling, Castling), After) :-
  % Extract the board from the state.
  board(Before, BeforeBoard),

```

```
  %%%% BOARD
  % Update the castling squares.
  board:castling_squares(Castling, KingFrom, RookFrom,
   KingTo, RookTo),
  % Castle the king.
  board:move_piece(BeforeBoard, KingFrom, KingTo,
   AfterBoardKing),
  % Castle the rook.
  board:move_piece(AfterBoardKing, RookFrom, RookTo,
   AfterBoard),
  % Save the board.
  update_board(Before, AfterBoard, BoardState),

  %%%% TURN
  update_turn(BoardState, TurnState),

  %%%% CASTLING RIGHTS
  update_castling(TurnState, CastlingState),

  %%%% EN PASSANT
  reset_enpassant(CastlingState, EnPassantState),

  %%%% HALF-MOVE COUNTER
  inc_halfcount(EnPassantState, HCState),

  %%%% FULL-MOVE COUNTER
  inc_fullcount(HCState, After).

%% apply_move(+Before: state, +Move: move, -After: state).
%
%  Applies the given capture move to the state.
%
%  @param Before the current state
%  @param Move the move to apply
%  @param After the resulting state, after applying the move
apply_move(Before, move(capture, From, To), After) :-
  % Extract the board from the state.
  board(Before, BeforeBoard),

  %%%% BOARD
  % Move the piece on the board.
  board:move_piece(BeforeBoard, From, To, AfterBoard),
  % Save the board.
  update_board(Before, AfterBoard, BoardState),

  %%%% TURN
  update_turn(BoardState, TurnState),

```

```
  %%%% CASTLING RIGHTS
  update_castling(TurnState, CastlingState),

  %%%% EN PASSANT
  reset_enpassant(CastlingState, EnPassantState),

  %%%% HALF-MOVE COUNTER
  reset_halfcount(EnPassantState, HCState),

  %%%% FULL-MOVE COUNTER
  inc_fullcount(HCState, After).

%% apply_move(+Before: state, +Move: move, -After: state).
%
%  Applies the given en-passant move to the state.
%
%  @param Before the current state
%  @param Move the move to apply
%  @param After the resulting state, after applying the move
apply_move(Before, move(en_passant, From, EPCapture, To),
    After) :-
  % Extract the board from the state.
  board(Before, BeforeBoard),

  %%%% BOARD
  % Move the piece.
  board:move_piece(BeforeBoard, From, To, AfterMove),
  % Clear het en passant square.
  board:clear(AfterMove, EPCapture, AfterBoard),
  % Save the board.
  update_board(Before, AfterBoard, BoardState),

  %%%% TURN
  update_turn(BoardState, TurnState),

  %%%% CASTLING RIGHTS
  update_castling(TurnState, CastlingState),

  %%%% EN PASSANT
  reset_enpassant(CastlingState, EnPassantState),

  %%%% HALF-MOVE COUNTER
  reset_halfcount(EnPassantState, HCState),

  %%%% FULL-MOVE COUNTER
  inc_fullcount(HCState, After).

%% apply_move(+Before: state, +Move: move, -After: state).
```

```
%
%  Applies the given move to the state.
%
%  @param Before the current state
%  @param Move the move to apply
%  @param After the resulting state, after applying the move
apply_move(Before, move(move, From, EPSquare, To), After) :-
  % Extract the board from the state.
  board(Before, BeforeBoard),

  %%%% BOARD
  % Move the piece.
  board:move_piece(BeforeBoard, From, To, AfterBoard),
  % Save the board.
  update_board(Before, AfterBoard, BoardState),

  %%%% TURN
  update_turn(BoardState, TurnState),

  %%%% CASTLING RIGHTS
  update_castling(TurnState, CastlingState),

  %%%% EN PASSANT
  update_enpassant(CastlingState, EPSquare, EnPassantState),

  %%%% HALF-MOVE COUNTER
  reset_halfcount(EnPassantState, HCState),

  %%%% FULL-MOVE COUNTER
  inc_fullcount(HCState, After).
apply_move(Before, move(move, From, To), After) :-
  % Extract the board from the state.
  board(Before, BeforeBoard),

  %%%% BOARD
  % Move the piece.
  board:move_piece(BeforeBoard, From, To, AfterBoard),
  % Save the board.
  update_board(Before, AfterBoard, BoardState),

  %%%% TURN
  update_turn(BoardState, TurnState),

  %%%% CASTLING RIGHTS
  update_castling(TurnState, CastlingState),

  %%%% EN PASSANT
  reset_enpassant(CastlingState, EnPassantState),
```

```

  %%%% HALF-MOVE COUNTER
  % Get a piece that may be captured en-passant.
  board:piece_at(BeforeBoard, From, MovedPiece),
  inc_halfcount(EnPassantState, MovedPiece, HCState),

  %%%% FULL-MOVE COUNTER
  inc_fullcount(HCState, After).

%% apply_move(+Before: state, +Move: move, -After: state).
%
%  Applies the given promotion move to the state.
%
%  @param Before the current state
%  @param Move the move to apply
%  @param After the resulting state, after applying the move
apply_move(Before, move(promotion, Piece, From, To), After)
    :-
  % Extract the board from the state.
  board(Before, BeforeBoard),
  % Extract the turn from the state.
  turn(Before, Turn),

  %%%% BOARD
  % Clear the current square.
  board:clear(BeforeBoard, From, ClearedBoard),
  % Set the piece on the square.
  board:set_piece(ClearedBoard, To, piece(Piece, Turn),
     AfterBoard),
  % Save the board.
  update_board(Before, AfterBoard, BoardState),

  %%%% TURN
  update_turn(BoardState, TurnState),

  %%%% CASTLING RIGHTS
  update_castling(TurnState, CastlingState),

  %%%% EN PASSANT
  reset_enpassant(CastlingState, EnPassantState),

  %%%% HALF-MOVE COUNTER
  reset_halfcount(EnPassantState, HCState),

  %%%% FULL-MOVE COUNTER
  inc_fullcount(HCState, After).

```

```prolog
%% attacking(+Board: board, +Player: player, +Square: square).
%
%  Validates whether a square is under attack by the given player.
%
%  @param Board the board
%  @param Player the attacker
%  @param Square the square that must be checked
attacking(Board, Player, Square) :-
  % Get a piece of the current player.
  board:piece_at(Board, Position, piece(Type, Player)),
  % Validate if this piece is attacking the square.
  movement:attacking(Board, piece(Type, Player), Position, Square), !.

%% attacking_squares(+Board: board, +Player: player, +Attacked: list).
%
%  Get all squares that are under attack by the given player.
%
%  @param Board the board
%  @param Player the attacker
%  @param Attacked the squares that are under attack
attacking_squares(Board, Player, Attacked) :-
  % Find all squares on the board.
  findall(X, board:square(X), AllSquares),
  % Filter out the squares that are not under attack.
  include(attacking(Board, Player), AllSquares, Attacked).

%% board(+State: state, -Board: board).
%
%  Extract the board from the state.
%
%  @param State the state
%  @param Board the board
board(state(Board, _, _, _, _, _), Board).

%% can_castle(+Board: board, +Type: castling).
%
%  Validates a given castling right.
%
%  @param Board the board
%  @param Type the castling right to validate
can_castle(Board, castling(Type, Color)) :-
  % Get the king and rook squares.
  board:castling_squares(castling(Type, Color), KingFrom, RookFrom, _, _),
```

```
  % Validate the king is at the correct position.
  board:piece_at(Board, KingFrom, piece(king, Color)),
  % Validate the rook is at the correct position.
  board:piece_at(Board, RookFrom, piece(rook, Color)).

%% castling(+State: state, -C: list).
%
%  Extract the castling rights from the state.
%
%  @param State the state
%  @param C the castling rights
castling(state(_, _, C, _, _, _), C).

%% check(+State: state, +Player: turn).
%
%  Validates the given player is in check.
%
%  @param State the state
%  @param Player the player
check(State, Player) :-
  % Extract the board from the state.
  board(State, Board),
  % Validate the in-check status.
  board:check(Board, Player).

% Enemies of the given players.
enemy(black, white).
enemy(white, black).

%% en_passant(+State: state, -EP: square).
%
%  Extract the en-passant square from the state.
%
%  @param State the state
%  @param EP the en-passant square
en_passant(state(_, _, _, EP, _, _), EP).

%% full_count(+State: state, -FC: integer).
%
%  Extract the full-move counter from the state.
%
%  @param State the state
%  @param FC the full move counter
full_count(state(_, _, _, _, _, FC), FC).

%% inc_fullcount(+State1: state, -State2: state).
%
%  Increments the full-move counter
```

```
%
%  @param State1 the initial state
%  @param State2 the updated state
inc_fullcount(state(B, white, C, EP, HC, FC), state(B,
    white, C, EP, HC, FC1)) :-
  % The counter should be incremented since black has moved.
  succ(FC, FC1).
inc_fullcount(state(B, black, C, EP, HC, FC), state(B,
    black, C, EP, HC, FC)).

%% half_count(+State: state, -HC: integer).
%
%  Extract the half-move counter from the state.
%
%  @param State the state
%  @param HC the half move counter
half_count(state(_, _, _, _, HC, _), HC).

%% inc_halfcount(+State1: state, -State2: state).
%
%  Increments the half-move counter
%
%  @param State1 the initial state
%  @param State2 the updated state
inc_halfcount(state(B, T, C, EP, HC, FC), state(B, T, C, EP,
    HC1, FC)) :-
  % Increment the half-move counter.
  succ(HC, HC1).

%% inc_fullcount(+State1: state, +Piece: piece, -State2:
    state).
%
%  Increments the half-move counter based on the piece.
%
%  @param State1 the initial state
%  @param Piece the piece
%  @param State2 the updated state
inc_halfcount(state(B, T, C, EP, _, FC), piece(pawn, _),
    state(B, T, C, EP, 0, FC)) :- !.
inc_halfcount(state(B, T, C, EP, HC, FC), _, state(B, T, C,
    EP, HC1, FC)) :-
  % Draw condition.
  succ(HC, HC1), HC1 < 75.

%% piece_at(+State: state, +Square: square, -Piece: piece).
%
%  Get the piece at the given square.
%
```

```
%  @param State the state
%  @param Square the square
%  @param Piece the piece at the given square
piece_at(State, Square, Piece) :-
  % Extract the board from the state.
  board(State, Board),
  % Get the piece at the square.
  board:piece_at(Board, Square, Piece).

%% reset_enpassant(+State1: state, -State2: state).
%
%  Resets the en-passant square.
%
%  @param State1 the initial state
%  @param State2 the updated state
reset_enpassant(state(B, T, C, _, HC, FC), state(B, T, C,
    none, HC, FC)).

%% reset_halfcount(+State1: state, -State2: state).
%
%  Resets the half-move counter.
%
%  @param State1 the initial state
%  @param State2 the updated state
reset_halfcount(state(B, T, C, EP, _, FC), state(B, T, C,
    EP, 0, FC)).

%% turn(+State: state, -Turn: turn).
%
%  Extract the turn from the state.
%
%  @param State the state
%  @param Turn the turn
turn(state(_, Turn, _, _, _, _), Turn).

%% update_board(+State1: state, +Board: board, -State2:
    state).
%
%  Updates the board in the state.
%
%  @param State1 the initial state
%  @param Board the board to replace
%  @param State2 the updated state
update_board(state(_, T, C, EP, HC, FC), Board, state(Board,
    T, C, EP, HC, FC)).

%% update_castling(+State1: state, -State2: state).
%
```

```
%  Updates the castling rights in the state.
%
%  @param State1 the initial state
%  @param State2 the updated state
update_castling(state(B, T, C, EP, HC, FC), state(B, T, C1,
    EP, HC, FC)) :-
  % Include all valid rights.
  include(can_castle(B), C, C1).

%% update_enpassant(+State1: state, +EP: square, -State2:
    state).
%
%  Updates the en-passant square in the state.
%
%  @param State1 the initial state
%  @param EP the en-passant square
%  @param State2 the updated state
update_enpassant(state(B, T, C, _, HC, FC), EP, state(B, T,
    C, EP, HC, FC)).

%% update_turn(+State1: state, +Turn: turn, -State2: state).
%
%  Updates the turn in the state.
%
%  @param State1 the initial state
%  @param Turn the turn to replace
%  @param State2 the updated state
update_turn(state(B, _, C, EP, HC, FC), Turn, state(B, Turn,
    C, EP, HC, FC)).
update_turn(state(B, white, C, EP, HC, FC), state(B, black,
    C, EP, HC, FC)).
update_turn(state(B, black, C, EP, HC, FC), state(B, white,
    C, EP, HC, FC)).
```

src/movement/bishop.pl

```
:- module(bishop, []).

:- use_module('../board').
:- use_module('../state').
:- use_module(positions).

%% move(+Board: board, +Turn: turn, +move:move: -move:move).
%
%  Formulates a capture move.
%
%  @param Board the current board
```

```
% @param Turn the owner of the bishop
% @param move the unprocessed move
% @param move(capture) the capturing move
move(Board, Turn, move(From, _, To), move(capture, From,
    To)) :-
  % Verify the destination square contains an enemy piece.
  board:enemy(Board, To, Turn), !.

%% move(+Board: board, +Turn: turn, +move:move: -move:move).
%
% Formulates a regular walking move.
%
% @param Board the current board
% @param Turn the owner of the bishop
% @param move the unprocessed move
% @param move(move) the walking move
move(Board, _, move(From, _, To), move(move, From, To)) :-
  % Verify the destination square does not contain any piece.
  board:free(Board, To).

%% moves(+State: state, +Square: square, +Turn: turn,
    -Moves: list).
%
% Finds all valid moves for a bishop on the given Square.
%
% @param State the current game state
% @param Square the square that contains a bishop
% @param Turn the owner of the bishop
% @param Moves the resulting available moves
moves(State, Square, Turn, Moves) :-
  % Extract the board from the state.
  state:board(State, Board),
  % Get all bishop moves from the current square.
  positions:bishop_moves(Square, BishopMoves),
  % Validate every found bishop move.
  include(movement:path_clear(Board), BishopMoves,
    FilteredMoves),
  % Convert the moves to either a move or a capture.
  convlist(move(Board, Turn), FilteredMoves, Moves).
```

src/movement/king.pl

```
:- module(king, []).

:- use_module('../board').
:- use_module('../movement').
:- use_module('../state').
```

```

%% castle(+State: state, +Square:, +Turn: turn, +move:move).
%
%  Formulates a castling move (kingside).
%
%  @param State the current game state
%  @param Square the current king square
%  @param Turn the owner of the king
%  @param move(castling) the castling move
castle(State, Square, Turn, move(castling,
    castling(kingside, Turn))) :-
  % Verify the king is on a valid castling square.
  board:castling_squares(castling(kingside, Turn), Square,
    RookSquare, _, _),
  % Extract the board from the state.
  state:board(State, Board),
  % Extract the castling possibilities from the state.
  state:castling(State, Castlings),
  % Verify this castling type may be executed.
  member(castling(kingside, Turn), Castlings),
  % Verify there is no piece between the king and the rook.
  movement:path_clear(Board, move(Square, right,
    RookSquare)).

%% castle(+State: state, +Square:, +Turn: turn, +move:move).
%
%  Formulates a castling move (queenside).
%
%  @param State the current game state
%  @param Square the current king square
%  @param Turn the owner of the king
%  @param move(castling) the castling move
castle(State, Square, Turn, move(castling,
    castling(queenside, Turn))) :-
  % Verify the king is on a valid castling square.
  board:castling_squares(castling(queenside, Turn), Square,
    RookSquare, _, _),
  % Extract the board from the state.
  state:board(State, Board),
  % Extract the castling possibilities from the state.
  state:castling(State, Castlings),
  % Verify this castling type may be executed.
  member(castling(queenside, Turn), Castlings),
  % Verify there is no piece between the king and the rook.
  movement:path_clear(Board, move(Square, left, RookSquare)).

%% move(+Board: board, +Turn: turn, +move:move: -move:move).
%
```

```
%  Formulates a capture move.
%
%  @param Board the current board
%  @param Turn the owner of the king
%  @param move the unprocessed move
%  @param move(capture) the capturing move
move(Board, Turn, move(From, To), move(capture, From, To)) :-
  % Verify the destination square contains an enemy piece.
  board:enemy(Board, To, Turn), !.

%% move(+Board: board, +Turn: turn, +move:move: -move:move).
%
%  Formulates a regular walking move.
%
%  @param Board the current board
%  @param Turn the owner of the king
%  @param move the unprocessed move
%  @param move(move) the walking move
move(Board, _, move(From, To), move(move, From, To)) :-
  % Verify the destination square does not contain any piece.
  board:free(Board, To).

%% moves(+State: state, +Square: square, +Turn: turn, -list).
%
%  Finds all valid moves for a knight on the given Square.
%
%  @param State the current game state
%  @param Square the square that contains a king
%  @param Turn the owner of the king
%  @param Moves the resulting available moves
moves(State, Square, Turn, [FilteredMoves, CastlingMoves]) :-
  % Extract the board from the state.
  state:board(State, Board),

  % Get all king moves from the current square.
  positions:king_moves(Square, KingMoves),
  % Convert the moves to either a move or a capture.
  convlist(move(Board, Turn), KingMoves, FilteredMoves),

  % Find all castling moves from the current square and game
   state.
  findall(C, castle(State, Square, Turn, C), CastlingMoves).
```

src/movement/knight.pl

```
:- module(knight, []).

```

```prolog
:- use_module('../board').
:- use_module('../state').
:- use_module(positions).

%% move(+Board: board, +Turn: turn, +move:move: -move:move).
%
%  Formulates a capture move.
%
%  @param Board the current board
%  @param Turn the owner of the knight
%  @param move the unprocessed move
%  @param move(capture) the capturing move
move(Board, Turn, move(From, To), move(capture, From, To)) :-
  % Verify the destination square contains an enemy piece.
  board:enemy(Board, To, Turn), !.

%% move(+Board: board, +Turn: turn, +move:move: -move:move).
%
%  Formulates a regular walking move.
%
%  @param Board the current board
%  @param Turn the owner of the knight
%  @param move the unprocessed move
%  @param move(move) the walking move
move(Board, _, move(From, To), move(move, From, To)) :-
  % Verify the destination square does not contain any piece.
  board:free(Board, To).

%% moves(+State: state, +Square: square, +Turn: turn,
    -Moves: list).
%
%  Finds all valid moves for a knight on the given Square.
%
%  @param State the current game state
%  @param Square the square that contains a knight
%  @param Turn the owner of the knight
%  @param Moves the resulting available moves
moves(State, Square, Turn, Moves) :-
  % Extract the board from the state.
  state:board(State, Board),
  % Get all knight moves from the current square.
  positions:knight_moves(Square, KnightMoves),
  % Convert the moves to either a move or a capture.
  convlist(move(Board, Turn), KnightMoves, Moves).
```

src/movement/pawn.pl

```
:- module(pawn, []).

:- use_module('../board').
:- use_module('../movement').
:- use_module('../state').

%% move(+State: state, +Turn: turn, +move:move: -move:move).
%
%  Formulates a capture move.
%
%  @param State the state
%  @param Turn the owner of the pawn
%  @param move the unprocessed move
%  @param move(capture) the capturing move
capture(State, Turn, move(From, To), move(capture, From,
    To)) :-
  % Extract the board from the state.
  state:board(State, Board),
  % Verify the destination is not a promotion square.
  \+ promotion_square(To),
  % Verify the destination square contains an enemy piece.
  board:enemy(Board, To, Turn).

%% move(+State: state, +Turn: turn, +move: move,
    -PromotionMoves: list).
%
%  Formulates a promotion-capture move.
%
%  @param State the state
%  @param Turn the owner of the pawn
%  @param move the unprocessed move
%  @param PromotionMoves the promotion moves
capture(State, Turn, move(From, To), PromotionMoves) :-
  % Extract the board from the state.
  state:board(State, Board),
  % Verify the destination is a promotion square.
  promotion_square(To),
  % Verify the destination square contains an enemy piece.
  board:enemy(Board, To, Turn),
  % Get all possible promotions.
  bagof(Move, promotion_move(Move, From, To),
    PromotionMoves).

%% move(+State: state, +Turn: turn, +move: move, -move:
    move).
%
```

```prolog
%  Formulates an en-pasant capture move.
%
%  @param S the state
%  @param Turn the owner of the pawn
%  @param move the unprocessed move
%  @param move(en_passant) the capturing move
capture(S, Turn, move(SR/SC, DR/DC), move(en_passant, SR/SC,
    SR/DC, DR/DC)) :-
  % Extract the board from the state.
  state:board(S, Board),
  % Extract the en-passant square from the state.
  state:en_passant(S, DR/DC),
  % Verify the en-passant square contains an enemy pawn.
  board:enemy(Board, SR/DC, Turn),
  % Verify the destination square does not contain any piece.
  board:free(Board, DR/DC).

%% move(+Board: board, +Turn: turn, +move: move, -move:
    move).
%
%  Formulates a pawn double walking move, for initial
    positions.
%
%  @param Board the board
%  @param Turn the owner of the pawn
%  @param move the unprocessed move
%  @param move(move) the walking move
move(Board, _, move(From, EP, To), move(move, From, EP, To))
    :-
  % Verify the skipped square does not contain any piece.
  board:free(Board, EP),
  % Verify the destination square does not contain any piece.
  board:free(Board, To).

%% move(+Board: board, +Turn: turn, +move: move,
    PromotionMoves: list).
%
%  Formulates a promotion move.
%
%  @param Board the board
%  @param Turn the owner of the pawn
%  @param move the unprocessed move
%  @param PromotionMoves the promotion moves
move(Board, _, move(From, To), PromotionMoves) :-
  % Verify the destination is a promotion square.
  promotion_square(To),
  % Verify the destination square does not contain any piece.
  board:free(Board, To),
```

```
  % Get all possible promotions.
  bagof(Move, promotion_move(Move, From, To), PromotionMoves).

%% move(+Board: board, +Turn: turn, +move: move, -move: move).
%
%  Formulates a pawn double walking move, for initial positions.
%
%  @param Board the board
%  @param Turn the owner of the pawn
%  @param move the unprocessed move
%  @param move(move) the walking move
move(Board, _, move(From, To), move(move, From, To)) :-
  % Verify the destination is not a promotion square.
  \+ promotion_square(To),
  % Verify the destination square does not contain any piece.
  board:free(Board, To).

%% moves(+State: state, +Square: square, +Turn: turn, -list).
%
%  Finds all valid moves for a pawn on the given Square.
%
%  @param State the current game state
%  @param Square the square that contains a pawn
%  @param Turn the owner of the pawn
%  @param Moves the resulting available moves
moves(State, Square, Turn, [Moves, Captures, EnPassants]) :-
  % Extract the board from the state.
  state:board(State, Board),

  % Get the pawn move from the current square.
  positions:pawn(Square, Turn, PawnMove),
  % Convert the move to either a move or a promotion.
  convlist(move(Board, Turn), [PawnMove], Moves),

  % Find all capture moves.
  findall(X, positions:pawn_capture(Square, Turn, X), PawnCaptures),
  % Convert the moves to either a capture or a promotion-capture.
  convlist(capture(State, Turn), PawnCaptures, Captures),

  % Find all en-passant moves.
  findall(X, positions:pawn_enpassant(Square, Turn, X), EnPassantMove),
  % Convert the moves to either a walk or a capture.
```

```
  convlist(move(Board, Turn), EnPassantMove, EnPassants).

% Valid promotions.
promotion_move(move(promotion, bishop, From, To), From, To).
promotion_move(move(promotion, knight, From, To), From, To).
promotion_move(move(promotion, queen, From, To), From, To).
promotion_move(move(promotion, rook, From, To), From, To).

% Squares that allow pawn promotion.
promotion_square(1/_).
promotion_square(8/_).
```

src/movement/positions.pl

```
:- module(positions, []).

:- dynamic saved_bishop_moves/2.
:- dynamic saved_king_moves/2.
:- dynamic saved_knight_moves/2.
:- dynamic saved_rook_moves/2.

%% bishop_attacks(+From: square, +Direction: direction, +To:
   square).
%
%  Verifies whether or not a bishop can attack a square,
   given a starting
%  square.
%
%  @param From the starting square
%  @param Direction the direction of the move
%  @param To the destination square
bishop_attacks(From, Direction, To) :- bishop(From,
   move(From, Direction, To)).

%% bishop_moves(+R/C: square, -Moves: list).
%
%  Gets all bishop moves from the given starting square.
%
%  @param R/C the starting square
%  @param Moves the moves
bishop_moves(R/C, Moves) :-
  % Fetch the moves from the cache.
  saved_bishop_moves(R/C, Moves), !.
bishop_moves(R/C, Moves) :-
  % Generate all moves.
  setof(X, bishop(R/C, X), Moves),
  % Store the moves in the cache for future use.
```

```
  assertz(saved_bishop_moves(R/C, Moves)).

%% bishop(+R/C: square, -move: move).
%
%  Generates a bishop move.
%
%  @param R/C the starting square
%  @param move the move
bishop(R/C, move(R/C, down_left, R1/C1)) :-
  % Diagonal offset.
  between(1, 8, I),
  % Diagonal movement.
  R1 is R - I, C1 is C - I,
  % Bounds checking.
  between(1, R, R1), between(1, C, C1).
bishop(R/C, move(R/C, down_right, R1/C1)) :-
  % Diagonal offset.
  between(1, 8, I),
  % Diagonal movement.
  R1 is R - I, C1 is C + I,
  % Bounds checking.
  between(1, R, R1), between(C, 8, C1).
bishop(R/C, move(R/C, up_left, R1/C1)) :-
  % Diagonal offset.
  between(1, 8, I),
  % Diagonal movement.
  R1 is R + I, C1 is C - I,
  % Bounds checking.
  between(R, 8, R1), between(1, C, C1).
bishop(R/C, move(R/C, up_right, R1/C1)) :-
  % Diagonal offset.
  between(1, 8, I),
  % Diagonal movement.
  R1 is R + I, C1 is C + I,
  % Bounds checking.
  between(R, 8, R1), between(C, 8, C1).

%% king_attacks(+From: square, +Direction: direction, +To:
   square).
%
%  Verifies whether or not a king can attack a square, given
   a starting square.
%
%  @param From the starting square
%  @param Direction the direction of the move
%  @param To the destination square
king_attacks(From, To) :- king(From, move(From, To)).

```

```
%% king_moves(+R/C: square, -Moves: list).
%
%  Gets all king moves from the given starting square.
%
%  @param R/C the starting square
%  @param Moves the moves
king_moves(R/C, Moves) :-
  % Fetch the moves from the cache.
  saved_king_moves(R/C, Moves), !.
king_moves(R/C, Moves) :-
  % Generate all moves.
  setof(X, king(R/C, X), Moves),
  % Store the moves in the cache.
  assertz(saved_king_moves(R/C, Moves)).

%% king(+R/C: square, -move: move).
%
%  Generates a king move.
%
%  @param R/C the starting square
%  @param move the move
king(R/C, move(R/C, R1/C1)) :- R < 8, C < 8, R1 is R + 1, C1
    is C + 1.
king(R/C, move(R/C, R1/C)) :- R < 8, R1 is R + 1.
king(R/C, move(R/C, R1/C1)) :- R < 8, C > 1, R1 is R + 1, C1
    is C - 1.
king(R/C, move(R/C, R/C1)) :- C < 8, C1 is C + 1.
king(R/C, move(R/C, R/C1)) :- C > 1, C1 is C - 1.
king(R/C, move(R/C, R1/C1)) :- R > 1, C < 8, R1 is R - 1, C1
    is C + 1.
king(R/C, move(R/C, R1/C)) :- R > 1, R1 is R - 1.
king(R/C, move(R/C, R1/C1)) :- R > 1, C > 1, R1 is R - 1, C1
    is C - 1.

%% knight_attacks(+From: square, +Direction: direction, +To:
    square).
%
%  Verifies whether or not a knight can attack a square,
    given a starting
%  square.
%
%  @param From the starting square
%  @param Direction the direction of the move
%  @param To the destination square
knight_attacks(From, To) :- knight(From, move(From, To)).

%% knight_moves(+R/C: square, -Moves: list).
%
```

```
%  Gets all knight moves from the given starting square.
%
%  @param R/C the starting square
%  @param Moves the moves
knight_moves(R/C, Moves) :-
  % Fetch the moves from the cache.
  saved_knight_moves(R/C, Moves), !.
knight_moves(R/C, Moves) :-
  % Generate all moves.
  setof(X, knight(R/C, X), Moves),
  % Store the moves in the cache for future use.
  assertz(saved_knight_moves(R/C, Moves)).

%% knight(+R/C: square, -move: move).
%
%  Generates a knight move.
%
%  @param R/C the starting square
%  @param move the move
knight(R/C, move(R/C, R1/C1)) :- R < 7, C < 8, R1 is R + 2,
   C1 is C + 1.
knight(R/C, move(R/C, R1/C1)) :- R < 7, C > 1, R1 is R + 2,
   C1 is C - 1.
knight(R/C, move(R/C, R1/C1)) :- R < 8, C < 7, R1 is R + 1,
   C1 is C + 2.
knight(R/C, move(R/C, R1/C1)) :- R < 8, C > 2, R1 is R + 1,
   C1 is C - 2.
knight(R/C, move(R/C, R1/C1)) :- R > 1, C < 7, R1 is R - 1,
   C1 is C + 2.
knight(R/C, move(R/C, R1/C1)) :- R > 1, C > 2, R1 is R - 1,
   C1 is C - 2.
knight(R/C, move(R/C, R1/C1)) :- R > 2, C < 8, R1 is R - 2,
   C1 is C + 1.
knight(R/C, move(R/C, R1/C1)) :- R > 2, C > 1, R1 is R - 2,
   C1 is C - 1.

%% pawn_attacks(+From: square, +Direction: direction, +To:
   square).
%
%  Verifies whether or not a pawn can attack a square, given
   a starting square.
%
%  @param From the starting square
%  @param Direction the direction of the move
%  @param To the destination square
pawn_attacks(R/C, Turn, R1/C1) :- pawn_capture(R/C, Turn,
   move(R/C, R1/C1)).

```

```
%% pawn_capture(+R/C: square, +Turn: turn, -move: move).
%
%  Generates a pawn move.
%
%  @param R/C the starting square
%  @param Turn the turn
%  @param move the move
pawn_capture(R/C, black, move(R/C, R1/C1)) :-
  R > 1, C < 8, R1 is R - 1, C1 is C + 1.
pawn_capture(R/C, black, move(R/C, R1/C1)) :-
  R > 1, C > 1, R1 is R - 1, C1 is C - 1.
pawn_capture(R/C, white, move(R/C, R1/C1)) :-
  R < 8, C < 8, R1 is R + 1, C1 is C + 1.
pawn_capture(R/C, white, move(R/C, R1/C1)) :-
  R < 8, C > 1, R1 is R + 1, C1 is C - 1.

%% pawn_enpassant(+R/C: square, +Turn: turn, -move: move).
%
%  Generates a pawn en-passant move.
%
%  @param R/C the initial square
%  @param Turn the turn
%  @param move the move
pawn_enpassant(7/C, black, move(7/C, 6/C, 5/C)) :-
    between(1, 8, C).
pawn_enpassant(2/C, white, move(2/C, 3/C, 4/C)) :-
    between(1, 8 ,C).

%% pawn(+R/C: square, +Turn: turn, -move: move).
%
%  Generates a pawn move.
%
%  @param R/C the starting square
%  @param Turn the turn
%  @param move the move
pawn(R/C, black, move(R/C, R1/C)) :- R > 1, R1 is R - 1.
pawn(R/C, white, move(R/C, R1/C)) :- R < 8, R1 is R + 1.

%% queen_attacks(+From: square, +Direction: direction, +To:
    square).
%
%  Verifies whether or not a queen can attack a square,
    given a starting square.
%
%  @param From the starting square
%  @param Direction the direction of the move
%  @param To the destination square
```

```
queen_attacks(From, Direction, To) :- bishop(From, move(From, Direction, To)), !.
queen_attacks(From, Direction, To) :- rook(From, move(From, Direction, To)).

%% queen_moves(+R/C: square, -Moves: list).
%
%  Gets all queen moves from the given starting square.
%
%  @param R/C the starting square
%  @param Moves the moves
queen_moves(R/C, Moves) :-
  % Fetch the bishop from the cache.
  saved_bishop_moves(R/C, BishopMoves),
  % Fetch the rook from the cache.
  saved_rook_moves(R/C, RookMoves),
  % Merge both move sets.
  append(BishopMoves, RookMoves, Moves), !.
queen_moves(R/C, Moves) :-
  % Generate all bishop moves.
  setof(X, bishop(R/C, X), BishopMoves),
  % Generate all rook moves.
  setof(X, rook(R/C, X), RookMoves),
  % Store the bishop moves in the cache for future use.
  assertz(saved_bishop_moves(R/C, BishopMoves)),
  % Store the rook moves in the cache for future use.
  assertz(saved_rook_moves(R/C, RookMoves)),
  % Merge both move sets.
  append(BishopMoves, RookMoves, Moves).

%% rook_attacks(+From: square, +Direction: direction, +To: square).
%
%  Verifies whether or not a rook can attack a square, given a starting square.
%
%  @param From the starting square
%  @param Direction the direction of the move
%  @param To the destination square
rook_attacks(From, Direction, To) :- rook(From, move(From, Direction, To)).

%% rook_moves(+R/C: square, -Moves: list).
%
%  Gets all rook moves from the given starting square.
%
%  @param R/C the starting square
%  @param Moves the moves
```

```
rook_moves(R/C, Moves) :-
  % Fetch the moves from the cache.
  saved_rook_moves(R/C, Moves), !.
rook_moves(R/C, Moves) :-
  % Generate all moves.
  setof(X, rook(R/C, X), Moves),
  % Store the moves in the cache for future use.
  assertz(saved_rook_moves(R/C, Moves)).

%% rook(+R/C: square, -move: move).
%
%  Generates a rook move.
%
%  @param R/C the starting square
%  @param move the move
rook(R/C, move(R/C, down, R1/C)) :- B is R - 1, between(1,
   B, R1).
rook(R/C, move(R/C, left, R/C1)) :- B is C - 1, between(1,
   B, C1).
rook(R/C, move(R/C, right, R/C1)) :- B is C + 1, between(B,
   8, C1).
rook(R/C, move(R/C, up, R1/C)) :- B is R + 1, between(B, 8,
   R1).
```

src/movement/queen.pl

```
:- module(queen, []).

:- use_module('../board').
:- use_module('../state').
:- use_module(positions).

%% move(+Board: board, +Turn: turn, +move:move: -move:move).
%
%  Formulates a capture move.
%
%  @param Board the current board
%  @param Turn the owner of the queen
%  @param move the unprocessed move
%  @param move(capture) the capturing move
move(Board, Turn, move(From, _, To), move(capture, From,
   To)) :-
  % Verify the destination square contains an enemy piece.
  board:enemy(Board, To, Turn), !.

%% move(+Board: board, +Turn: turn, +move:move: -move:move).
%
```

```
%  Formulates a regular walking move.
%
%  @param Board the current board
%  @param Turn the owner of the queen
%  @param move the unprocessed move
%  @param move(move) the walking move
move(Board, _, move(From, _, To), move(move, From, To)) :-
  % Verify the destination square does not contain any piece.
  board:free(Board, To).

%% moves(+State: state, +Square: square, +Turn: turn,
   -Moves: list).
%
%  Finds all valid moves for a queen on the given Square.
%
%  @param State the current game state
%  @param Square the square that contains a queen
%  @param Turn the owner of the queen
%  @param Moves the resulting available moves
moves(State, Square, Turn, Moves) :-
  % Extract the board from the state.
  state:board(State, Board),
  % Get all queen moves from the current square.
  positions:queen_moves(Square, QueenMoves),
  % Validate every found queen move.
  include(movement:path_clear(Board), QueenMoves,
   FilteredMoves),
  % Convert the moves to either a move or a capture.
  convlist(move(Board, Turn), FilteredMoves, Moves).
```

src/movement/rook.pl

```
:- module(rook, []).

:- use_module('../board').
:- use_module('../state').
:- use_module(positions).

%% move(+Board: board, +Turn: turn, +move:move: -move:move).
%
%  Formulates a capture move.
%
%  @param Board the current board
%  @param Turn the owner of the rook
%  @param move the unprocessed move
%  @param move(capture) the capturing move
```

```
move(Board, Turn, move(From, _, To), move(capture, From, To)) :-
  % Verify the destination square contains an enemy piece.
  board:enemy(Board, To, Turn), !.

%% move(+Board: board, +Turn: turn, +move:move: -move:move).
%
%  Formulates a regular walking move.
%
%  @param Board the current board
%  @param Turn the owner of the rook
%  @param move the unprocessed move
%  @param move(move) the walking move
move(Board, _, move(From, _, To), move(move, From, To)) :-
  % Verify the destination square does not contain any piece.
  board:free(Board, To).

%% moves(+State: state, +Square: square, +Turn: turn, -Moves: list).
%
%  Finds all valid moves for a rook on the given Square.
%
%  @param State the current game state
%  @param Square the square that contains a rook
%  @param Turn the owner of the rook
%  @param Moves the resulting available moves
moves(State, Square, Turn, Moves) :-
  % Extract the board from the state.
  state:board(State, Board),
  % Get all rook moves from the current square.
  positions:rook_moves(Square, RookMoves),
  % Validate every found rook move.
  include(movement:path_clear(Board), RookMoves, FilteredMoves),
  % Convert the moves to either a move or a capture.
  convlist(move(Board, Turn), FilteredMoves, Moves).
```

src/util/draw.pl

Dit bestand bevat ASCII tekens die niet geprint konden worden.

```
:- module(draw, []).

asciiPiece(bishop, black, '\u265D').
asciiPiece(king, black, '\u265A').
asciiPiece(knight, black, '\u265E').
asciiPiece(pawn, black, '\u265F').
```

```
asciiPiece(queen, black, '\u265B').
asciiPiece(rook, black, '\u265C').

asciiPiece(bishop, white, '\u2657').
asciiPiece(king, white, '\u2654').
asciiPiece(knight, white, '\u2658').
asciiPiece(pawn, white, '\u2659').
asciiPiece(queen, white, '\u2655').
asciiPiece(rook, white, '\u2656').

drawBoard(board(R1, R2, R3, R4, R5, R6, R7, R8)) :-
  write('␣␣␣'), drawRow(R8), write('␣␣␣␣8'), nl, nl,
  write('␣␣␣'), drawRow(R7), write('␣␣␣␣7'), nl, nl,
  write('␣␣␣'), drawRow(R6), write('␣␣␣␣6'), nl, nl,
  write('␣␣␣'), drawRow(R5), write('␣␣␣␣5'), nl, nl,
  write('␣␣␣'), drawRow(R4), write('␣␣␣␣4'), nl, nl,
  write('␣␣␣'), drawRow(R3), write('␣␣␣␣3'), nl, nl,
  write('␣␣␣'), drawRow(R2), write('␣␣␣␣2'), nl, nl,
  write('␣␣␣'), drawRow(R1), write('␣␣␣␣1'), nl, nl,
  write('␣␣␣a␣␣␣b␣␣␣c␣␣␣d␣␣␣e␣␣␣f␣␣␣g␣␣␣h'), nl, nl.

drawBoard(state(Board, _, _, _, _, _)) :- drawBoard(Board).

drawLine(0, _) :- !.
drawLine(N, C) :- N1 is N-1, write(C), drawLine(N1, C).

drawPiece(none) :- !, write('\u00B7').
drawPiece(piece(Type,Color)) :- asciiPiece(Type, Color,
    AsciiCode), write(AsciiCode).

drawRow(row(P1, P2, P3, P4, P5, P6, P7, P8)) :-
  drawPiece(P1),
  write('␣␣␣'), drawPiece(P2),
  write('␣␣␣'), drawPiece(P3),
  write('␣␣␣'), drawPiece(P4),
  write('␣␣␣'), drawPiece(P5),
  write('␣␣␣'), drawPiece(P6),
  write('␣␣␣'), drawPiece(P7),
  write('␣␣␣'), drawPiece(P8).
```