APSC 143 Course Project (Fall 2025)

Nick Mertin, MASc, EIT

October 22, 2025

Contents

There will be a scheduled update to this document on Monday, October 27, to add rubrics as well as to make any clarifications necessary based on questions received by then.

1 Overview

The course project is one of the major assessments in this course, worth a total of 20 % of your overall grade. This is split into 5 % for the proposal and 15 % for the final submission due at the end of the semester; additionally, there is an opportunity to earn up to 5 % bonus in your overall course grade for going above and beyond the requirements. The project depends on content covered in weeks 1 through 7. This document contains all information which you will need to complete the project; please read it in full, and reach out if you have any questions.

The objective of the project is to create a simple chess analysis engine, based on some initial starter code which is provided for you. The chess engine will take as input the sequence of moves made by a pair of players, in a standard notation which is specified in Section 3, and produce an output describing the state of the game after the specified moves, as described in Section 4. The rules of chess are described in Appendix B, and examples of input and corresponding output are given in Appendix C.

2 The Plan

Figure 1 shows a plan for the behaviour of the program, in the form of a flowchart, along with the functions from the starter code associated with each step. You will need to complete the tasks outlined in the following subsections. For each task, it is recommended that you plan out your approach before starting; this planning work can form the basis of the relevant section of your final report, and it will also help you to ensure that you are writing good code. Be sure to familiarize yourself with all expectations for your submission, as described in Section 5.

2.1 Board State Representation

Determine the fields necessary to represent the state of the board (`struct chess_board` in `board.h`), and write code to initialize this board state to the starting configuration of a chess game (`board_initialize()` in `board.c`). You need to consider everything relevant to applying the rules in Appendix B, including the positions of all pieces and whether either player is allowed to castle.

2.2 Move Representation

Determine the fields necessary to represent a single move (`struct chess_move` in `board.h`). The challenging part of this task is determining how to represent possibly incomplete information; as discussed in Section 3.3, some moves may not specify the previous/current square of the piece which is moving, and the code you write for Section 2.4 will need to modify the contents of the move fields to fill in this information. You should also think about how special moves such as castling and promotion will be represented.

2.3 Input Parsing

Write code to parse a move from standard input (`parse_move()` in `parser.c`), according to the format specified in Section 3. As noted in Section 5.3, this should be accomplished using the C standard library functions `getchar()` and `ungetchar()`. The details of this code will partly depend on your approach to Section 2.2.

In the event that the input does not correspond to the specified format, you should exit the program by calling the provided `panic()` function to generate an error message according to Section 4. This function has the same interface as `printf()` from the standard library; it takes as arguments a format string followed by a corresponding list of variadic arguments and never returns (it exits the program). You don't need to understand how this function works, you can just call it! One case of generating an error message is already provided as reference.

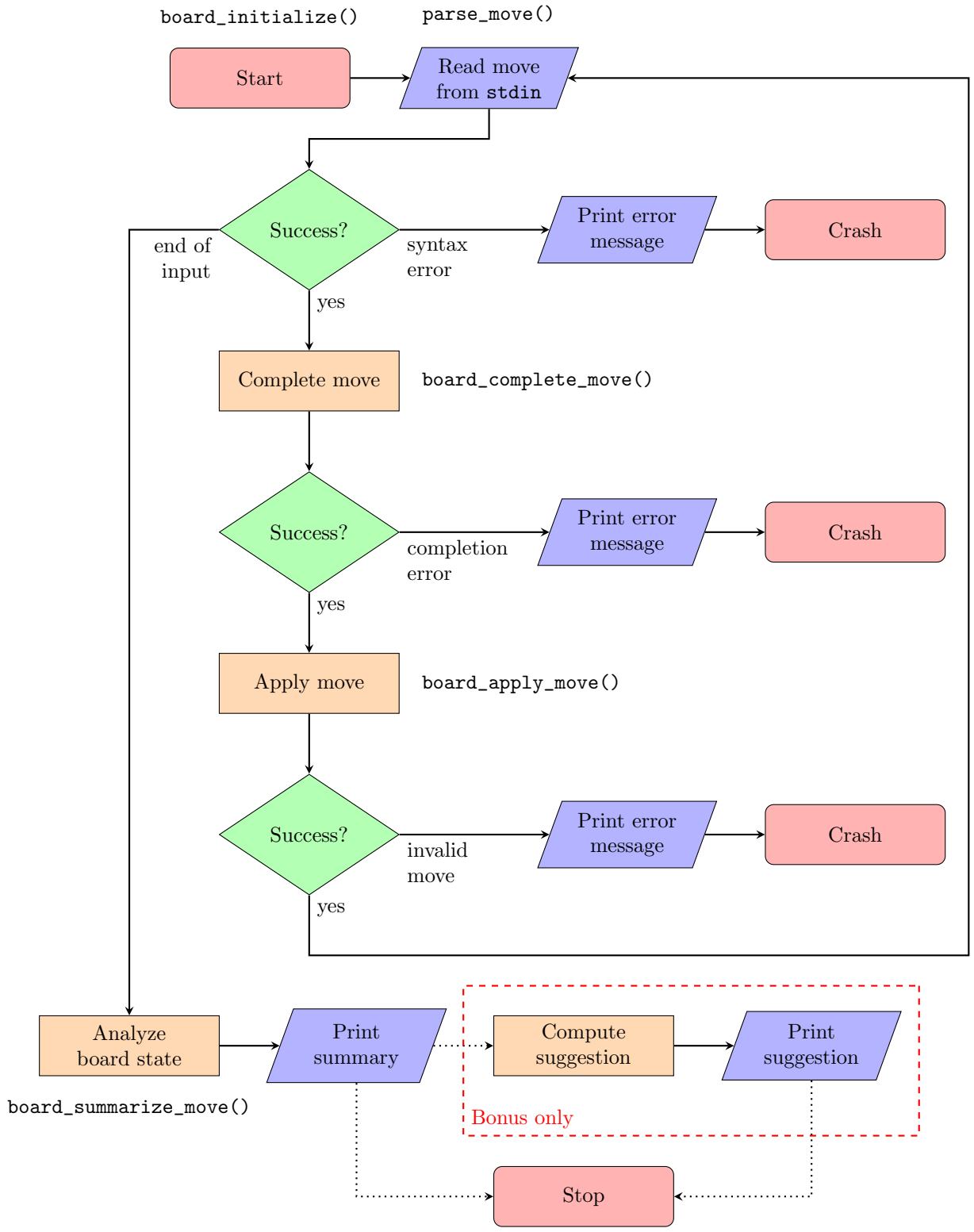


Figure 1: Flowchart showing the intended behaviour of the program and how it relates to the functions you are expected to fill in.

2.4 Move Completion

Write code to complete an incomplete move (`board_complete_move()` in `board.c`). The only information which is potentially missing in an incomplete move is the previous square of the piece which is missing; this task requires looking at both the current state of the board and the known details of the move to infer this information. The details of this code will heavily depend on your choice of representation for both board state and moves. If the necessary information cannot be inferred, the code should generate an error message according to Section 4, using the same tools discussed in Section 2.3.

2.5 Move Application

Write code to apply a move to the board state (`board_apply_move()` in `board.c`). This code may assume that the move is complete, but it does need to check that the move is legal in the current board state; assuming the move is legal, it should make all necessary changes to the board state to reflect the move. As with Section 2.4, the details of this code will heavily depend on your choice of representation for both board state and moves. If the move is illegal (including if the move is capturing but there is no piece to capture, or the opposite), the code should generate an error message according to Section 4, using the same tools discussed in Section 2.3.

2.6 Board State Classification

Write code to analyze the board state and print a summary of the outcome (`board_summarize()` in `board.c`), using the format specified in Section 4. The actual printing should be done using the C standard library function `printf()`. This task mainly comes down to writing code to determine two pieces of information: whether the next player has any legal moves, and if they don't, whether they are in check.

2.7 Move Recommendation (Bonus)

Extend `board_summarize()` with code to recommend a move for the next player, if the game is incomplete. To get full marks for this task, you must implement an algorithm which always produces a recommendation with the following properties, with documentation in the final report justifying that claim.

1. The recommended move must always be legal.
2. If there is a move which enables the player to force a checkmate in at most three moves including this one (i.e., no matter how the opponent plays, the current player may respond in such a way that ensures they win), then such a move must be recommended.
3. The recommended move cannot allow the opponent to force checkmate in at most 3 moves, unless such moves are the only legal ones.
4. If neither of the above strategic scenarios hold, there must be some strategic idea behind the recommendation algorithm (e.g., maximizing the value of opponent pieces captured minus own pieces captured).

No specific guidance is provided on how you should approach this bonus challenge; it is up to you, and, unlike the main project, it may require skills beyond weeks 1 through 7 of the course.

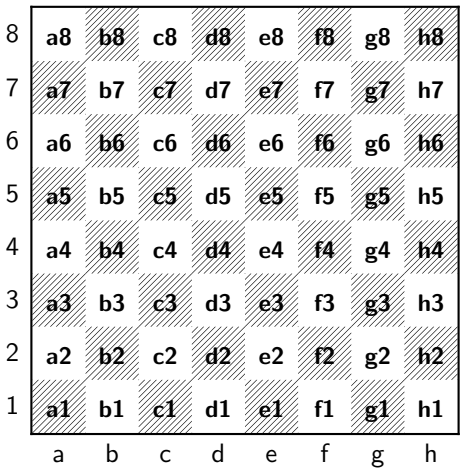
3 Input Format

The program will read, from standard input, the sequence of chess moves in a simplified version of algebraic notation[1], described below.

[1] `https://en.wikipedia.org/wiki/Algebraic_notation_(chess)`

3.1 The Board

Squares on the 8 by 8 chess board are identified by their *rank* (row), indicated by numbers **1** through **8** and *file* (column), indicated by letters **a** through **h**. Rank **1** is drawn at the bottom and is closest to the player using the white pieces.



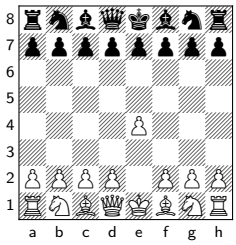
3.2 Pawn Moves

Moves by pawns are recorded by indicating the square which the pawn moved to; as long as we know from context which player is moving, the rules of chess mean that it is unambiguous which pawn is moving.

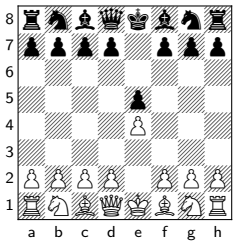
Example 3.1. Consider the input:

```
e4 e5 f3
```

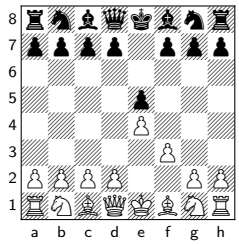
These moves result in the following board states:



e2–e4



e7–e5



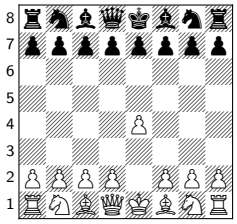
f2–f3

Recall that a pawn can capture by moving diagonally, and can only move diagonally when capturing. When a pawn captures, the move is prefixed by the pawn's previous file and the letter `x`.

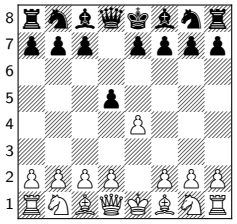
Example 3.2. Consider the input:

```
e4 d5 exd5
```

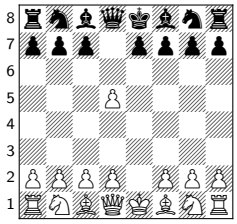
These moves result in the following board states:



e2–e4



d7–d5

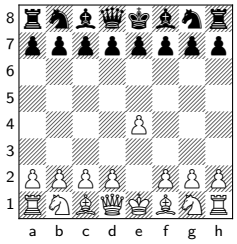


e4×d5

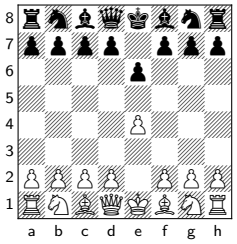
Example 3.3. The same notation is used for a pawn capturing *en passant*; consider the input:

```
e4 e6 e5 d5 exd6
```

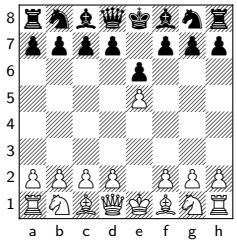
These moves result in the following board states:



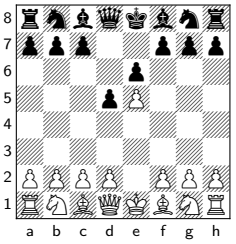
e2–e4



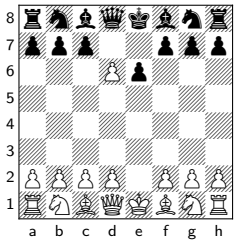
e7–e6



e4–e5



d7–d5



e5×d6 e.p.

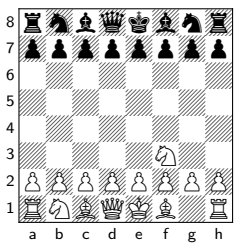
3.3 Regular Piece Moves

Regular moves by pieces other than pawns are recorded by writing a letter indicating the type of piece (`K` for king, `Q` for queen, `R` for rook, `B` for bishop, `N` for knight) followed by the coordinates of the destination square. If the move is a capture, then the letter `x` is inserted between the piece type and the coordinates.

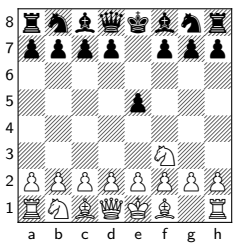
Example 3.4. Consider the input:

```
Nf3 e5 Nxe5
```

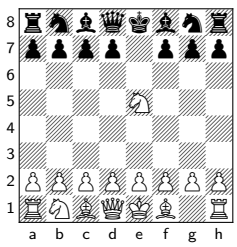
These moves result in the following board states:



♘**g1–f3**



e7–e5



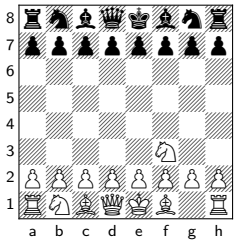
♘**f3×e5**

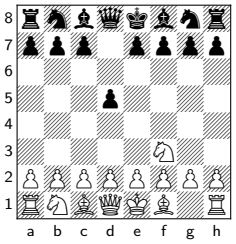
Unlike with pawns, sometimes more than one piece of the same type from the same player is able to move to the specified destination square. In this case, it is disambiguated by indicating the previous file and/or rank of the piece after the piece type indicator.

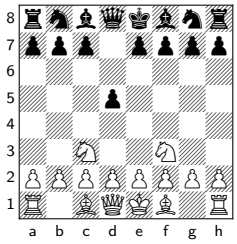
Example 3.5. Consider the input:

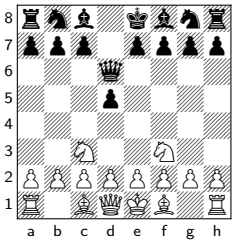
```
Nf3 d5 Nc3 Qd6 Nd4 e5 Ndb5
```

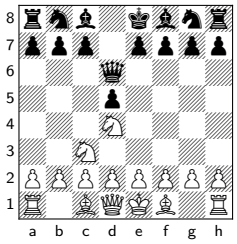
For the last move, either of white's knights could move to the target square **b5**; this is disambiguated by specifying that the knight in the **d** file is the one which is moving. These moves result in the following board states:

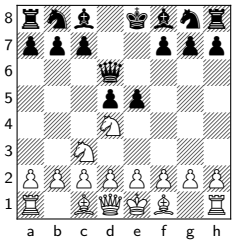

♘**g1–f3**

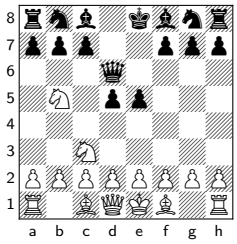

d7–d5


♘**b1–c3**


♕**d8–d6**


♘**f3–d4**


e7–e5


♘**d4–b5**

Remark. It is not an error for the previous file and/or rank to be specified unnecessarily, but it is an error for it to be omitted when it is necessary for determining the piece which is moving. For the purposes of this project, it is necessary if the player has more than one piece of the correct type in a position where it could potentially move to the target square, even if only one could legally move (e.g., because moving the other would leave the king in check).

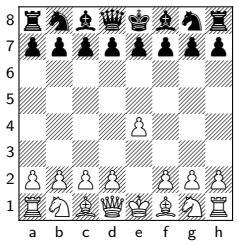
3.4 Castling

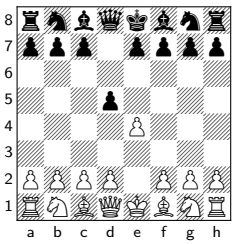
There are special notations for the two castling moves by either player: short (king-side) castles are indicated by `O-O`, and long (queen-side) castles are indicated by `O-O-O`.

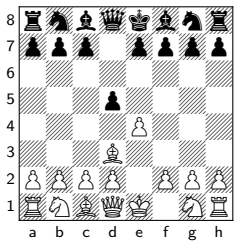
Example 3.6. Consider the input:

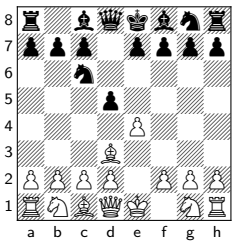
```
e4 d5 Bd3 Nc6 Nf3 Bg4 exd5 Qxd5 O-O O-O-O
```

These moves result in the following board states:


e2–e4

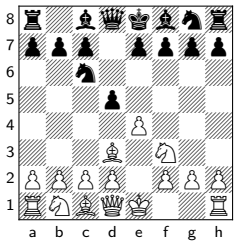

d7–d5

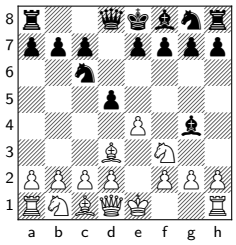

♗**f1–d3**

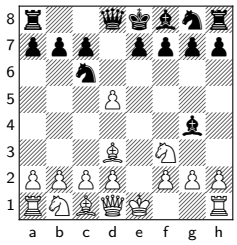

♘**b8–c6**

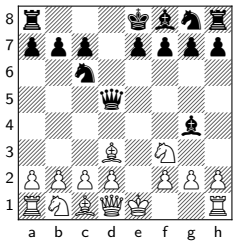
Invalid Input	Reason(s)
`e4 d5 i4`	there is no **i** file
`e4 d5 ed5`	a pawn moving files must capture, and the move must be written `exd5`
`e4 d5 exd5a5`	missing space between moves

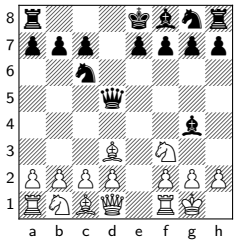
Table 1: Examples of syntactically invalid input for the program.

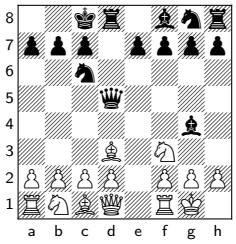












3.5 Promotion

A pawn which reaches the opposite side of the board promotes into a queen, rook, bishop, or knight. This is indicated by appending `=` followed by the type of the new piece to the move; for example, a pawn promoting to a queen on the **e** file would be recorded as `e8=Q`, and a pawn capturing from **b2** to **a1** and promoting to a knight would be recorded as `bxa1=K`.

3.6 Complete Move Sequences

A complete move sequence (i.e., the complete input to your program) consists of any number of moves, separated by one or more spaces each time. There can also be leading and trailing spaces. The input ends with a new line (indicated by a character which is either `'\n'` or `'\r'`), and the program can stop reading when it encounters such a character.

Example 3.7. We have already provided several examples of valid move sequences, and several extended examples are provided in Appendix C. Table 1 provides several examples of move sequences which are invalid due to syntax issues.

4 Output Format

The output from the program must be a single line in one of the following forms, in accordance with Section 2:

- `PLAYER wins by checkmate`, where `PLAYER` is either `black` or `white`
- `draw by stalemate`
- `game incomplete`

- `parse error at character 'X'`, where `X` is replaced by the problematic character
- `move completion error: PLAYER PIECE to SQUARE`, where:
  - `PLAYER` is either `black` or `white`
  - `PIECE` is one of `king`, `queen`, `rook`, `bishop`, `knight`, or `pawn`
  - `SQUARE` is the target coordinates, e.g., `e7`
  - In the case of a castle, consider the king to be the piece that is moving
- `illegal move: PLAYER PIECE from SQUARE to SQUARE`, similar to above

Examples 3.1 to 3.6 should all result in the output `game incomplete`. For examples of complete games, see Appendix C.

Example 4.1. Consider the input:

```
e4 d5 i4
```

This should result in the output:

```
parse error at character 'i'
```

Example 4.2. Consider the input:

```
e4 e5 exd5
```

This should result in the output:

```
move completion error: white pawn to d5
```

Example 4.3. Consider the input:

```
e4 d5 exd5 O-O
```

This should result in the output:

```
illegal move: black king from e8 to g8
```

If you are attempting the bonus, then in the case where the output is `game incomplete`, you must also print out a second line of output, of the following form, describing the suggested move:

```
suggest PLAYER PIECE from SQUARE to SQUARE
```

If the suggested move is a castle, follow the convention of describing the king's movement. If the suggested move is a pawn promotion, then use the following extended form:

```
suggest PLAYER pawn from SQUARE to SQUARE promoting to PIECE
```

5 Deliverables

5.1 Proposal

The *proposal* for the project is a short document which will outline how you intend to approach the problem, beyond what is specified in these instructions. It can be up to 2 pages long, and will be marked according to the rubric in Section A.1. It should be organized into the following sections, each of which should be very brief (around 1 paragraph).

1. **Introduction**
   - Summarize the scope of work for the project, and the scope of the proposal document.

- Assume the reader is a technical manager, i.e., someone with technical expertise, but who is not familiar with this project and its requirements.

2. **Board State Representation**
   - What are all the fields you will need to represent the board state?
   - What will their types by, and what will their *semantics* be (i.e., what will their values mean regarding the actual board state)?
   - In your representation, how will the initial state of the game be represented?
3. **Move Representation**
   - What are all the fields you will need to represent a move?
   - What will their types by, and what will their *semantics* be (i.e., what will their values mean regarding the actual board state)?
   - How will partial specification of the starting square be represented, and how will this look different for a move that has been completed?
   - How will special moves (i.e., castling and promotion) be represented?

5.2 Final Report

The final report is a longer document which acompanies your code and makes the case for why it correctly addresses the problem. It can be up to 15 pages long, and will be marked, alongside the code, according to the rubric in Section A.2. It should be organized into the following sections.

1. **Introduction**
   - Summarize the scope of work for the project, and the scope of the final report.
   - Assume the reader is a technical manager, i.e., someone with technical expertise, but who is not familiar with this project and its requirements.
2. **Board State Representation**
   - Explain the fields of `struct chess_board`.
   - Why did you choose the types that you did, and are the semantics of the field values?
   - Explain how `board_initialize` correctly initializes the board state.
3. **Move Representation**
   - Explain the fields of `struct chess_move`.
   - Why did you choose the types that you did, and are the semantics of the field values?
   - How is partial specification of the starting square represented, and how does this look different for a move that has been completed?
   - How are special moves (i.e., castling and promotion) represented?
4. **Input Parsing**
   - Explain the behaviour of `parse_move`.
   - How does it process the input text?
   - How does it ensure that invalid input is never incorrectly parsed as a valid move?
5. **Move Completion**
   - Explain the behaviour of `board_complete_move`.
   - How does it correctly infer the starting square of a move?

6. **Move Application**
    - Explain the behaviour of `board_apply_move`.
    - How does it check that a move is legal?
    - How does it update the board state?
7. **Board State Classification**
    - Explain the behaviour of `board_summarize`.
    - How does it check whether the player has legal moves?
    - How does it differentiate between checkmate and stalemate?
8. **Move Recommendation** (optional bonus)
    - Explain the behaviour of `board_recommend_move`.
    - How does it look for forced checkmate?
    - How does it deny an opponent's forced checkmate?
    - How does it make a strategic recommendation otherwise?

5.3 Code

The code that you submit should be based on the starter code. You should complete the code by completing the fields of `struct chess_move` and `struct chess_board` and the contents of the functions `parse_move()`, `board_initialize()`, `board_complete_move()`, `board_apply_move()`, and `board_summarize()`. You may also create additional functions within the files `board.c` and `parser.c` to be used in your code if you wish, but you should not modify the prototypes of the functions in the starter code or the contents of `main.c`; similarly, you may create additional user-defined types for use in the fields of `struct chess_move` and `struct chess_board` or the parameters or return types of your additional functions. Also, the only standard library functions which you are allowed to use are `getchar()`, `ungetchar()`, and `printf()`; see the week 7 content for information on how to use these.

6 Academic Integrity

Academic integrity means honestly and in good faith representing your work when completing and submitting an assessment; it is part of your professional responsibility as future engineers. A *departure from academic integrity* (DFAI) is the term used by the Senate Academic Integrity Procedures[2] to refer to an action you might take which breaches those requirements. In the context of this project, there are two main cases of DFAIs which I worry about:

1. **Plagiarism**, **Facilitation**, and **Unauthorized Collaboration**

    These are the classic types of DFAIs, which stem from people in some way sharing work beyond what is permitted by the rules of the assessment. Put simply: you are not allowed to collaborate, or share or receive materials related to the project, with anyone outside of your group (and the teaching team). You are allowed, and in fact encouraged, to collaborate *within your groups*; this is a group project after all!

    I, unfortunately, have a lot of experience prosecuting these types of DFAI; ask around if you don't believe me. Please don't make me use that experience.

2. **Unauthorized Content Generation** and **Unauthorized Use of Intellectual Property**

    This is where we talk about ChatGPT, Copilot, and the like.

[2] `https://www.queensu.ca/secretariat/policies/senate/academic-integrity-procedures-requirements-faculties-schools`

I am not stupid. I know that a lot of people use generative AI tools for a variety of purposes, including asking questions about concepts, generating study materials, and directly working on assessments. While I don't use them for any purposes personally, I recognize that some uses are reasonably justifiable; however, some are not. While different courses may have different policies, I want to draw a clear line around what is not legitimate for this project, and I and that policy to make sense to you in the context of what kind of real-world work this degree is supposed to be preparing you for.

This policy comes down to two rules which I am imposing:

(a) **You must come up with your own design.** You need to design the solution to the problem. If you manage to prompt-engineer your way to having an AI tool design your solution because the problem turns out to be simple enough for that to work, you've fully missed the point of the project. *Of course the problem is somewhat simple; this is a first-year first-semester course! We haven't taught you enough yet to fairly give you a more complicated problem!* Real-world engineering challenges will be much more complicated, involving orders of magnitude more interacting components than anything you will touch in your undergraduate degree, and the prevailing evidence suggests that, while AI tools may be able to help with some smaller tasks, they are not anywhere close to being able to address such large-system engineering problems. That's good news for you; that's what you'll get paid for in the future, if you work in industry as a practising engineer! However, there is no way you'll be able to solve the big, complicated problems if you're not able to solve the small, constrained problems from a first-year course. If you are unclear how to follow this rule in practice, look to the next rule.

In summary: you should *want* to know how to solve these types of problems yourself! This is one of the main capabilities expected from someone with the degree you signed up for, and if you're better at it, you will have a significant edge in looking for work! If your group is stuck, come talk to me at office hours or send me an email, and I'd be happy to provide some guidance; accordingly, plan your work schedule so that you don't leave something you might need to ask for help with until the last minute.

(b) **You cannot share my materials or the details of the project anywhere, including uploading them to AI tools.** That includes this document, the starter code with or without your modifications, and even your own paraphrased descriptions of the problem. This is actually an extremely common corporate information security policy that I've adapted to the academic context; the main reason for it in industry is that these AI tools are run and hosted by third parties, and so the company loses control of any information that their employees provide them. This could lead to the company's own trade secrets leaking to competitors, or it could create massive liability if it involves the trade secrets or other intellectual property of clients or partner companies. Respecting such boundaries, regardless how unlikely you may think it is that an actual loss will occur as a result, will be an important part of your future professional responsibilities as an engineer.

In order to enforce academic integrity, if I have reason to suspect that a DFAI has occurred, I may ask you to explain your work in the absence of any other resources, among other ways of gathering information as part of my investigation. I really do not enjoy that part of the job and do not want to do it, but it is part of my job and my responsibility as an instructor and I will do it if and when necessary.

In recognition of this policy, both your proposal and your final report must include the following sentence at the start, below your names:

> We, the authors, have read and understand the academic integrity policy provided in the project instructions, and we attest that all work we are submitting for this assessment adheres to it.

I hope these policies make sense to you. If you are unclear about the rules, or about my rationale for them, please feel free to reach out, and we can discuss it further.

Finally, consider that the university, as an institution with an accredited engineering program, has a responsibility to assess its students' abilities to apply their skills to open-ended design scenarios like the one

I present you with in this project. This take-home group project model for that kind of assessment requires a degree of trust in the students to approach it, honestly. If that trust turns out to be misplaced, the alternative is to try to assess that capability through proctored exams of some sort instead, which would probably be worse for everyone involved but much easier to enforce academic integrity in. Instructors routinely compare notes on these types of assessments when deciding how to run future ones, and there is a strong desire in some cases to push more toward proctored assessments in the face of increasing inappropriate use of AI tools; even in cases where it isn't enough evidence to prosecute a DFAI, we can usually tell when a submission went beyond the permitted use of AI. Because of past issues of this sort in programming courses, people well above my pay grade have specifically expressed interest in how things turn out in this respect for APSC 143 this year. I suggest that you think about the number of courses and assessments you likely have left in your degree and what kind of evidence you want this course to contribute to those decisions.

Appendix A Rubrics

A.1 Proposal

This document will be updated by Monday, October 27, with a rubric for the proposal.

A.2 Final Submission

This document will be updated by Monday, October 27, with a rubric for the final report and code, and a separate rubric for the potential bonus marks.

Appendix B The Rules of Chess

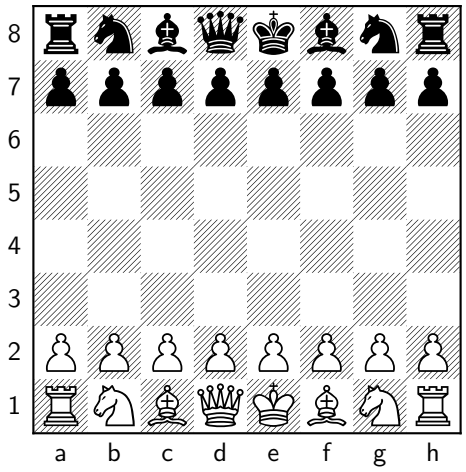
This appendix provides a brief summary of the rules of chess[3], for your reference as you work on the project. You are encouraged to look at examples throughout this document or at the wealth of further examples of chess games and resources freely available online[4] if you are unsure about the application of these rules.

B.1 The Board and Pieces

The board is an 8 by 8 grid of alternating light and dark squares, with two equivalent sets of pieces for two players; one set of pieces is the *white* pieces, and the other is the *black* pieces. In this document, the pieces are often denoted by small glyphs such as ♖, with the inside of the shape filled for black pieces and unfilled for white pieces. The available piece types are pawn ♙, knight ♘, bishop ♗, rook ♖, queen ♕, and king ♔. The initial state of the board is as follows.

[3] `https://handbook.fide.com/chapter/E012023`

[4] `https://en.wikipedia.org/wiki/Rules_of_chess`



The vertical columns of the board are called *files*, and the horizontal rows are called *ranks*. Each square is always either empty or occupied by a single piece.

B.2 Moves

The players, starting with the one using the white pieces, alternate making *moves*, in which they move one of their own pieces from one square to another. The move may result in *capturing* an opponent piece in the progress, in which case that opponent piece is removed from the board. The various types of moves are covered in the following subsections.

B.2.1 Pawn Moves

A pawn ♙ may only move *forward*, i.e., toward the opposite side of the board; ordinarily, it may only move one square forward, and the square it moves to must be empty. If it is the pawn's first move of the game, and the two squares immediately in front of it are empty, then it may move two squares forward in one move. Neither of these types of moves may capture a piece.

If a square immediately on a forward diagonal from a pawn is occupied by an opponent piece (i.e., one square forward and to the right or forward and to the left), then it may move to that square, capturing that opponent piece. Note that the pawn may not move to that square unless it is capturing.

In addition to the above means of capturing, a pawn may capture another pawn *en passant* (French for "in passing") as follows. If the opponent, in the most recent move, moved a pawn forward two squares, and our own pawn is on a square immediately to the left or right of the opponent pawn's destination square, then it may move diagonally to the empty square immediately behind the opponent pawn, capturing it in the process. See Example 3.3 for such a scenario.

Finally, if a pawn reaches the furthest rank from where it started, it is exchanged for a knight, bishop, rook, or queen of the same colour but a different type, called a *promotion*. The player chooses the type of piece to promote to, but it is not optional.

B.2.2 Regular Piece Moves

For every type of piece except a pawn, the regular legal moves are a regular pattern around the current square occupied by the piece. If there is an opponent piece on the destination square, then it is captured. The legal moves are as follows:

- A knight ♘ may move two squares horizontally and one square vertically, or vice versa, regardless of pieces in other squares.

- A bishop ♗ may move any distance along either diagonal (i.e., moving the same number of squares horizontally as vertically), as long as all squares directly between its starting and ending square are empty.
- A rook ♖ may move any distance vertically or horizontally (but not both in the same move), as long as all squares directly between its starting and ending square are empty.
- A queen ♕ may make any move that a bishop or rook may make.
- A king ♔ may move to any of the up to 8 immediately adjacent squares, horizontally, vertically, or on the diagonal.

B.2.3 Castling

Each player may, at any point in the game, *short castle* or *long castle* their king, which results in their king moving to the **g** file (short) or **c** file (long), remaining on its starting rank, if the following conditions all hold:

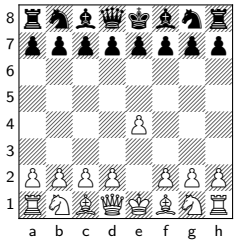
- The player has not moved that king in any previous move in the game.
- The player has not moved the rook toward which the king is moving in any previous move in the game.
- All squares between the king and rook involved are empty.

As a result of the move, the king moves as described, and the rook moves to the square between the previous and new squares of the king, all as part of one move. Castling never results in capturing a piece. See Example 3.6 for an example scenario involving both types of castling.

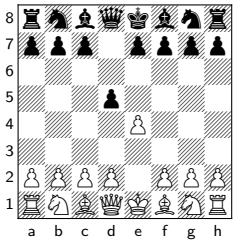
B.3 Check

A player is considered to be *in check* if the other player would be able to capture their king, if it was their turn.

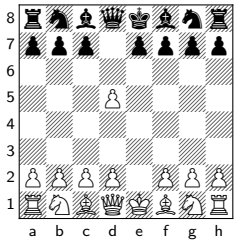
Example B.1. The following sequence of moves leaves white in check.



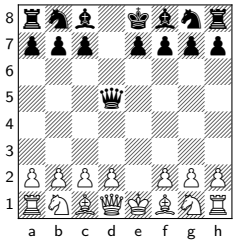
e2–e4



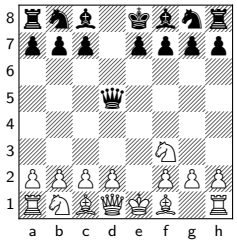
d7–d5



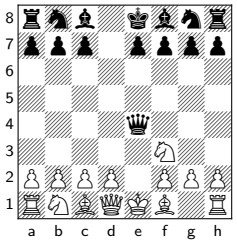
e4×d5



♕d8×d5



♘g1–f3



♕d5–e4+

A move is not legal if, in the resulting board state, the player is in check; this means that a player cannot put themselves in check, and if put in check by their opponent, must play a move which takes them out of check.

B.4 End of the Game

The game ends when it is a player's turn to make a move, but they do not have any legal moves to make. If that player is in check, then they are said to be in *checkmate*, and thus their opponent has won the game by checkmate. If they are not in check, then the game is in *stalemate* and is a draw. See Appendix C for examples of checkmates and stalemates.

Remark. The official rules of chess include several other ways for a game to end, such as repeating the same position 3 times; however, for the purposes of this project, we will only consider a game to be over if it is in a state of checkmate or stalemate.

Appendix C Example Input

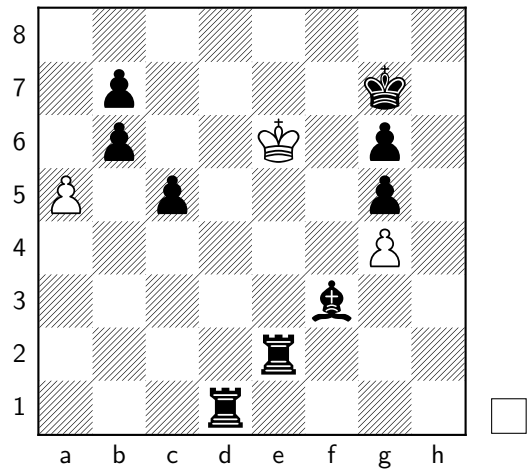
Example C.1. Consider the following input:

```
d4 Nf6 Bf4 g6 e3 Bg7 Bd3 d5 Nd2 c6 c3 Qb6 Qb3 Nbd7 Ngf3 Nh5 Qxb6 axb6 h3 Nxf4 exf4 Nf6 a3
↪  O-O O-O Nh5 g3 Bxh3 Rfe1 e6 c4 Bg4 cxd5 exd5 Ne5 Bxe5 dxe5 c5 f3 Bd7 g4 Nxf4 Bc2 Bb5
↪  Nb1 Rfe8 Nc3 Ba6 Ba4 Re7 Rad1 d4 Ne4 Kg7 Nd6 Nd3 Re2 Nxe5 Rf2 Nd3 Rg2 Nf4 Rh2 Ne2 Kg2
↪  Nf4 Kg3 Nd5 Rdh1 Rh8 Bc2 Ne3 Kf4 Nxc2 Rxc2 Re2 Rcc1 Rxb2 Ne4 Rd8 Ng3 d3 Ne4 d2 Rcd1
↪  Rd4 Ke5 Be2 Rxh7 Kxh7 Ng5 Kg7 Rh1 f6 Ke6 fxg5 a4 Bxf3 Rg1 d1=Q Rxd1 Rxd1 a5 Re2
```

The program should produce the following output:

```
black wins by checkmate
```

The final board state is as follows.



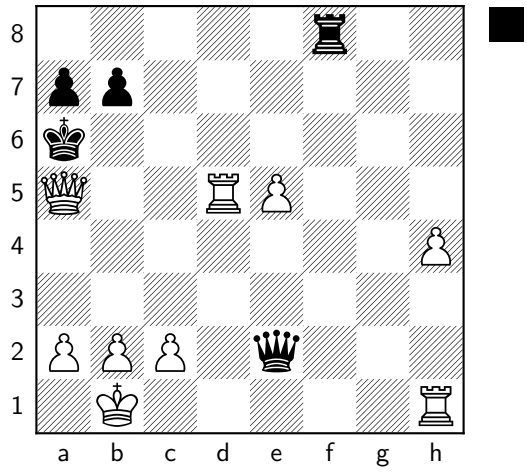
Example C.2. Consider the following input:

```
e4 c6 d4 d5 e5 f6 f4 Nh6 Nc3 Bg4 Be2 Bxe2 Qxe2 fxe5 fxe5 g6 Bg5 Qb6 O-O-O Nf5 Nf3 h6 g4
↪  hxg5 gxf5 g4 fxg6 gxf3 Qxf3 Rg8 Qf7 Kd7 Qxg8 Na6 Qf7 Bh6 Kb1 Nc7 g7 Bxg7 Qxg7 Ne6 Qg4
↪  Rf8 Ne2 c5 dxc5 Qxc5 h4 Qe3 Rxd5 Kc7 Qxe6 Qxe2 Qxe7 Kb6 Qc5 Ka6 Qa5
```

The program should produce the following output:

```
white wins by checkmate
```

The final board state is as follows.



Example C.3. Consider the following input:

```
e4 d6 d4 Nf6 Nf3 Nxe4 Bd3 Nf6 O-O g6 c4 Bg7 Bg5 O-O Nc3 h6 Bxf6 Bxf6 Be4 Bg7 Qb3 Nc6 d5
↪  Ne5 Nxe5 Bxe5 Rad1 f5 Bf3 Qe8 Na4 h5 c5 g5 h3 g4 hxg4 hxg4 Be2 Qh5 g3 f4 Rd3 f3 Bd1
↪  Qh3 Bxf3 gxf3 Rxf3 Rxf3 Qxf3 Bg4 Qb3 Rf8 Re1 Bf3 Qxf3 Rxf3 Re3 Rxe3 fxe3 Qxg3 Kf1
↪  Qxe3 cxd6 exd6 Nc3 Bxc3 bxc3 Qxc3 Ke2 Qb2 Kd3 Qxa2 Ke4 Kf7 Kf5 Qc4 Kg5 Qe4 Kh5 Qf4
```

The program should produce the following output:

```
draw by stalemate
```

The final board state is as follows.