

Logisch Programmeren

Project Schaakcomputer

Pieter De Clercq

25 juni 2018

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 taken 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

```
1: function SCORE(situatie)
2:   mijn_score  $\leftarrow$  SCORE_SUB(situatie, huidige_speler)
3:   tegenstander_score  $\leftarrow$  SCORE_SUB(situatie, tegenstander)
4:   return mijn_score – tegenstander_score
5: function SCORE_SUB(situatie, speler)
6:   stukken  $\leftarrow$  alle stukken van speler
7:   scoresi  $\leftarrow$  WAARDE(stukkeni)
8:   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

```
1: function MINIMAX(situatie, diepte)
2:   situaties  $\leftarrow$  VOLGENDE_SPELSITUATIES(situatie)
3:   if diepte == 0 of situaties ==  $\emptyset$  then
4:     return situatie, SCORE(situatie)
5:   beurt  $\leftarrow$  state.beurt
6:   if beurt == mijn_speler then
7:     beste_score  $\leftarrow$   $-\infty$ 
8:     beste_situatie  $\leftarrow$  situatie
9:     for all kandidaat  $\in$  situaties do
10:      st, sc  $\leftarrow$  MINIMAX(kandidaat, diepte – 1)
11:      if sc > beste_score then
12:        beste_score  $\leftarrow$  sc
13:        beste_situatie  $\leftarrow$  st
14:     return beste_situatie, beste_score
15:   else
16:     beste_score  $\leftarrow$   $+\infty$ 
17:     beste_situatie  $\leftarrow$  situatie
18:     for all kandidaat  $\in$  states do
19:      st, sc  $\leftarrow$  MINIMAX(kandidaat, diepte – 1)
20:      if sc < beste_score then
21:        beste_score  $\leftarrow$  sc
22:        beste_situatie  $\leftarrow$  st
23:     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

```
1: function ALPHABETA(situatie, diepte,  $\alpha$ ,  $\beta$ )
2:   situaties  $\leftarrow$  VOLGENDE_SPELSITUATIES(situatie)
3:   if diepte == 0 of situaties ==  $\emptyset$  then
4:     return situatie, SCORE(situatie)
5:   beurt  $\leftarrow$  state.beurt
6:   if beurt == mijn_speler then
7:     beste_score  $\leftarrow$   $-\infty$ 
8:     beste_situatie  $\leftarrow$  situatie
9:     for all kandidaat  $\in$  situaties do
10:      st, sc  $\leftarrow$  ALPHABETA(kandidaat, diepte - 1,  $\alpha$ ,  $\beta$ )
11:      if sc > beste_score then
12:        beste_score  $\leftarrow$  sc
13:        beste_situatie  $\leftarrow$  st
14:         $\alpha \leftarrow$  MAX( $\alpha$ , sc)
15:        if  $\beta \leq \alpha$  then
16:          return beste_situatie, beste_score
17:     return beste_situatie, beste_score
18:   else
19:     beste_score  $\leftarrow$   $+\infty$ 
20:     beste_situatie  $\leftarrow$  situatie
21:     for all kandidaat  $\in$  situaties do
22:      st, sc  $\leftarrow$  ALPHABETA(kandidaat, diepte - 1,  $\alpha$ ,  $\beta$ )
23:      if sc < beste_score then
24:        beste_score  $\leftarrow$  sc
25:        beste_situatie  $\leftarrow$  st
26:         $\beta \leftarrow$  MIN( $\beta$ , sc)
27:        if  $\beta \leq \alpha$  then
28:          return beste_situatie, beste_score
29:   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

```
1: function ALPHABETA(situatie, diepte,  $\alpha$ ,  $\beta$ )
2:   :
3:   if beurt == mijn_speler then
4:     :
5:     for all kandidaat  $\in$  situaties do
6:       st, sc  $\leftarrow$  ALPHABETA(kandidaat, diepte - 1,  $\alpha$ ,  $\beta$ )
7:       pick_equal  $\leftarrow$  RANDOM(0, 1) < 0.5
8:       if sc > beste_score of (sc == beste_score en pick_equal) then
9:         beste_score  $\leftarrow$  sc
10:      :
11:   return beste_situatie, beste_score
12: else
13:   :
14:   for all kandidaat  $\in$  situaties do
15:     st, sc  $\leftarrow$  ALPHABETA(kandidaat, diepte - 1,  $\alpha$ ,  $\beta$ )
16:     pick_equal  $\leftarrow$  RANDOM(0, 1) < 0.5
17:     if sc < beste_score of (sc == beste_score en pick_equal) then
18:       beste_score  $\leftarrow$  sc
19:     :
20:   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

```
1: function ALPHABETA(situatie, diepte,  $\alpha$ ,  $\beta$ )
2:   :
3:   if beurt == mijn_speler then
4:     :
5:     for all kandidaat  $\in$  situaties do
6:       nieuwediepte  $\leftarrow$  DYNAMISCHE_DIEPTE(situaties, diepte)
7:       st, sc  $\leftarrow$  ALPHABETA(kandidaat, nieuwediepte,  $\alpha$ ,  $\beta$ )
8:       pick_equal  $\leftarrow$  RANDOM(0, 1) < 0.5
9:       if sc > beste_score of (sc == beste_score en pick_equal) then
10:        beste_score  $\leftarrow$  sc
11:      :
12:     return beste_situatie, beste_score
13:   else
14:     :
15:     for all kandidaat  $\in$  situaties do
16:       nieuwediepte  $\leftarrow$  DYNAMISCHE_DIEPTE(situaties, diepte)
17:       st, sc  $\leftarrow$  ALPHABETA(kandidaat, nieuwediepte,  $\alpha$ ,  $\beta$ )
18:       pick_equal  $\leftarrow$  RANDOM(0, 1) < 0.5
19:       if sc < beste_score of (sc == beste_score en pick_equal) then
20:        beste_score  $\leftarrow$  sc
21:      :
22:     return beste_situatie, beste_score
23: function DYNAMISCHE_DIEPTE(situaties, diepte)
24:   if situaties.length < 4 then
25:     return diepte + 1
26:   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

```
1 :- module(board, []).
2
3 :- use_module(state).
4
5 % A list of all the possible castling squares in this order:
6 % (castling_type, KingInitial, RookInitial, KingDestination,
7 %   RookDestination).
8 castling_squares(castling(kingside, black), 8/5, 8/8, 8/7,
9   8/6).
10 castling_squares(castling(kingside, white), 1/5, 1/8, 1/7,
11   1/6).
12 castling_squares(castling(queenside, black), 8/5, 8/1, 8/3,
13   8/6).
14 castling_squares(castling(queenside, white), 1/5, 1/1, 1/3,
15   1/4).
16
17 %% check(+Board: board, +Player: turn).
18 %
19 %   Validates the given player is in check.
20 %
21 %   @param Board the board
22 %   @param Player the player to verify
23 check(Board, Player) :-
24   % Get the location of the player's king.
25   piece_at(Board, KingSquare, piece(king, Player)),
26   % Get the enemy of the given player.
27   state:enemy(Player, Enemy),
28   % Verify that the enemy is attacking the king of the given
29   %   player.
30   state:attacking(Board, Enemy, KingSquare), !.
31
32 %% clear(+Board: board, +R/C: square, -After: board).
33 %
34 %   Removes a piece from the board.
35 %
36 %   @param Before the initial board
37 %   @param R/C the square at which the piece should be removed
38 %   @param After the resulting board
39 clear(Before, R/C, After) :-
40   nth1_row(Before, R, RowBefore),
41   piece_replace(RowBefore, C, none, RowAfter),
42   row_replace(Before, R, RowAfter, After).
```

```

37
38 %% enemy(+Board: board, +Square: square, +turn).
39 %
40 %   Validates the given square contains any piece that
    belongs to the given
41 %   player's enemy.
42 %
43 %   @param Board the board
44 %   @param Square the square to check
45 %   @param black-white the player
46 enemy(Board, Square, black) :- piece_at(Board, Square,
    piece(_, white)).
47 enemy(Board, Square, white) :- piece_at(Board, Square,
    piece(_, black)).
48
49 %% free(+Board: board, +R/C: square).
50 %
51 %   Succeeds if the given square does not contain any piece.
52 %
53 %   @param Board the board
54 %   @param R/C the square
55 free(Board, R/C) :-
56     % Extract the row from the board.
57     nth1_row(Board, R, Row),
58     % Verify the piece at the given square is none.
59     nth1_piece(Row, C, none).
60
61 %% mine(+Board: board, +Square: square, +Player: turn).
62 %
63 %   Validates the given square contains any piece that
    belongs to the given
64 %   player.
65 %
66 %   @param Board the board
67 %   @param Square the square to check
68 %   @param Player the player to match
69 mine(Board, Square, Player) :- piece_at(Board, Square,
    piece(_, Player)).
70
71 %% move_piece(+Before: board, +From: square, +To: square,
    -After: board).
72 %
73 %   Moves a piece on the board.
74 %
75 %   @param Before the initial board
76 %   @param From the current square
77 %   @param To the destination square
78 %   @param After the resulting board

```

```

79 move_piece(Before, From, To, After) :-
80     % Get the piece to replace.
81     piece_at(Before, From, Piece),
82     % Remove the piece from the square.
83     set_piece(Before, From, none, Removed),
84     % Put the piece at the destination square.
85     set_piece(Removed, To, Piece, After).
86
87 %% nth1_piece(+board:board, +integer, +R: -R: row).
88 %
89 %   Gets the row at the given row number on the board.
90 %
91 %   @param board the Board
92 %   @param integer the row number
93 %   @param R the row at that row number
94 nth1_row(board(R, _, _, _, _, _, _), 1, R).
95 nth1_row(board(_, R, _, _, _, _, _), 2, R).
96 nth1_row(board(_, _, R, _, _, _, _), 3, R).
97 nth1_row(board(_, _, _, R, _, _, _), 4, R).
98 nth1_row(board(_, _, _, _, R, _, _), 5, R).
99 nth1_row(board(_, _, _, _, _, R, _), 6, R).
100 nth1_row(board(_, _, _, _, _, _, R), 7, R).
101 nth1_row(board(_, _, _, _, _, _, _), 8, R).
102
103 %% nth1_piece(+row:row, +integer, +R: -P: piece).
104 %
105 %   Gets the piece at the given column in the row.
106 %
107 %   @param row the row
108 %   @param integer the column number
109 %   @param P the piece at that position
110 nth1_piece(row(P, _, _, _, _, _, _), 1, P).
111 nth1_piece(row(_, P, _, _, _, _, _), 2, P).
112 nth1_piece(row(_, _, P, _, _, _, _), 3, P).
113 nth1_piece(row(_, _, _, P, _, _, _), 4, P).
114 nth1_piece(row(_, _, _, _, P, _, _), 5, P).
115 nth1_piece(row(_, _, _, _, _, P, _), 6, P).
116 nth1_piece(row(_, _, _, _, _, _, P), 7, P).
117 nth1_piece(row(_, _, _, _, _, _, _), 8, P).
118
119 %% piece_at(+Board:board, +R/C: square, -Piece: piece).
120 %
121 %   Gets the piece at a given square;
122 %
123 %   @param Board the board
124 %   @param R/C the square
125 %   @param Piece the piece at that square.
126 piece_at(Board, R/C, Piece) :-

```

```

127 % Extract the row of the piece from the board.
128 nth1_row(Board, R, Row),
129 % Extract the column of the piece from the row.
130 nth1_piece(Row, C, Piece).
131
132 %% piece_replace(+row:row, +integer, +P: Piece, -row:row).
133 %
134 % Replaces a piece in a row.
135 %
136 % @param row the initial row
137 % @param integer the column number to replace
138 % @param P the new piece to put on the given column
139 % @param row the resulting row
140 piece_replace(row(_, P2, P3, P4, P5, P6, P7, P8), 1, P,
141 row(P, P2, P3, P4, P5, P6, P7, P8)).
142 piece_replace(row(P1, _, P3, P4, P5, P6, P7, P8), 2, P,
143 row(P1, P, P3, P4, P5, P6, P7, P8)).
144 piece_replace(row(P1, P2, _, P4, P5, P6, P7, P8), 3, P,
145 row(P1, P2, P, P4, P5, P6, P7, P8)).
146 piece_replace(row(P1, P2, P3, _, P5, P6, P7, P8), 4, P,
147 row(P1, P2, P3, P, P5, P6, P7, P8)).
148 piece_replace(row(P1, P2, P3, P4, _, P6, P7, P8), 5, P,
149 row(P1, P2, P3, P4, P, P6, P7, P8)).
150 piece_replace(row(P1, P2, P3, P4, P5, _, P7, P8), 6, P,
151 row(P1, P2, P3, P4, P5, P, P7, P8)).
152 piece_replace(row(P1, P2, P3, P4, P5, P6, _, P8), 7, P,
153 row(P1, P2, P3, P4, P5, P6, P, P8)).
154 piece_replace(row(P1, P2, P3, P4, P5, P6, P7, _), 8, P,
155 row(P1, P2, P3, P4, P5, P6, P7, P)).
156
157 %% row_replace(+board:board, +integer, +R: row,
158 -board:board).
159 %
160 % Replaces a row on the board.
161 %
162 % @param board the initial board
163 % @param integer the row number to replace
164 % @param R the new row to put on the given row number
165 % @param board the resulting board
166 row_replace(board(_, R2, R3, R4, R5, R6, R7, R8), 1, R,
167 board(R, R2, R3, R4, R5, R6, R7, R8)).
168 row_replace(board(R1, _, R3, R4, R5, R6, R7, R8), 2, R,
169 board(R1, R, R3, R4, R5, R6, R7, R8)).
170 row_replace(board(R1, R2, _, R4, R5, R6, R7, R8), 3, R,
171 board(R1, R2, R, R4, R5, R6, R7, R8)).
172 row_replace(board(R1, R2, R3, _, R5, R6, R7, R8), 4, R,
173 board(R1, R2, R3, R, R5, R6, R7, R8)).

```

```

161 row_replace(board(R1, R2, R3, R4, _, R6, R7, R8), 5, R,
    board(R1, R2, R3, R4, R, R6, R7, R8)).
162 row_replace(board(R1, R2, R3, R4, R5, _, R7, R8), 6, R,
    board(R1, R2, R3, R4, R5, R, R7, R8)).
163 row_replace(board(R1, R2, R3, R4, R5, R6, _, R8), 7, R,
    board(R1, R2, R3, R4, R5, R6, R, R8)).
164 row_replace(board(R1, R2, R3, R4, R5, R6, R7, _), 8, R,
    board(R1, R2, R3, R4, R5, R6, R7, R)).
165
166 %% set_piece(+Before:board, +R/C: square, +Piece:piece,
    -After:board).
167 %
168 %   Sets the given piece on the board at square R/C.
169 %
170 %   @param Before the initial board
171 %   @param R/C the square to put the piece on
172 %   @param Piece the piece to put
173 %   @param After the resulting board
174 set_piece(Before, R/C, Piece, After) :-
175     % Fetch the row to put the piece on.
176     nth1_row(Before, R, RowBefore),
177     % Put the piece on the correct row.
178     piece_replace(RowBefore, C, Piece, RowAfter),
179     % Put the modified row in the resulting board.
180     row_replace(Before, R, RowAfter, After), !.
181
182 %% square(R/C).
183 %
184 %   Validates or generates a square on the playfield.
185 %
186 %   @param R/C the row and column of the square
187 square(R/C) --> between(1, 8, R), between(1, 8, C).
188 square(R/C) :- between(1, 8, R), between(1, 8, C).

```

src/fen.pl

```

1 :- module(fen, []).
2
3 :- use_module(library(dcg/basics)).
4
5 %% parse(FenString:string, State:state).
6 %
7 %   Parses a given FEN string into a state. Can also perform
    the reverse
8 %   conversion operation.
9 %
10 %   @param FenString the FEN string

```

```

11 % @param State the corresponding state
12 parse(FenString, State) :-
13     % Apply the DCG to the string to obtain the state, or
      reverse.
14     phrase(state(State), FenString).
15
16 % DCG parser for a state, based on the FEN specifications.
17 state(state(Board, Turn, Castling, EnPassant, HalfCount,
      FullCount)) -->
18     board(Board), " ", turn(Turn), " ", castlings(Castling),
19     " ", en_passant(EnPassant), " ", half_count(HalfCount),
20     " ", full_count(FullCount).
21
22 % DCG parser for a board.
23 board(board(R1, R2, R3, R4, R5, R6, R7, R8)) -->
24     row(R8), "/", row(R7), "/", row(R6), "/", row(R5),
25     "/", row(R4), "/", row(R3), "/", row(R2), "/", row(R1).
26
27 % DCG parser for castling.
28 castlings([]) --> "-", !.
29 castlings(Cs) --> castling_possibilities(Cs).
30 % castlings[] kan niet opnieuw gebruikt worden want dan zou
      K-kq ook geldig zijn
31 castling_possibilities([]) --> [].
32 castling_possibilities([C | Cs]) --> castling(C),
      castling_possibilities(Cs).
33
34 % DCG parser for the castling possibilities.
35 castling(castling(kingside, black)) --> "k".
36 castling(castling(kingside, white)) --> "K".
37 castling(castling(queenside, black)) --> "q".
38 castling(castling(queenside, white)) --> "Q".
39
40 % DCG parser for the en passant squares.
41 en_passant(none) --> "-".
42 en_passant(3/1) --> "a3".
43 en_passant(3/2) --> "b3".
44 en_passant(3/3) --> "c3".
45 en_passant(3/4) --> "d3".
46 en_passant(3/5) --> "e3".
47 en_passant(3/6) --> "f3".
48 en_passant(3/7) --> "g3".
49 en_passant(3/8) --> "h3".
50 en_passant(6/1) --> "a6".
51 en_passant(6/2) --> "b6".
52 en_passant(6/3) --> "c6".
53 en_passant(6/4) --> "d6".
54 en_passant(6/5) --> "e6".

```

```

55 en_passant(6/6) --> "f6".
56 en_passant(6/7) --> "g6".
57 en_passant(6/8) --> "h6".
58
59 % DCG parser for the full-move counter.
60 full_count(N) --> integer(N), {N > 0}.
61
62 % DCG parser for the half-move counter.
63 half_count(N) --> integer(N), {N >= 0}.
64
65 % DCG parser for pieces.
66 piece(piece(bishop, black)) --> "b".
67 piece(piece(bishop, white)) --> "B".
68 piece(piece(king, black)) --> "k".
69 piece(piece(king, white)) --> "K".
70 piece(piece(knight, black)) --> "n".
71 piece(piece(knight, white)) --> "N".
72 piece(piece(pawn, black)) --> "p".
73 piece(piece(pawn, white)) --> "P".
74 piece(piece(queen, black)) --> "q".
75 piece(piece(queen, white)) --> "Q".
76 piece(piece(rook, black)) --> "r".
77 piece(piece(rook, white)) --> "R".
78
79 % DCG parser for pieces on a row.
80 piece(Piece, Left, Left1) --> piece(Piece), {Left1 is
    Left-1, Left1 >= 0}.
81
82 % DCG parser for pieces on a row.
83 pieces([], 0) --> [].
84
85 % DCG parser for empty squares on a row.
86 pieces([none, none, none, none, none, none, none, none], 8)
    --> "8".
87 pieces([none, none, none, none, none, none, none | X], I)
    --> "7", !,
88     {I1 is I-7, I1 >= 0},
89     pieces(X, I1).
90 pieces([none, none, none, none, none, none | X], I) --> "6",
    !,
91     {I1 is I-6, I1 >= 0},
92     pieces(X, I1).
93 pieces([none, none, none, none, none | X], I) --> "5", !,
94     {I1 is I-5, I1 >= 0},
95     pieces(X, I1).
96 pieces([none, none, none, none | X], I) --> "4", !,
97     {I1 is I-4, I1 >= 0},
98     pieces(X, I1).

```



```

99 pieces([none, none, none | X], I) --> "3", !,
100 {I1 is I-3, I1 >= 0},
101 pieces(X, I1).
102 pieces([none, none | X], I) --> "2", !,
103 {I1 is I-2, I1 >= 0},
104 pieces(X, I1).
105 pieces([none | X], I) --> "1", !,
106 {I1 is I-1, I1 >= 0},
107 pieces(X, I1).
108
109 % DCG parser for pieces on a row.
110 pieces([H|R], Left) --> piece(H, Left, Left1), pieces(R,
    Left1).
111
112 % DCG parser (FEN to state) for a row.
113 row(row(A, B, C, D, E, F, G, H)) --> {var(A), !},
114 pieces(Row, 8),
115 {
116     flatten(Row, [A, B, C, D, E, F, G, H])
117 }.
118
119 % DCG parser (state to FEN) for a row.
120 row(row(A, B, C, D, E, F, G, H)) --> {nonvar(A), !},
121 pieces([A, B, C, D, E, F, G, H], 8).
122
123 % DCG parser for turns.
124 turn(black) --> "b".
125 turn(white) --> "w".

```

src/main.pl

```

1 #!/usr/bin/env swipl
2
3 :- use_module(fen).
4 :- use_module(minimax).
5 :- use_module(movement).
6 :- use_module(state).
7
8 % Set the correct string mode.
9 :- set_prolog_flag(double_quotes, chars).
10 % Increase the stack limit to allow deeper searches.
11 :- set_prolog_flag(stack_limit, 2 000 000 000).
12 % Set the initialization goal.
13 :- initialization(main, main).
14
15 %% parse(+Argv:list, -State:state).
16 %

```

```

17 % Parses the given FEN input arguments into a state.
18 %
19 % @param Argv the arguments to parse
20 % @param State the resulting state
21 parse(Argv, State) :-
22     % Concatenate all the separate arguments into a FEN string.
23     atomic_list_concat(Argv, ' ', FenRaw),
24     % Convert the string to the correct representation.
25     atom_codes(FenRaw, FenString),
26     % Parse the FEN string into a state.
27     fen:parse(FenString, State).
28
29 %% validate(+Player:turn, +State:state).
30 %
31 % Validates a given state. A state is valid if the given
32 % player is not in
33 % check.
34 % @param Player the player's king that may not be attacked
35 % @param State the state to validate
36 validate(Player, State) :-
37     % Validate the player is not in check.
38     \+ state:check(State, Player).
39
40 %% write_draw().
41 %
42 % Writes out the "DRAW" message.
43 write_draw() :-
44     % Convert the string "DRAW" to a list of atoms that can be
45     % printed.
46     atom_codes(Draw, "DRAW"),
47     % Print the string.
48     write(Draw), nl.
49 %% write_fen(+State:state).
50 %
51 % Writes out a given state in FEN notation.
52 %
53 % @param State the state to print
54 write_fen(State) :-
55     % Parse the state to its corresponding FEN notation.
56     fen:parse(ResultFen, State),
57     % Convert the FEN string to atoms that can be printed.
58     atom_codes(ResultRaw, ResultFen),
59     % Write out the result.
60     write(ResultRaw), nl.
61
62 %% main(+Argv:list).

```

```

63 %
64 % Regular main function that prints out the best move. This
    function will, if
65 % successful, halt the program.
66 %
67 % @param Argv the input arguments
68 main(Argv) :-
69     % Precondition for this function: FEN has 6 parts.
70     length(Argv, 6),
71     % Parse the arguments into a state.
72     parse(Argv, State),
73     % Disable the garbage collector during calculations.
74     set_prolog_flag(gc, false),
75     % Perform minimax and alpha-beta pruning on the state to
        get the next best
76     % state for the current player.
77     minimax:alphabeta(State, 3, NextState),
78     % Re-instantiate the garbage collector.
79     set_prolog_flag(gc, false), garbage_collect,
80     % Write out the resulting move.
81     write_fen(NextState),
82     % Halt execution.
83     halt(0).
84
85 %% main(+Argv:list).
86 %
87 % Testing main function that prints out all next moves.
    This function will, if
88 % successful, halt the program.
89 %
90 % @param Argv the input arguments
91 main(Argv) :-
92     % Precondition: there should be 6 FEN arguments and 1 TEST
        argument.
93     length(Argv, 7),
94     % Remove the TEST argument from the input.
95     exclude(=('TEST'), Argv, ArgvClean),
96     % Parse the arguments into a state.
97     parse(ArgvClean, State),
98     % Get the current player.
99     state:turn(State, Player),
100    % Generate all possible moves.
101    movement:all_moves(State, Moves),
102    % Apply all possible moves to obtain the resulting states.
103    maplist(state:apply_move(State), Moves, ResultStates),
104    % Remove the states where the player is in check.
105    include(validate(Player), ResultStates, ValidStates),
106    % Verify at least one legal move is available.

```

```

107 length(ValidStates, AmountStates), AmountStates > 0,
108 % Write the FEN representation for every valid state.
109 maplist(write_fen(), ValidStates), !,
110 % Halt execution.
111 halt(0).
112
113 %% main(+Argv:list).
114 %
115 % Main function that prints out a draw. A draw is assumed
    if the regular and
116 % testing main functions have not found any valid moves.
117 %
118 % @param Argv the input arguments
119 main(Argv) :-
120     % The amount of input arguments must be 6 or 7 (testing).
121     (length(Argv, 6) ; length(Argv, 7)),
122     % Write out a draw.
123     write_draw(),
124     % Halt execution.
125     halt(0).

```

src/minimax.pl

```

1 :- module(minimax, []).
2
3 :- use_module(movement).
4 :- use_module(state).
5
6 %% adjust_depth(+Moves: list, +Depth: integer, -NewDepth:
    integer).
7 %
8 % Adjusts the depth based on the amount of moves. If the
    amount of moves is
9 % very small, the tree is allowed to go deeper one more
    level to possibly
10 % choose a better move.
11 %
12 % @param Moves the moves
13 % @param Depth the current depth
14 % @param the adjusted depth
15 adjust_depth(Moves, Depth, NewDepth) :-
16     % Get the amount of moves.
17     length(Moves, I),
18     I < 4,
19     NewDepth is Depth + 1, !.
20 adjust_depth(_, Depth, Depth).
21

```

```

22 %% alphabeta(+InitialState: state, +MaxDepth: integer,
    -BestState: state).
23 %
24 % Entry point for the alpha-beta pruning minimax function.
25 %
26 % @param InitialState the current state
27 % @param MaxDepth the maximum depth to search
28 % @param BestState the best move
29 alphabeta(InitialState, MaxDepth, BestState) :-
30     % Get the current turn from the game state.
31     state:turn(InitialState, Player),
32     % Start the alpha-beta pruning.
33     alphabeta(InitialState, Player, MaxDepth, -999 999 999,
        +999 999 999, BestState, _).
34
35 %% alphabeta(+State: state, +Player: player, +Depth:
    integer, +Alpha: integer,
36 %%             +Beta: integer, -BestNextState: state,
    -BestScore: integer).
37 %
38 % Alpha-beta pruning function.
39 %
40 % @param State the current state
41 % @param Player the current player
42 % @param Depth the maximum remaining depth to search
43 % @param Alpha the alpha bound
44 % @param Beta the beta bound
45 % @param BestNextState the best move
46 % @param BestScore the best move's score
47 alphabeta(State, Player, 0, _, _, _, Score) :-
48     % Base case, don't search any further.
49     score(State, Player, Score), !.
50 alphabeta(Current, Player, Depth, Alpha, Beta,
    BestNextState, BestScore) :-
51     % Generate all next moves.
52     movement:all_moves(Current, NextMoves),
53     % Get the adjusted depth.
54     adjust_depth(NextMoves, Depth, NewDepth),
55     % Apply all next moves.
56     maplist(state:apply_move(Current), NextMoves, NextStates),
57     % Find the best move.
58     best(NextStates, Player, NewDepth, Alpha, Beta,
        BestNextState, BestScore), !.
59 alphabeta(State, Player, _, _, _, _, Score) :-
60     % No more moves are available, return the score of the
        current state.
61     score(State, Player, Score).
62

```

```

63 %% best(+States: list, +Player: player, +Depth: integer,
    +Alpha: integer,
64 %%      +Beta: integer, -BestState: state, -BestScore:
    integer).
65 %
66 % Evaluate all states and get the best state (recursive
    step).
67 %
68 % @param States the states to evaluate
69 % @param Player the current player
70 % @param Depth the current depth
71 % @param Alpha the alpha bound
72 % @param Beta the beta bound
73 % @param BestState the best move
74 % @param BestScore the best score
75 best([State1 | States], Player, Depth, Alpha, Beta,
    BestState, BestScore) :-
76     % Decrease the depth.
77     Deeper is Depth - 1,
78     % Perform alpha-beta pruning on the next depth.
79     alphabeta(State1, Player, Deeper, Alpha, Beta, _, Score1),
80     % Evaluate to find the best move.
81     evaluate(State1, Score1, States, Player, Depth, Alpha,
        Beta, BestState, BestScore).
82
83 %% better_of(+State1: state, +Score1: integer, +State2:
    state, +Score2: integer,
84 %%      -BestState: state, -BestScore: integer).
85 %
86 % Get the best state between two states.
87 %
88 % @param State1 the first state
89 % @param Score1 the first score
90 % @param State2 the second state
91 % @param Score2 the second score
92 % @param BestState the best move
93 % @param BestScore the best score
94 better_of(State1, Score, _, Score, _, State1, Score) :-
95     % Choose either one if both scores are equal.
96     random(0, 2, 1), !.
97 better_of(_, Score, State2, Score, _, State2, Score).
98 better_of(State1, Score1, _, Score2, Player, State1, Score1)
    :-
99     % Current player's turn -> Maximize.
100     turn(State1, Player), Score1 > Score2, !.
101 better_of(State1, Score1, _, Score2, Player, State1, Score1)
    :-
102     % Enemy's turn -> Minimize.

```

```

103 \+ turn(State1, Player), Score1 < Score2, !.
104 better_of(_, _, State2, Score2, _, State2, Score2).
105
106 %% bounds(+Alpha: integer, +Beta: integer, +State: state,
    +Score: integer,
107 %%         +Player: turn, -NewAlpha: integer, -NewBeta:
    integer).
108 %
109 %   Update the alpha or beta bound.
110 %
111 %   @param Alpha the current alpha
112 %   @param Beta the current beta
113 %   @param State the State
114 %   @param Score the score for the given state
115 %   @param Player the player
116 %   @param NewAlpha the updated alpha value
117 %   @param NewBeta the updated beta value
118 bounds(Alpha, Beta, State, Score, Player, Score, Beta) :-
119     % Verify if the alpha should be updated.
120     turn(State, Player), Score > Alpha, !.
121 bounds(Alpha, Beta, State, Score, Player, Alpha, Score) :-
122     % Verify if the alpha should be updated.
123     \+ turn(State, Player), Score < Beta, !.
124 bounds(Alpha, Beta, _, _, _, Alpha, Beta).
125
126 %% evaluate(+State: state, +Score: integer, +States: list,
    +Player: turn,
127 %%         +Depth: integer, +Alpha: integer, +Beta:
    integer, -BestState: state,
128 %%         -BestScore: integer).
129 %
130 %   Evaluates the current state and determines the best one.
131 %
132 %   @param State the State
133 %   @param Score the score for the given state
134 %   @param State other states to evaluate
135 %   @param Player the player
136 %   @param Depth the current depth
137 %   @param Alpha the current alpha
138 %   @param Beta the current beta
139 %   @param BestState the best move
140 %   @param BestScore the best score
141 evaluate(State, Score, [], _, _, _, State, Score) :- !.
142 evaluate(State, Score, States, Player, Depth, Alpha, Beta,
    BestState, BestScore) :-
143     % Calculate the new alpha and beta bounds.
144     bounds(Alpha, Beta, State, Score, Player, NewAlpha,
        NewBeta),

```

```

145 % Determine the next best move.
146 best(States, Player, Depth, NewAlpha, NewBeta, State1,
    Score1),
147 % Determine the best move between the lower depth and the
    current best.
148 better_of(State, Score, State1, Score1, Player, BestState,
    BestScore).
149
150 %% piece_score(+Piece: piece, -Score: integer).
151 %
152 % Get the minmax score for a given piece.
153 %
154 % @param Piece the piece
155 % @param Score the score
156 piece_score(bishop, 3 000).
157 piece_score(king, 1 000 000).
158 piece_score(knight, 12 000).
159 piece_score(pawn, 1 000).
160 piece_score(queen, 25 000).
161 piece_score(rook, 5 000).
162
163 %% score(+State: state, +Player: turn, -Score: integer).
164 %
165 % Evaluate a given state and player.
166 %
167 % @param State the State
168 % @param Player the player
169 % @param Score the score
170 score(State, Player, Score) :-
171     % Get the enemy of the player.
172     state:enemy(Player, Enemy),
173     % Get this player's score.
174     score_sub(State, Player, MyScore),
175     % Get the enemy's score.
176     score_sub(State, Enemy, EnemyScore),
177     % Calculate the final score.
178     Score is MyScore - EnemyScore.
179
180 %% score_sub(+State: state, +Player: turn, -Score: integer).
181 %
182 % Evaluate a given state and player, subroutine.
183 %
184 % @param State the State
185 % @param Player the player
186 % @param Score the score
187 score_sub(State, Player, Score) :-
188     % Find all pieces of the player.

```



```

189 findall(Type, state:piece_at(State, _, piece(Type,
    Player)), Pieces),
190 % Map each piece to its score.
191 maplist(piece_score, Pieces, PieceScores),
192 % Calculate the sum of all the scores.
193 sum_list(PieceScores, Score).
194
195 %% turn(+State: state, -Turn: turn).
196 %
197 % Get the current player, this is the enemy of the player
    in the state, since
198 % since the state has the next player (the move is already
    applied).
199 %
200 % @param State the State
201 % @param Turn the turn
202 turn(State, Turn) :-
203     % Get the opposite of the turn in the state.
204     \+ state:turn(State, Turn), !.
205 turn(State, Turn) :-
206     % Get the turn from the state.
207     state:turn(State, Player),
208     % Get the enemy of the turn from the state.
209     state:enemy(Player, Turn).

```

src/movement.pl

```

1 :- module(movement, []).
2
3 :- use_module(board).
4 :- use_module('movement/bishop', [moves/4 as bishop_moves]).
5 :- use_module('movement/king', [moves/4 as king_moves]).
6 :- use_module('movement/knight', [moves/4 as knight_moves]).
7 :- use_module('movement/pawn', [moves/4 as pawn_move]).
8 :- use_module('movement/positions').
9 :- use_module('movement/queen', [moves/4 as queen_moves]).
10 :- use_module('movement/rook', [moves/4 as rook_moves]).
11 :- use_module(state).
12
13 %% all_moves(+State: state, +Square: square, +Piece: piece,
    -Moves: list).
14 %
15 % Gets all available moves for the given
16 %
17 % @param From the starting square
18 % @param Direction the direction of the move
19 % @param To the destination square

```

```

20 all_moves(State, Moves) :-
21     % Find all moves on the board.
22     findall(X, board:square(X), AllSquares),
23     % Generate all moves for every square.
24     maplist(all_moves(State), AllSquares, AllMoves),
25     % Flatten the found moves.
26     flatten(AllMoves, Moves).
27
28 %% all_moves(+State: state, +Square: square, -Moves: list).
29 %
30 % Gets all available moves for the given location.
31 %
32 % @param State the current state
33 % @param Square the square to generate moves for
34 % @param Moves the moves found
35 all_moves(State, Square, Moves) :-
36     % Get the current player.
37     state:turn(State, Turn),
38     % Get the piece at the given square, only generate moves
39     % for the current
40     state:piece_at(State, Square, piece(Type, Turn)), !,
41     % Get all moves.
42     all_moves(State, Square, piece(Type, Turn), Moves).
43 all_moves(_, _, []).
44
45 %% all_moves(+State: state, +Square: square, +Piece: piece,
46 % -Moves: list).
47 %
48 % Gets all available moves for the given piece at the given
49 % location.
50 %
51 % @param State the current state
52 % @param Square the square to generate moves for
53 % @param Piece the piece at the given square
54 % @param Moves the moves found
55 all_moves(State, Square, piece(bishop, Turn), Moves) :-
56     bishop_moves(State, Square, Turn, Moves), !.
57 all_moves(State, Square, piece(knight, Turn), Moves) :-
58     knight_moves(State, Square, Turn, Moves), !.
59 all_moves(State, Square, piece(king, Turn), Moves) :-
60     king_moves(State, Square, Turn, Moves), !.
61 all_moves(State, Square, piece(pawn, Turn), Moves) :-
62     pawn_move(State, Square, Turn, Moves), !.
63 all_moves(State, Square, piece(queen, Turn), Moves) :-
64     queen_moves(State, Square, Turn, Moves), !.
65 all_moves(State, Square, piece(rook, Turn), Moves) :-
66     rook_moves(State, Square, Turn, Moves), !.

```

```

65 all_moves(_, _, _, []).
66
67 %% attacking(+Board: board, +Piece: piece, +Current: square,
    -Target: square).
68 %
69 % Gets whether a square can attack another square.
70 %
71 % @param Board the board
72 % @param Piece the piece at the given Current square
73 % @param Current the starting piece
74 % @param Target the target to attack
75 attacking(Board, piece(bishop, _), Current, Target) :-
76     positions:bishop_attacks(Current, Direction, Target),
77     path_clear(Board, move(Current, Direction, Target)).
78 attacking(_, piece(king, _), Current, Target) :-
79     positions:king_attacks(Current, Target).
80 attacking(_, piece(knight, _), Current, Target) :-
81     positions:knight_attacks(Current, Target).
82 attacking(_, piece(pawn, Color), Current, Target) :-
83     positions:pawn_attacks(Current, Color, Target).
84 attacking(Board, piece(queen, _), Current, Target) :-
85     positions:queen_attacks(Current, Direction, Target),
86     path_clear(Board, move(Current, Direction, Target)).
87 attacking(Board, piece(rook, _), Current, Target) :-
88     positions:rook_attacks(Current, Direction, Target),
89     path_clear(Board, move(Current, Direction, Target)).
90
91 %% path_clear(+Board: board, +Path: move).
92 %
93 % Verifies that a path contains only empty squares.
94 %
95 % @param Board the board
96 % @param Path the path containing the path to follow
97 path_clear(_, move(Square, _, Square)) :- !.
98 path_clear(Board, move(From, Direction, To)) :-
99     % Get the next square on the path.
100     path_next(From, Direction, Next),
101     % Verify this square is empty using the subroutine.
102     path_clear_sub(Board, move(Next, Direction, To)).
103
104 %% path_clear_sub(+Board: board, +Path: move).
105 %
106 % Verifies that a path contains only empty squares,
    subroutine. This routine is
107 % required, because else the starting square that contains
    the piece would also
108 % need to be empty, which is obviously always false.
109 %

```

```

110 % @param Board the board
111 % @param Path the path containing the path to follow
112 path_clear_sub(_, move(Square, _, Square)) :- !.
113 path_clear_sub(Board, move(From, Direction, To)) :-
114     % Verify the square is empty.
115     board:free(Board, From),
116     % Get the next square on the path.
117     path_next(From, Direction, Next),
118     % Verify this square is empty using the subroutine.
119     path_clear_sub(Board, move(Next, Direction, To)).
120
121 %% path_next(+R/C: square, +Direction: direction, -R1/C1:
    square).
122 %
123 % Get the next square on a path in a given direction.
124 %
125 % @param R/C the current square
126 % @param Direction the direction of the path
127 % @param R1/C1 the next square
128 path_next(R/C, down, R1/C1) :- R1 is R - 1.
129 path_next(R/C, down_left, R1/C1) :- R1 is R - 1, C1 is C - 1.
130 path_next(R/C, down_right, R1/C1) :- R1 is R - 1, C1 is C +
    1.
131 path_next(R/C, left, R1/C1) :- C1 is C - 1.
132 path_next(R/C, right, R1/C1) :- C1 is C + 1.
133 path_next(R/C, up, R1/C1) :- R1 is R + 1.
134 path_next(R/C, up_left, R1/C1) :- R1 is R + 1, C1 is C - 1.
135 path_next(R/C, up_right, R1/C1) :- R1 is R + 1, C1 is C + 1.

```

src/state.pl

```

1 :- module(state, []).
2
3 :- use_module(board).
4 :- use_module(movement).
5
6 %% apply_move(+Before: state, +Move: move, -After: state).
7 %
8 % Applies the given castling move to the state.
9 %
10 % @param Before the current state
11 % @param Move the move to apply
12 % @param After the resulting state, after applying the move
13 apply_move(Before, move(castling, Castling), After) :-
14     % Extract the board from the state.
15     board(Before, BeforeBoard),
16

```

```

17  %%%% BOARD
18  % Update the castling squares.
19  board:castling_squares(Castling, KingFrom, RookFrom,
20  KingTo, RookTo),
21  % Castle the king.
22  board:move_piece(BeforeBoard, KingFrom, KingTo,
23  AfterBoardKing),
24  % Castle the rook.
25  board:move_piece(AfterBoardKing, RookFrom, RookTo,
26  AfterBoard),
27  % Save the board.
28  update_board(Before, AfterBoard, BoardState),
29
30  %%%% TURN
31  update_turn(BoardState, TurnState),
32
33  %%%% CASTLING RIGHTS
34  update_castling(TurnState, CastlingState),
35
36  %%%% EN PASSANT
37  reset_enpassant(CastlingState, EnPassantState),
38
39  %%%% HALF-MOVE COUNTER
40  inc_halfcount(EnPassantState, HCState),
41
42  %%%% FULL-MOVE COUNTER
43  inc_fullcount(HCState, After).
44
45 %% apply_move(+Before: state, +Move: move, -After: state).
46 %
47 % Applies the given capture move to the state.
48 %
49 % @param Before the current state
50 % @param Move the move to apply
51 % @param After the resulting state, after applying the move
52 apply_move(Before, move(capture, From, To), After) :-
53 % Extract the board from the state.
54 board(Before, BeforeBoard),
55
56 %%%% BOARD
57 % Move the piece on the board.
58 board:move_piece(BeforeBoard, From, To, AfterBoard),
59 % Save the board.
60 update_board(Before, AfterBoard, BoardState),
61
62 %%%% TURN
63 update_turn(BoardState, TurnState),
64

```

```

62  %%%% CASTLING RIGHTS
63  update_castling(TurnState, CastlingState),
64
65  %%%% EN PASSANT
66  reset_enpassant(CastlingState, EnPassantState),
67
68  %%%% HALF-MOVE COUNTER
69  reset_halfcount(EnPassantState, HCState),
70
71  %%%% FULL-MOVE COUNTER
72  inc_fullcount(HCState, After).
73
74 %% apply_move(+Before: state, +Move: move, -After: state).
75 %
76 %   Applies the given en-passant move to the state.
77 %
78 %   @param Before the current state
79 %   @param Move the move to apply
80 %   @param After the resulting state, after applying the move
81 apply_move(Before, move(en_passant, From, EPCapture, To),
82   After) :-
83   % Extract the board from the state.
84   board(Before, BeforeBoard),
85   %%%% BOARD
86   % Move the piece.
87   board:move_piece(BeforeBoard, From, To, AfterMove),
88   % Clear het en passant square.
89   board:clear(AfterMove, EPCapture, AfterBoard),
90   % Save the board.
91   update_board(Before, AfterBoard, BoardState),
92
93   %%%% TURN
94   update_turn(BoardState, TurnState),
95
96   %%%% CASTLING RIGHTS
97   update_castling(TurnState, CastlingState),
98
99   %%%% EN PASSANT
100  reset_enpassant(CastlingState, EnPassantState),
101
102  %%%% HALF-MOVE COUNTER
103  reset_halfcount(EnPassantState, HCState),
104
105  %%%% FULL-MOVE COUNTER
106  inc_fullcount(HCState, After).
107
108 %% apply_move(+Before: state, +Move: move, -After: state).

```

```

109 %
110 %   Applies the given move to the state.
111 %
112 %   @param Before the current state
113 %   @param Move the move to apply
114 %   @param After the resulting state, after applying the move
115 apply_move(Before, move(move, From, EPSquare, To), After) :-
116   % Extract the board from the state.
117   board(Before, BeforeBoard),
118
119   %%%% BOARD
120   % Move the piece.
121   board:move_piece(BeforeBoard, From, To, AfterBoard),
122   % Save the board.
123   update_board(Before, AfterBoard, BoardState),
124
125   %%%% TURN
126   update_turn(BoardState, TurnState),
127
128   %%%% CASTLING RIGHTS
129   update_castling(TurnState, CastlingState),
130
131   %%%% EN PASSANT
132   update_enpassant(CastlingState, EPSquare, EnPassantState),
133
134   %%%% HALF-MOVE COUNTER
135   reset_halfcount(EnPassantState, HCState),
136
137   %%%% FULL-MOVE COUNTER
138   inc_fullcount(HCState, After).
139 apply_move(Before, move(move, From, To), After) :-
140   % Extract the board from the state.
141   board(Before, BeforeBoard),
142
143   %%%% BOARD
144   % Move the piece.
145   board:move_piece(BeforeBoard, From, To, AfterBoard),
146   % Save the board.
147   update_board(Before, AfterBoard, BoardState),
148
149   %%%% TURN
150   update_turn(BoardState, TurnState),
151
152   %%%% CASTLING RIGHTS
153   update_castling(TurnState, CastlingState),
154
155   %%%% EN PASSANT
156   reset_enpassant(CastlingState, EnPassantState),

```

```

157
158     %%%% HALF-MOVE COUNTER
159     % Get a piece that may be captured en-passant.
160     board:piece_at(BeforeBoard, From, MovedPiece),
161     inc_halfcount(EnPassantState, MovedPiece, HCState),
162
163     %%%% FULL-MOVE COUNTER
164     inc_fullcount(HCState, After).
165
166 %% apply_move(+Before: state, +Move: move, -After: state).
167 %
168 %   Applies the given promotion move to the state.
169 %
170 %   @param Before the current state
171 %   @param Move the move to apply
172 %   @param After the resulting state, after applying the move
173 apply_move(Before, move(promotion, Piece, From, To), After)
174 :-
175     % Extract the board from the state.
176     board(Before, BeforeBoard),
177     % Extract the turn from the state.
178     turn(Before, Turn),
179
180     %%%% BOARD
181     % Clear the current square.
182     board:clear(BeforeBoard, From, ClearedBoard),
183     % Set the piece on the square.
184     board:set_piece(ClearedBoard, To, piece(Piece, Turn),
185                     AfterBoard),
186     % Save the board.
187     update_board(Before, AfterBoard, BoardState),
188
189     %%%% TURN
190     update_turn(BoardState, TurnState),
191
192     %%%% CASTLING RIGHTS
193     update_castling(TurnState, CastlingState),
194
195     %%%% EN PASSANT
196     reset_enpassant(CastlingState, EnPassantState),
197
198     %%%% HALF-MOVE COUNTER
199     reset_halfcount(EnPassantState, HCState),
200
201     %%%% FULL-MOVE COUNTER
202     inc_fullcount(HCState, After).

```



```

202 %% attacking(+Board: board, +Player: player, +Square:
    square).
203 %
204 %   Validates whether a square is under attack by the given
    player.
205 %
206 %   @param Board the board
207 %   @param Player the attacker
208 %   @param Square the square that must be checked
209 attacking(Board, Player, Square) :-
210     % Get a piece of the current player.
211     board:piece_at(Board, Position, piece(Type, Player)),
212     % Validate if this piece is attacking the square.
213     movement:attacking(Board, piece(Type, Player), Position,
        Square), !.
214
215 %% attacking_squares(+Board: board, +Player: player,
    +Attacked: list).
216 %
217 %   Get all squares that are under attack by the given player.
218 %
219 %   @param Board the board
220 %   @param Player the attacker
221 %   @param Attacked the squares that are under attack
222 attacking_squares(Board, Player, Attacked) :-
223     % Find all squares on the board.
224     findall(X, board:square(X), AllSquares),
225     % Filter out the squares that are not under attack.
226     include(attacking(Board, Player), AllSquares, Attacked).
227
228 %% board(+State: state, -Board: board).
229 %
230 %   Extract the board from the state.
231 %
232 %   @param State the state
233 %   @param Board the board
234 board(state(Board, _, _, _, _, _), Board).
235
236 %% can_castle(+Board: board, +Type: castling).
237 %
238 %   Validates a given castling right.
239 %
240 %   @param Board the board
241 %   @param Type the castling right to validate
242 can_castle(Board, castling(Type, Color)) :-
243     % Get the king and rook squares.
244     board:castling_squares(castling(Type, Color), KingFrom,
        RookFrom, _, _),

```

```

245 % Validate the king is at the correct position.
246 board:piece_at(Board, KingFrom, piece(king, Color)),
247 % Validate the rook is at the correct position.
248 board:piece_at(Board, RookFrom, piece(rook, Color)).
249
250 %% castling(+State: state, -C: list).
251 %
252 % Extract the castling rights from the state.
253 %
254 % @param State the state
255 % @param C the castling rights
256 castling(state(_, _, C, _, _, _), C).
257
258 %% check(+State: state, +Player: turn).
259 %
260 % Validates the given player is in check.
261 %
262 % @param State the state
263 % @param Player the player
264 check(State, Player) :-
265     % Extract the board from the state.
266     board(State, Board),
267     % Validate the in-check status.
268     board:check(Board, Player).
269
270 % Enemies of the given players.
271 enemy(black, white).
272 enemy(white, black).
273
274 %% en_passant(+State: state, -EP: square).
275 %
276 % Extract the en-passant square from the state.
277 %
278 % @param State the state
279 % @param EP the en-passant square
280 en_passant(state(_, _, _, EP, _, _), EP).
281
282 %% full_count(+State: state, -FC: integer).
283 %
284 % Extract the full-move counter from the state.
285 %
286 % @param State the state
287 % @param FC the full move counter
288 full_count(state(_, _, _, _, _, FC), FC).
289
290 %% inc_fullcount(+State1: state, -State2: state).
291 %
292 % Increments the full-move counter

```

```

293 %
294 % @param State1 the initial state
295 % @param State2 the updated state
296 inc_fullcount(state(B, white, C, EP, HC, FC), state(B,
    white, C, EP, HC, FC1)) :-
297     % The counter should be incremented since black has moved.
298     succ(FC, FC1).
299 inc_fullcount(state(B, black, C, EP, HC, FC), state(B,
    black, C, EP, HC, FC)).
300
301 %% half_count(+State: state, -HC: integer).
302 %
303 % Extract the half-move counter from the state.
304 %
305 % @param State the state
306 % @param HC the half move counter
307 half_count(state(_, _, _, _, HC, _), HC).
308
309 %% inc_halfcount(+State1: state, -State2: state).
310 %
311 % Increments the half-move counter
312 %
313 % @param State1 the initial state
314 % @param State2 the updated state
315 inc_halfcount(state(B, T, C, EP, HC, FC), state(B, T, C, EP,
    HC1, FC)) :-
316     % Increment the half-move counter.
317     succ(HC, HC1).
318
319 %% inc_fullcount(+State1: state, +Piece: piece, -State2:
    state).
320 %
321 % Increments the half-move counter based on the piece.
322 %
323 % @param State1 the initial state
324 % @param Piece the piece
325 % @param State2 the updated state
326 inc_halfcount(state(B, T, C, EP, _, FC), piece(pawn, _),
    state(B, T, C, EP, 0, FC)) :- !.
327 inc_halfcount(state(B, T, C, EP, HC, FC), _, state(B, T, C,
    EP, HC1, FC)) :-
328     % Draw condition.
329     succ(HC, HC1), HC1 < 75.
330
331 %% piece_at(+State: state, +Square: square, -Piece: piece).
332 %
333 % Get the piece at the given square.
334 %

```

```

335 % @param State the state
336 % @param Square the square
337 % @param Piece the piece at the given square
338 piece_at(State, Square, Piece) :-
339     % Extract the board from the state.
340     board(State, Board),
341     % Get the piece at the square.
342     board:piece_at(Board, Square, Piece).
343
344 %% reset_enpassant(+State1: state, -State2: state).
345 %
346 % Resets the en-passant square.
347 %
348 % @param State1 the initial state
349 % @param State2 the updated state
350 reset_enpassant(state(B, T, C, _, HC, FC), state(B, T, C,
    none, HC, FC)).
351
352 %% reset_halfcount(+State1: state, -State2: state).
353 %
354 % Resets the half-move counter.
355 %
356 % @param State1 the initial state
357 % @param State2 the updated state
358 reset_halfcount(state(B, T, C, EP, _, FC), state(B, T, C,
    EP, 0, FC)).
359
360 %% turn(+State: state, -Turn: turn).
361 %
362 % Extract the turn from the state.
363 %
364 % @param State the state
365 % @param Turn the turn
366 turn(state(_, Turn, _, _, _, _), Turn).
367
368 %% update_board(+State1: state, +Board: board, -State2:
    state).
369 %
370 % Updates the board in the state.
371 %
372 % @param State1 the initial state
373 % @param Board the board to replace
374 % @param State2 the updated state
375 update_board(state(_, T, C, EP, HC, FC), Board, state(Board,
    T, C, EP, HC, FC)).
376
377 %% update_castling(+State1: state, -State2: state).
378 %

```

```

379 % Updates the castling rights in the state.
380 %
381 % @param State1 the initial state
382 % @param State2 the updated state
383 update_castling(state(B, T, C, EP, HC, FC), state(B, T, C1,
    EP, HC, FC)) :-
384     % Include all valid rights.
385     include(can_castle(B), C, C1).
386
387 %% update_enpassant(+State1: state, +EP: square, -State2:
    state).
388 %
389 % Updates the en-passant square in the state.
390 %
391 % @param State1 the initial state
392 % @param EP the en-passant square
393 % @param State2 the updated state
394 update_enpassant(state(B, T, C, _, HC, FC), EP, state(B, T,
    C, EP, HC, FC)).
395
396 %% update_turn(+State1: state, +Turn: turn, -State2: state).
397 %
398 % Updates the turn in the state.
399 %
400 % @param State1 the initial state
401 % @param Turn the turn to replace
402 % @param State2 the updated state
403 update_turn(state(B, _, C, EP, HC, FC), Turn, state(B, Turn,
    C, EP, HC, FC)).
404 update_turn(state(B, white, C, EP, HC, FC), state(B, black,
    C, EP, HC, FC)).
405 update_turn(state(B, black, C, EP, HC, FC), state(B, white,
    C, EP, HC, FC)).

```

src/movement/bishop.pl

```

1 :- module(bishop, []).
2
3 :- use_module('../board').
4 :- use_module('../state').
5 :- use_module(positions).
6
7 %% move(+Board: board, +Turn: turn, +move:move: -move:move).
8 %
9 % Formulates a capture move.
10 %
11 % @param Board the current board

```

```

12 % @param Turn the owner of the bishop
13 % @param move the unprocessed move
14 % @param move(capture) the capturing move
15 move(Board, Turn, move(From, _, To), move(capture, From,
    To)) :-
16     % Verify the destination square contains an enemy piece.
17     board:enemy(Board, To, Turn), !.
18
19 %% move(+Board: board, +Turn: turn, +move:move: -move:move).
20 %
21 % Formulates a regular walking move.
22 %
23 % @param Board the current board
24 % @param Turn the owner of the bishop
25 % @param move the unprocessed move
26 % @param move(move) the walking move
27 move(Board, _, move(From, _, To), move(move, From, To)) :-
28     % Verify the destination square does not contain any piece.
29     board:free(Board, To).
30
31 %% moves(+State: state, +Square: square, +Turn: turn,
    -Moves: list).
32 %
33 % Finds all valid moves for a bishop on the given Square.
34 %
35 % @param State the current game state
36 % @param Square the square that contains a bishop
37 % @param Turn the owner of the bishop
38 % @param Moves the resulting available moves
39 moves(State, Square, Turn, Moves) :-
40     % Extract the board from the state.
41     state:board(State, Board),
42     % Get all bishop moves from the current square.
43     positions:bishop_moves(Square, BishopMoves),
44     % Validate every found bishop move.
45     include(movement:path_clear(Board), BishopMoves,
        FilteredMoves),
46     % Convert the moves to either a move or a capture.
47     convlist(move(Board, Turn), FilteredMoves, Moves).

```

src/movement/king.pl

```

1 :- module(king, []).
2
3 :- use_module('../board').
4 :- use_module('../movement').
5 :- use_module('../state').

```

```

6
7 %% castle(+State: state, +Square:, +Turn: turn, +move:move).
8 %
9 %   Formulates a castling move (kingside).
10 %
11 %   @param State the current game state
12 %   @param Square the current king square
13 %   @param Turn the owner of the king
14 %   @param move(castling) the castling move
15 castle(State, Square, Turn, move(castling,
    castling(kingside, Turn))) :-
16     % Verify the king is on a valid castling square.
17     board:castling_squares(castling(kingside, Turn), Square,
        RookSquare, _, _),
18     % Extract the board from the state.
19     state:board(State, Board),
20     % Extract the castling possibilities from the state.
21     state:castling(State, Castlings),
22     % Verify this castling type may be executed.
23     member(castling(kingside, Turn), Castlings),
24     % Verify there is no piece between the king and the rook.
25     movement:path_clear(Board, move(Square, right,
        RookSquare)).
26
27 %% castle(+State: state, +Square:, +Turn: turn, +move:move).
28 %
29 %   Formulates a castling move (queenside).
30 %
31 %   @param State the current game state
32 %   @param Square the current king square
33 %   @param Turn the owner of the king
34 %   @param move(castling) the castling move
35 castle(State, Square, Turn, move(castling,
    castling(queenside, Turn))) :-
36     % Verify the king is on a valid castling square.
37     board:castling_squares(castling(queenside, Turn), Square,
        RookSquare, _, _),
38     % Extract the board from the state.
39     state:board(State, Board),
40     % Extract the castling possibilities from the state.
41     state:castling(State, Castlings),
42     % Verify this castling type may be executed.
43     member(castling(queenside, Turn), Castlings),
44     % Verify there is no piece between the king and the rook.
45     movement:path_clear(Board, move(Square, left, RookSquare)).
46
47 %% move(+Board: board, +Turn: turn, +move:move: -move:move).
48 %

```

```

49 %   Formulates a capture move.
50 %
51 %   @param Board the current board
52 %   @param Turn the owner of the king
53 %   @param move the unprocessed move
54 %   @param move(capture) the capturing move
55 move(Board, Turn, move(From, To), move(capture, From, To)) :-
56     % Verify the destination square contains an enemy piece.
57     board:enemy(Board, To, Turn), !.
58
59 %% move(+Board: board, +Turn: turn, +move:move: -move:move).
60 %
61 %   Formulates a regular walking move.
62 %
63 %   @param Board the current board
64 %   @param Turn the owner of the king
65 %   @param move the unprocessed move
66 %   @param move(move) the walking move
67 move(Board, _, move(From, To), move(move, From, To)) :-
68     % Verify the destination square does not contain any piece.
69     board:free(Board, To).
70
71 %% moves(+State: state, +Square: square, +Turn: turn, -list).
72 %
73 %   Finds all valid moves for a knight on the given Square.
74 %
75 %   @param State the current game state
76 %   @param Square the square that contains a king
77 %   @param Turn the owner of the king
78 %   @param Moves the resulting available moves
79 moves(State, Square, Turn, [FilteredMoves, CastlingMoves]) :-
80     % Extract the board from the state.
81     state:board(State, Board),
82
83     % Get all king moves from the current square.
84     positions:king_moves(Square, KingMoves),
85     % Convert the moves to either a move or a capture.
86     convlist(move(Board, Turn), KingMoves, FilteredMoves),
87
88     % Find all castling moves from the current square and game
89     % state.
90     findall(C, castle(State, Square, Turn, C), CastlingMoves).

```

src/movement/knight.pl

```

1 :- module(knight, []).
2

```



```

3 :- use_module('../board').
4 :- use_module('../state').
5 :- use_module(positions).
6
7 %% move(+Board: board, +Turn: turn, +move:move: -move:move).
8 %
9 %   Formulates a capture move.
10 %
11 %   @param Board the current board
12 %   @param Turn the owner of the knight
13 %   @param move the unprocessed move
14 %   @param move(capture) the capturing move
15 move(Board, Turn, move(From, To), move(capture, From, To)) :-
16     % Verify the destination square contains an enemy piece.
17     board:enemy(Board, To, Turn), !.
18
19 %% move(+Board: board, +Turn: turn, +move:move: -move:move).
20 %
21 %   Formulates a regular walking move.
22 %
23 %   @param Board the current board
24 %   @param Turn the owner of the knight
25 %   @param move the unprocessed move
26 %   @param move(move) the walking move
27 move(Board, _, move(From, To), move(move, From, To)) :-
28     % Verify the destination square does not contain any piece.
29     board:free(Board, To).
30
31 %% moves(+State: state, +Square: square, +Turn: turn,
32         -Moves: list).
33 %
34 %   Finds all valid moves for a knight on the given Square.
35 %
36 %   @param State the current game state
37 %   @param Square the square that contains a knight
38 %   @param Turn the owner of the knight
39 %   @param Moves the resulting available moves
40 moves(State, Square, Turn, Moves) :-
41     % Extract the board from the state.
42     state:board(State, Board),
43     % Get all knight moves from the current square.
44     positions:knight_moves(Square, KnightMoves),
45     % Convert the moves to either a move or a capture.
46     convlist(move(Board, Turn), KnightMoves, Moves).

```

src/movement/pawn.pl

```
1 :- module(pawn, []).
2
3 :- use_module('../board').
4 :- use_module('../movement').
5 :- use_module('../state').
6
7 %% move(+State: state, +Turn: turn, +move:move: -move:move).
8 %
9 %   Formulates a capture move.
10 %
11 %   @param State the state
12 %   @param Turn the owner of the pawn
13 %   @param move the unprocessed move
14 %   @param move(capture) the capturing move
15 capture(State, Turn, move(From, To), move(capture, From,
    To)) :-
16     % Extract the board from the state.
17     state:board(State, Board),
18     % Verify the destination is not a promotion square.
19     \+ promotion_square(To),
20     % Verify the destination square contains an enemy piece.
21     board:enemy(Board, To, Turn).
22
23 %% move(+State: state, +Turn: turn, +move: move,
    -PromotionMoves: list).
24 %
25 %   Formulates a promotion-capture move.
26 %
27 %   @param State the state
28 %   @param Turn the owner of the pawn
29 %   @param move the unprocessed move
30 %   @param PromotionMoves the promotion moves
31 capture(State, Turn, move(From, To), PromotionMoves) :-
32     % Extract the board from the state.
33     state:board(State, Board),
34     % Verify the destination is a promotion square.
35     promotion_square(To),
36     % Verify the destination square contains an enemy piece.
37     board:enemy(Board, To, Turn),
38     % Get all possible promotions.
39     bagof(Move, promotion_move(Move, From, To),
        PromotionMoves).
40
41 %% move(+State: state, +Turn: turn, +move: move, -move:
    move).
42 %
```

```

43 % Formulates an en-pasant capture move.
44 %
45 % @param S the state
46 % @param Turn the owner of the pawn
47 % @param move the unprocessed move
48 % @param move(en_pasant) the capturing move
49 capture(S, Turn, move(SR/SC, DR/DC), move(en_pasant, SR/SC,
    SR/DC, DR/DC)) :-
50 % Extract the board from the state.
51 state:board(S, Board),
52 % Extract the en-pasant square from the state.
53 state:en_pasant(S, DR/DC),
54 % Verify the en-pasant square contains an enemy pawn.
55 board:enemy(Board, SR/DC, Turn),
56 % Verify the destination square does not contain any piece.
57 board:free(Board, DR/DC).
58
59 %% move(+Board: board, +Turn: turn, +move: move, -move:
    move).
60 %
61 % Formulates a pawn double walking move, for initial
    positions.
62 %
63 % @param Board the board
64 % @param Turn the owner of the pawn
65 % @param move the unprocessed move
66 % @param move(move) the walking move
67 move(Board, _, move(From, EP, To), move(move, From, EP, To))
    :-
68 % Verify the skipped square does not contain any piece.
69 board:free(Board, EP),
70 % Verify the destination square does not contain any piece.
71 board:free(Board, To).
72
73 %% move(+Board: board, +Turn: turn, +move: move,
    PromotionMoves: list).
74 %
75 % Formulates a promotion move.
76 %
77 % @param Board the board
78 % @param Turn the owner of the pawn
79 % @param move the unprocessed move
80 % @param PromotionMoves the promotion moves
81 move(Board, _, move(From, To), PromotionMoves) :-
82 % Verify the destination is a promotion square.
83 promotion_square(To),
84 % Verify the destination square does not contain any piece.
85 board:free(Board, To),

```

```

86 % Get all possible promotions.
87 bagof(Move, promotion_move(Move, From, To),
      PromotionMoves).
88
89 %% move(+Board: board, +Turn: turn, +move: move, -move:
      move).
90 %
91 % Formulates a pawn double walking move, for initial
      positions.
92 %
93 % @param Board the board
94 % @param Turn the owner of the pawn
95 % @param move the unprocessed move
96 % @param move(move) the walking move
97 move(Board, _, move(From, To), move(move, From, To)) :-
98 % Verify the destination is not a promotion square.
99 \+ promotion_square(To),
100 % Verify the destination square does not contain any piece.
101 board:free(Board, To).
102
103 %% moves(+State: state, +Square: square, +Turn: turn, -list).
104 %
105 % Finds all valid moves for a pawn on the given Square.
106 %
107 % @param State the current game state
108 % @param Square the square that contains a pawn
109 % @param Turn the owner of the pawn
110 % @param Moves the resulting available moves
111 moves(State, Square, Turn, [Moves, Captures, EnPassants]) :-
112 % Extract the board from the state.
113 state:board(State, Board),
114
115 % Get the pawn move from the current square.
116 positions:pawn(Square, Turn, PawnMove),
117 % Convert the move to either a move or a promotion.
118 convlist(move(Board, Turn), [PawnMove], Moves),
119
120 % Find all capture moves.
121 findall(X, positions:pawn_capture(Square, Turn, X),
      PawnCaptures),
122 % Convert the moves to either a capture or a
      promotion-capture.
123 convlist(capture(State, Turn), PawnCaptures, Captures),
124
125 % Find all en-passant moves.
126 findall(X, positions:pawn_enpassant(Square, Turn, X),
      EnPassantMove),
127 % Convert the moves to either a walk or a capture.

```

```

128   convlist(move(Board, Turn), EnPassantMove, EnPassants).
129
130 % Valid promotions.
131 promotion_move(move(promotion, bishop, From, To), From, To).
132 promotion_move(move(promotion, knight, From, To), From, To).
133 promotion_move(move(promotion, queen, From, To), From, To).
134 promotion_move(move(promotion, rook, From, To), From, To).
135
136 % Squares that allow pawn promotion.
137 promotion_square(1/_).
138 promotion_square(8/_).

```

src/movement/positions.pl

```

1 :- module(positions, []).
2
3 :- dynamic saved_bishop_moves/2.
4 :- dynamic saved_king_moves/2.
5 :- dynamic saved_knight_moves/2.
6 :- dynamic saved_rook_moves/2.
7
8 %% bishop_attacks(+From: square, +Direction: direction, +To:
   square).
9 %
10 % Verifies whether or not a bishop can attack a square,
   given a starting
11 % square.
12 %
13 % @param From the starting square
14 % @param Direction the direction of the move
15 % @param To the destination square
16 bishop_attacks(From, Direction, To) :- bishop(From,
   move(From, Direction, To)).
17
18 %% bishop_moves(+R/C: square, -Moves: list).
19 %
20 % Gets all bishop moves from the given starting square.
21 %
22 % @param R/C the starting square
23 % @param Moves the moves
24 bishop_moves(R/C, Moves) :-
25   % Fetch the moves from the cache.
26   saved_bishop_moves(R/C, Moves), !.
27 bishop_moves(R/C, Moves) :-
28   % Generate all moves.
29   setof(X, bishop(R/C, X), Moves),
30   % Store the moves in the cache for future use.

```

```

31  assertz(saved_bishop_moves(R/C, Moves)).
32
33 %% bishop(+R/C: square, -move: move).
34 %
35 %   Generates a bishop move.
36 %
37 %   @param R/C the starting square
38 %   @param move the move
39 bishop(R/C, move(R/C, down_left, R1/C1)) :-
40     % Diagonal offset.
41     between(1, 8, I),
42     % Diagonal movement.
43     R1 is R - I, C1 is C - I,
44     % Bounds checking.
45     between(1, R, R1), between(1, C, C1).
46 bishop(R/C, move(R/C, down_right, R1/C1)) :-
47     % Diagonal offset.
48     between(1, 8, I),
49     % Diagonal movement.
50     R1 is R - I, C1 is C + I,
51     % Bounds checking.
52     between(1, R, R1), between(C, 8, C1).
53 bishop(R/C, move(R/C, up_left, R1/C1)) :-
54     % Diagonal offset.
55     between(1, 8, I),
56     % Diagonal movement.
57     R1 is R + I, C1 is C - I,
58     % Bounds checking.
59     between(R, 8, R1), between(1, C, C1).
60 bishop(R/C, move(R/C, up_right, R1/C1)) :-
61     % Diagonal offset.
62     between(1, 8, I),
63     % Diagonal movement.
64     R1 is R + I, C1 is C + I,
65     % Bounds checking.
66     between(R, 8, R1), between(C, 8, C1).
67
68 %% king_attacks(+From: square, +Direction: direction, +To:
69     square).
69 %
70 %   Verifies whether or not a king can attack a square, given
71 %   a starting square.
72 %
73 %   @param From the starting square
74 %   @param Direction the direction of the move
75 %   @param To the destination square
76 king_attacks(From, To) :- king(From, move(From, To)).

```

```

77 %% king_moves(+R/C: square, -Moves: list).
78 %
79 % Gets all king moves from the given starting square.
80 %
81 % @param R/C the starting square
82 % @param Moves the moves
83 king_moves(R/C, Moves) :-
84     % Fetch the moves from the cache.
85     saved_king_moves(R/C, Moves), !.
86 king_moves(R/C, Moves) :-
87     % Generate all moves.
88     setof(X, king(R/C, X), Moves),
89     % Store the moves in the cache.
90     assertz(saved_king_moves(R/C, Moves)).
91
92 %% king(+R/C: square, -move: move).
93 %
94 % Generates a king move.
95 %
96 % @param R/C the starting square
97 % @param move the move
98 king(R/C, move(R/C, R1/C1)) :- R < 8, C < 8, R1 is R + 1, C1
    is C + 1.
99 king(R/C, move(R/C, R1/C)) :- R < 8, R1 is R + 1.
100 king(R/C, move(R/C, R1/C1)) :- R < 8, C > 1, R1 is R + 1, C1
    is C - 1.
101 king(R/C, move(R/C, R/C1)) :- C < 8, C1 is C + 1.
102 king(R/C, move(R/C, R/C1)) :- C > 1, C1 is C - 1.
103 king(R/C, move(R/C, R1/C1)) :- R > 1, C < 8, R1 is R - 1, C1
    is C + 1.
104 king(R/C, move(R/C, R1/C)) :- R > 1, R1 is R - 1.
105 king(R/C, move(R/C, R1/C1)) :- R > 1, C > 1, R1 is R - 1, C1
    is C - 1.
106
107 %% knight_attacks(+From: square, +Direction: direction, +To:
    square).
108 %
109 % Verifies whether or not a knight can attack a square,
    given a starting
110 % square.
111 %
112 % @param From the starting square
113 % @param Direction the direction of the move
114 % @param To the destination square
115 knight_attacks(From, To) :- knight(From, move(From, To)).
116
117 %% knight_moves(+R/C: square, -Moves: list).
118 %

```

```

119 % Gets all knight moves from the given starting square.
120 %
121 % @param R/C the starting square
122 % @param Moves the moves
123 knight_moves(R/C, Moves) :-
124     % Fetch the moves from the cache.
125     saved_knight_moves(R/C, Moves), !.
126 knight_moves(R/C, Moves) :-
127     % Generate all moves.
128     setof(X, knight(R/C, X), Moves),
129     % Store the moves in the cache for future use.
130     assertz(saved_knight_moves(R/C, Moves)).
131
132 %% knight(+R/C: square, -move: move).
133 %
134 % Generates a knight move.
135 %
136 % @param R/C the starting square
137 % @param move the move
138 knight(R/C, move(R/C, R1/C1)) :- R < 7, C < 8, R1 is R + 2,
139     C1 is C + 1.
140 knight(R/C, move(R/C, R1/C1)) :- R < 7, C > 1, R1 is R + 2,
141     C1 is C - 1.
142 knight(R/C, move(R/C, R1/C1)) :- R < 8, C < 7, R1 is R + 1,
143     C1 is C + 2.
144 knight(R/C, move(R/C, R1/C1)) :- R < 8, C > 2, R1 is R + 1,
145     C1 is C - 2.
146 knight(R/C, move(R/C, R1/C1)) :- R > 1, C < 7, R1 is R - 1,
147     C1 is C + 2.
148 knight(R/C, move(R/C, R1/C1)) :- R > 1, C > 2, R1 is R - 1,
149     C1 is C - 2.
150 knight(R/C, move(R/C, R1/C1)) :- R > 2, C < 8, R1 is R - 2,
151     C1 is C + 1.
152 knight(R/C, move(R/C, R1/C1)) :- R > 2, C > 1, R1 is R - 2,
153     C1 is C - 1.
154
155 %% pawn_attacks(+From: square, +Direction: direction, +To:
156     square).
157 %
158 % Verifies whether or not a pawn can attack a square, given
159 % a starting square.
160 %
161 % @param From the starting square
162 % @param Direction the direction of the move
163 % @param To the destination square
164 pawn_attacks(R/C, Turn, R1/C1) :- pawn_capture(R/C, Turn,
165     move(R/C, R1/C1)).

```



```

156 %% pawn_capture(+R/C: square, +Turn: turn, -move: move).
157 %
158 %   Generates a pawn move.
159 %
160 %   @param R/C the starting square
161 %   @param Turn the turn
162 %   @param move the move
163 pawn_capture(R/C, black, move(R/C, R1/C1)) :-
164     R > 1, C < 8, R1 is R - 1, C1 is C + 1.
165 pawn_capture(R/C, black, move(R/C, R1/C1)) :-
166     R > 1, C > 1, R1 is R - 1, C1 is C - 1.
167 pawn_capture(R/C, white, move(R/C, R1/C1)) :-
168     R < 8, C < 8, R1 is R + 1, C1 is C + 1.
169 pawn_capture(R/C, white, move(R/C, R1/C1)) :-
170     R < 8, C > 1, R1 is R + 1, C1 is C - 1.
171
172 %% pawn_enpassant(+R/C: square, +Turn: turn, -move: move).
173 %
174 %   Generates a pawn en-passant move.
175 %
176 %   @param R/C the initial square
177 %   @param Turn the turn
178 %   @param move the move
179 pawn_enpassant(7/C, black, move(7/C, 6/C, 5/C)) :-
180     between(1, 8, C).
181 pawn_enpassant(2/C, white, move(2/C, 3/C, 4/C)) :-
182     between(1, 8, C).
183
184 %% pawn(+R/C: square, +Turn: turn, -move: move).
185 %
186 %   Generates a pawn move.
187 %
188 %   @param R/C the starting square
189 %   @param Turn the turn
190 %   @param move the move
191 pawn(R/C, black, move(R/C, R1/C)) :- R > 1, R1 is R - 1.
192 pawn(R/C, white, move(R/C, R1/C)) :- R < 8, R1 is R + 1.
193
194 %% queen_attacks(+From: square, +Direction: direction, +To:
195 %   square).
196 %
197 %   Verifies whether or not a queen can attack a square,
198 %   given a starting square.
199 %
200 %   @param From the starting square
201 %   @param Direction the direction of the move
202 %   @param To the destination square

```

```

199 queen_attacks(From, Direction, To) :- bishop(From,
      move(From, Direction, To)), !.
200 queen_attacks(From, Direction, To) :- rook(From, move(From,
      Direction, To)).
201
202 %% queen_moves(+R/C: square, -Moves: list).
203 %
204 % Gets all queen moves from the given starting square.
205 %
206 % @param R/C the starting square
207 % @param Moves the moves
208 queen_moves(R/C, Moves) :-
209     % Fetch the bishop from the cache.
210     saved_bishop_moves(R/C, BishopMoves),
211     % Fetch the rook from the cache.
212     saved_rook_moves(R/C, RookMoves),
213     % Merge both move sets.
214     append(BishopMoves, RookMoves, Moves), !.
215 queen_moves(R/C, Moves) :-
216     % Generate all bishop moves.
217     setof(X, bishop(R/C, X), BishopMoves),
218     % Generate all rook moves.
219     setof(X, rook(R/C, X), RookMoves),
220     % Store the bishop moves in the cache for future use.
221     assertz(saved_bishop_moves(R/C, BishopMoves)),
222     % Store the rook moves in the cache for future use.
223     assertz(saved_rook_moves(R/C, RookMoves)),
224     % Merge both move sets.
225     append(BishopMoves, RookMoves, Moves).
226
227 %% rook_attacks(+From: square, +Direction: direction, +To:
      square).
228 %
229 % Verifies whether or not a rook can attack a square, given
      a starting square.
230 %
231 % @param From the starting square
232 % @param Direction the direction of the move
233 % @param To the destination square
234 rook_attacks(From, Direction, To) :- rook(From, move(From,
      Direction, To)).
235
236 %% rook_moves(+R/C: square, -Moves: list).
237 %
238 % Gets all rook moves from the given starting square.
239 %
240 % @param R/C the starting square
241 % @param Moves the moves

```

```

242 rook_moves(R/C, Moves) :-
243     % Fetch the moves from the cache.
244     saved_rook_moves(R/C, Moves), !.
245 rook_moves(R/C, Moves) :-
246     % Generate all moves.
247     setof(X, rook(R/C, X), Moves),
248     % Store the moves in the cache for future use.
249     assertz(saved_rook_moves(R/C, Moves)).
250
251 %% rook(+R/C: square, -move: move).
252 %
253 %   Generates a rook move.
254 %
255 %   @param R/C the starting square
256 %   @param move the move
257 rook(R/C, move(R/C, down, R1/C)) :- B is R - 1, between(1,
    B, R1).
258 rook(R/C, move(R/C, left, R/C1)) :- B is C - 1, between(1,
    B, C1).
259 rook(R/C, move(R/C, right, R/C1)) :- B is C + 1, between(B,
    8, C1).
260 rook(R/C, move(R/C, up, R1/C)) :- B is R + 1, between(B, 8,
    R1).

```

src/movement/queen.pl

```

1 :- module(queen, []).
2
3 :- use_module('../board').
4 :- use_module('../state').
5 :- use_module(positions).
6
7 %% move(+Board: board, +Turn: turn, +move:move: -move:move).
8 %
9 %   Formulates a capture move.
10 %
11 %   @param Board the current board
12 %   @param Turn the owner of the queen
13 %   @param move the unprocessed move
14 %   @param move(capture) the capturing move
15 move(Board, Turn, move(From, _, To), move(capture, From,
    To)) :-
16     % Verify the destination square contains an enemy piece.
17     board:enemy(Board, To, Turn), !.
18
19 %% move(+Board: board, +Turn: turn, +move:move: -move:move).
20 %

```

```

21 %   Formulates a regular walking move.
22 %
23 %   @param Board the current board
24 %   @param Turn the owner of the queen
25 %   @param move the unprocessed move
26 %   @param move(move) the walking move
27 move(Board, _, move(From, _, To), move(move, From, To)) :-
28     % Verify the destination square does not contain any piece.
29     board:free(Board, To).
30
31 %% moves(+State: state, +Square: square, +Turn: turn,
32     -Moves: list).
33 %
34 %   Finds all valid moves for a queen on the given Square.
35 %
36 %   @param State the current game state
37 %   @param Square the square that contains a queen
38 %   @param Turn the owner of the queen
39 %   @param Moves the resulting available moves
39 moves(State, Square, Turn, Moves) :-
40     % Extract the board from the state.
41     state:board(State, Board),
42     % Get all queen moves from the current square.
43     positions:queen_moves(Square, QueenMoves),
44     % Validate every found queen move.
45     include(movement:path_clear(Board), QueenMoves,
46         FilteredMoves),
47     % Convert the moves to either a move or a capture.
48     convlist(move(Board, Turn), FilteredMoves, Moves).

```

src/movement/rook.pl

```

1 :- module(rook, []).
2
3 :- use_module('../board').
4 :- use_module('../state').
5 :- use_module(positions).
6
7 %% move(+Board: board, +Turn: turn, +move:move: -move:move).
8 %
9 %   Formulates a capture move.
10 %
11 %   @param Board the current board
12 %   @param Turn the owner of the rook
13 %   @param move the unprocessed move
14 %   @param move(capture) the capturing move

```

```

15 move(Board, Turn, move(From, _, To), move(capture, From,
    To)) :-
16     % Verify the destination square contains an enemy piece.
17     board:enemy(Board, To, Turn), !.
18
19 %% move(+Board: board, +Turn: turn, +move:move: -move:move).
20 %
21 %   Formulates a regular walking move.
22 %
23 %   @param Board the current board
24 %   @param Turn the owner of the rook
25 %   @param move the unprocessed move
26 %   @param move(move) the walking move
27 move(Board, _, move(From, _, To), move(move, From, To)) :-
28     % Verify the destination square does not contain any piece.
29     board:free(Board, To).
30
31 %% moves(+State: state, +Square: square, +Turn: turn,
    -Moves: list).
32 %
33 %   Finds all valid moves for a rook on the given Square.
34 %
35 %   @param State the current game state
36 %   @param Square the square that contains a rook
37 %   @param Turn the owner of the rook
38 %   @param Moves the resulting available moves
39 moves(State, Square, Turn, Moves) :-
40     % Extract the board from the state.
41     state:board(State, Board),
42     % Get all rook moves from the current square.
43     positions:rook_moves(Square, RookMoves),
44     % Validate every found rook move.
45     include(movement:path_clear(Board), RookMoves,
        FilteredMoves),
46     % Convert the moves to either a move or a capture.
47     convlist(move(Board, Turn), FilteredMoves, Moves).

```

src/util/draw.pl

Dit bestand bevat ASCII tekens die niet geprint konden worden.

```

1 :- module(draw, []).
2
3 asciiPiece(bishop, black, '\u265D').
4 asciiPiece(king, black, '\u265A').
5 asciiPiece(knight, black, '\u265E').
6 asciiPiece(pawn, black, '\u265F').

```

```

7  asciiPiece(queen, black, '\u265B').
8  asciiPiece(rook, black, '\u265C').
9
10 asciiPiece(bishop, white, '\u2657').
11 asciiPiece(king, white, '\u2654').
12 asciiPiece(knight, white, '\u2658').
13 asciiPiece(pawn, white, '\u2659').
14 asciiPiece(queen, white, '\u2655').
15 asciiPiece(rook, white, '\u2656').
16
17 drawBoard(board(R1, R2, R3, R4, R5, R6, R7, R8)) :-
18     write('    '), drawRow(R8), write('    8'), nl, nl,
19     write('    '), drawRow(R7), write('    7'), nl, nl,
20     write('    '), drawRow(R6), write('    6'), nl, nl,
21     write('    '), drawRow(R5), write('    5'), nl, nl,
22     write('    '), drawRow(R4), write('    4'), nl, nl,
23     write('    '), drawRow(R3), write('    3'), nl, nl,
24     write('    '), drawRow(R2), write('    2'), nl, nl,
25     write('    '), drawRow(R1), write('    1'), nl, nl,
26     write('a    b    c    d    e    f    g    h'), nl, nl.
27
28 drawBoard(state(Board, _, _, _, _, _)) :- drawBoard(Board).
29
30 drawLine(0, _) :- !.
31 drawLine(N, C) :- N1 is N-1, write(C), drawLine(N1, C).
32
33 drawPiece(none) :- !, write('\u00B7').
34 drawPiece(piece(Type, Color)) :- asciiPiece(Type, Color,
35     AsciiCode), write(AsciiCode).
36
37 drawRow(row(P1, P2, P3, P4, P5, P6, P7, P8)) :-
38     drawPiece(P1),
39     write('    '), drawPiece(P2),
40     write('    '), drawPiece(P3),
41     write('    '), drawPiece(P4),
42     write('    '), drawPiece(P5),
43     write('    '), drawPiece(P6),
44     write('    '), drawPiece(P7),
45     write('    '), drawPiece(P8).

```