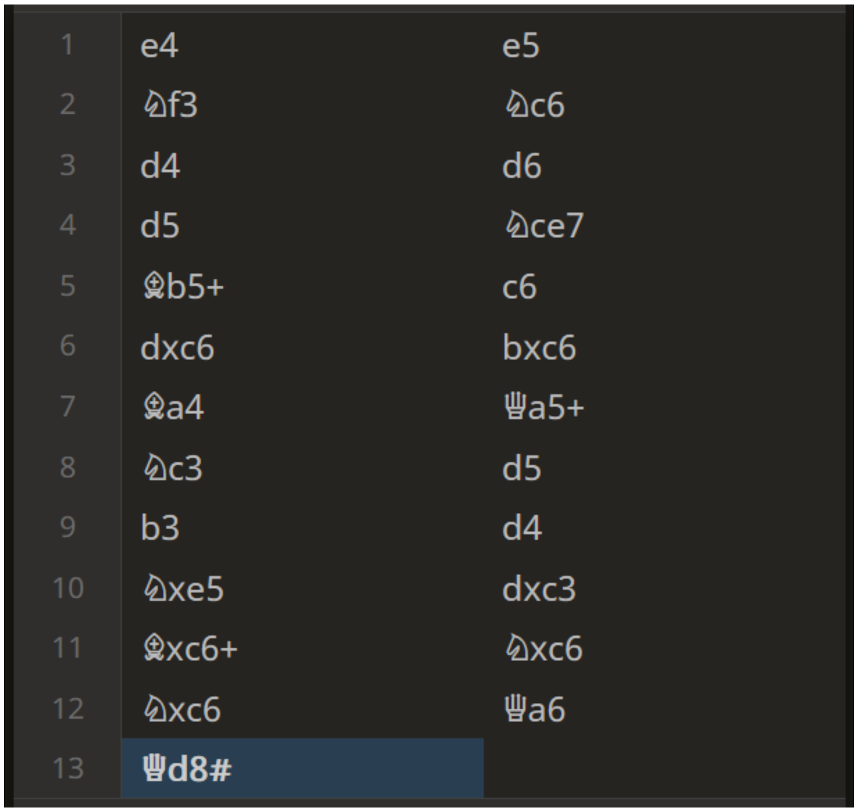
Wall Chess/Simplified Algebraic Chess Notation

CSCI 4890

## Introduction (General Description)

Chess can be played on a wall with a set like this:

These are examples of strings you'd find in a game:



| | | |
|---|---|---|
| 1 | e4 | e5 |
| 2 | ♘f3 | ♘c6 |
| 3 | d4 | d6 |
| 4 | d5 | ♘ce7 |
| 5 | ♗b5+ | c6 |
| 6 | dxc6 | bxc6 |
| 7 | ♗a4 | ♕a5+ |
| 8 | ♘c3 | d5 |
| 9 | b3 | d4 |
| 10 | ♘xe5 | dxc3 |
| 11 | ♗xc6+ | ♘xc6 |
| 12 | ♘xc6 | ♕a6 |
| 13 | ♕d8# | |

A device that could handle the notation on the wall accurately would be a lovely convenience. This is because it is straight forward to transfer over the board games to a chess engine and see which moves were good/bad/in-between.

I thought about this problem and realized that the files, ranks, and pieces, which take the form

files = {a,b,c,d,e,f,g,h}

ranks = {1,2,3,4,5,6,7,8}

pieces = {♔,♕,♖,♗,♘,♚,♛,♜,♝,♞}

Could be generalized as

{F, r, P}.

In terms of Computer Science, we aren't really worried about Pf3 or ♘f3. I am calling this observation/design choice "**Simplified Algebraic Chess Notation**". It is the same as the long established Algebraic Notation (see: https://en.wikipedia.org/wiki/Algebraic_notation_(chess)) except that alphabet members {F,r,P} represent any file, rank, or piece choice.

The machine will accept valid move strings, but not validate said moves. Proving move legality requires things like bitboards and again, we are interested in a "note-taking" device, not a "chess machine".

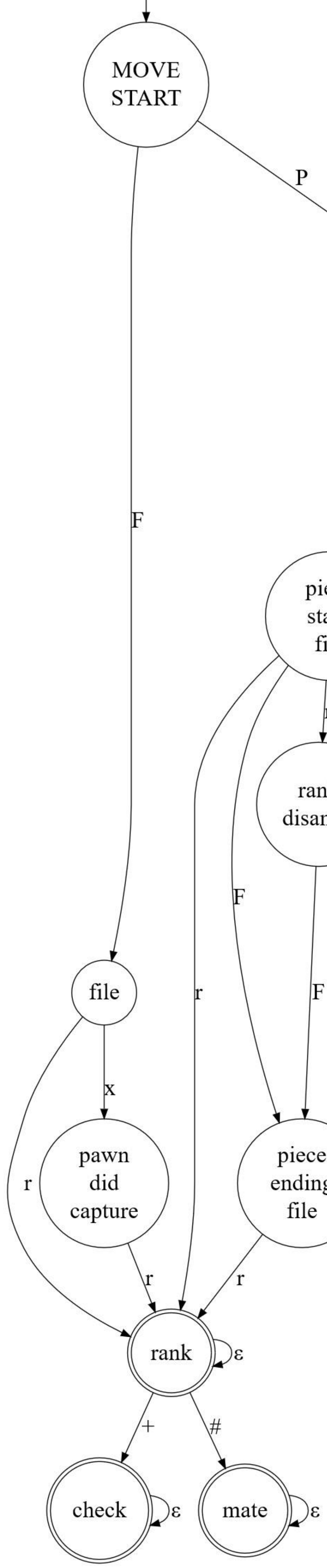
With this Simplified Algebraic Chess Notation notion, we will start with an NFA and convert it to:

- **Regular Language/Expression**
- **Context-Free Grammar**
- **Pushdown Automata**

Near the conclusion, we will take a look at **CFG with the full algebraic notation.**

**NFA**

On the next page, we will look at the NFA and explain the Alphabet + define the Tuples.


MOVE START
P
piece
F
r
piece start. rank
F
piece start. file
x
F
r
x
rank disamb.
x
x
x
file
r
F
F
piece did capture
F
x
r
pawn did capture
piece ending file
r
r
rank
ε
+
#
check
ε
mate
ε

From **Definition 1.37** of the Sipser Text:

Q = {MOVE START, piece, file, file disamb., rank disamb., piece did capture, piece ending file, pawn did capture, rank, mate, check}

Σ = {P,F,x,r,+,#,ε}

δ =

| | P | F | x | r | + | # | ε |
|---|---|---|---|---|---|---|---|
| MOVE START | piece | file | ∅ | ∅ | ∅ | ∅ | ∅ |
| piece | ∅ | piece start. file | piece did capture | piece start. rank | ∅ | ∅ | ∅ |
| file | ∅ | ∅ | pawn did capture | rank | ∅ | ∅ | ∅ |
| piece start. rank | ∅ | piece start. file | piece did capture | ∅ | ∅ | ∅ | ∅ |
| piece start. file | ∅ | piece ending file | piece did capture | {rank, rank disamb.} | ∅ | ∅ | ∅ |
| rank disamb. | ∅ | piece ending file | piece did capture | ∅ | ∅ | ∅ | ∅ |
| piece did capture | ∅ | piece ending file | ∅ | ∅ | ∅ | ∅ | ∅ |
| piece ending file | ∅ | ∅ | ∅ | rank | ∅ | ∅ | ∅ |
| pawn did capture | ∅ | ∅ | ∅ | rank | ∅ | ∅ | ∅ |
| rank | ∅ | ∅ | ∅ | ∅ | check | mate | rank |
| mate | ∅ | ∅ | ∅ | ∅ | ∅ | ∅ | mate |
| check | ∅ | ∅ | ∅ | ∅ | ∅ | ∅ | check |

q0 = MOVE START

F = {rank, check, mate}

Semi-formally speaking: The **exact goal of this machine** is that it should accept the strings within the set:

S = {Fr, Fr+, Fr#, Fxr, Fxr+, Fxr#, PFr, PFr+, PFr#, PxFr, PxFr+, PxFr#, PFFr, PFFr+, PFFr#, PFxFr, PFxFr+, PFxFr#, PrFr, PrFr+, PrFr#, PrxFr, PrxFr+, PrxFr#, PFrFr, PFrFr+, PFrFr#, PFrxFr, PFrxFr+, PFrxFr#}

And **no other** strings.

**Regular Expression/Language**

The incoming chart was constructed from the first prototype of this project, a regex. But that regex was constructed before the notion of Algebraic Chess Notation → Simplified Algebraic Chess Notation. Once the Simplified notion was established, it was observed that elegant Regular Expressions (here, not Regexes per se) were straightforward to construct:

| | | | | | |
|---|---|---|---|---|---|
| **Fr** | **Fxr** | **PFFr** | **PFxFr** | **PFrFr** | **PFrxFr** |
| **Fr+** | **Fxr+** | **PFFr+** | **PFxFr+** | **PFrFr+** | **PFrxFr+** |
| **Fr#** | **Fxr#** | **PFFr#** | **PFxFr#** | **PFrFr#** | **PFrxFr#** |
| **One File, One Rank, pawn** | | **Two Files, One Rank** | | **Two Files, Two Ranks** | |
| **"Pawn Move"** | | **"File Disamb.Piece Move"** | | **"File & Rank Disamb.Piece Move"** | |

| | | | |
|---|---|---|---|
| **PFr** | **PxFr** | **PrFr** | **PrxFr** |
| **PFr+** | **PxFr+** | **PrFr+** | **PrxFr+** |
| **PFr#** | **PxFr#** | **PrFr#** | **PrxFr#** |
| **One File, One Rank (piece)** | | **One File, Two Ranks** | |
| **"Piece Move"** | | **"Rank Disamb. Piece Move"** | |

A Regular Expression was made for each case of the moves, and a notion that each case of the moves could be its own machine was established. (Additional context is available on the Appendix). This greatly simplified understanding, and results in these Regular Expressions:

| | | |
|---|---|---|
| F[x]r[+\|#] | PF{2}r[+\|#] U PFxFr[+\|#] | PFr[x]Fr[+\|#] |
| PF[x]r[+\|#] | Pr[x]Fr[+\|#] | |

The Regular Expressions Unified:

$$F[x]r[+|\#]\cup$$

$$PF[x]r[+|\#]\cup$$

$$PFFr[+|\#]\cup$$

$$PFxFr[+|\#]\cup$$

$$Pr[x]Fr[+|\#]\cup$$

$$PFr[x]Fr[+|\#]$$

The conversion of NFA to Regular Expression CFG led to insights that fueled the following refactors to the overall project logic:

- Let the starting variable be considered a kind of central hub where each of the Machines/Cases/Moves could be chosen.

- The capture symbol and epsilon were interchangeable in a sense (of course in the game they are much different, but in terms of constructing valid strings they are interchangeable) so a rule `X -> ϵ | x` would be really convenient.
- For LaTex purposes, the Moves could have trivial `Q_n -> whatever_move` rules.

With these ideas, we can construct the following Grammar:

**Context Free Grammar**

$$
\begin{aligned}
MOVESTART &\rightarrow Q_1 \cup Q_2 \cup Q_3 \cup Q_4 \cup Q_5 \\
Q_1 &\rightarrow \text{Pawn Move} \\
Q_2 &\rightarrow \text{Piece Move} \\
Q_3 &\rightarrow \text{File Disamb. Piece Move} \\
Q_4 &\rightarrow \text{Rank Disamb. Piece Move} \\
Q_5 &\rightarrow \text{Doubly Disamb. Piece Move} \\
\text{Pawn Move} &\rightarrow FXr[+|\#] \\
\text{Piece Move} &\rightarrow PFXr[+|\#] \\
\text{File Disamb. Piece Move} &\rightarrow PFFr[+|\#] \\
\text{Rank Disamb. Piece Move} &\rightarrow PrXFr[+|\#] \\
\text{Doubly Disamb. Piece Move} &\rightarrow PFrXFr[+|\#] \\
X &\rightarrow \epsilon \,|\, x
\end{aligned}
$$

**Pushdown Automata**

The PA is great for this problem, because the Stack allows far more flexibility among the states. This is shines in the End Rank state, which can now “process” the final rank & file. In the NFA, that last file had to be wrangled very precisely across numerous states, and resulted in a brittle abstraction. The NFA was redone multiple times and was error prone although constructing it was still a crucial exercise.

From **Definition 2.13** of the Sipser text:

Q = {Move Start, Pawn Move, End Rank, Check or Mate, Piece, Piece file, Rank Disamb., Piece Capture}

Σ = {P,F,x,r,+,#,ε}

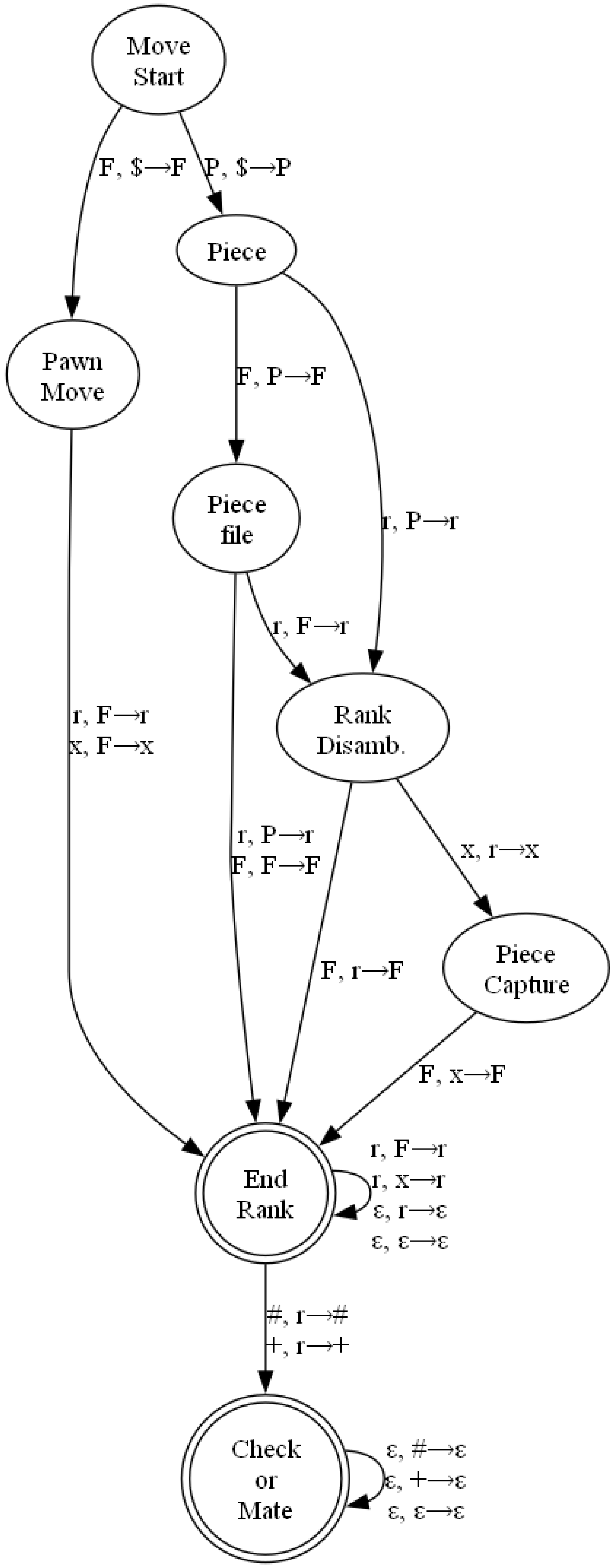
Γ = {$,P,F,x,r,+,#,ε}

$q_0$ = Move Start

F = {End Rank, Check or Mate}

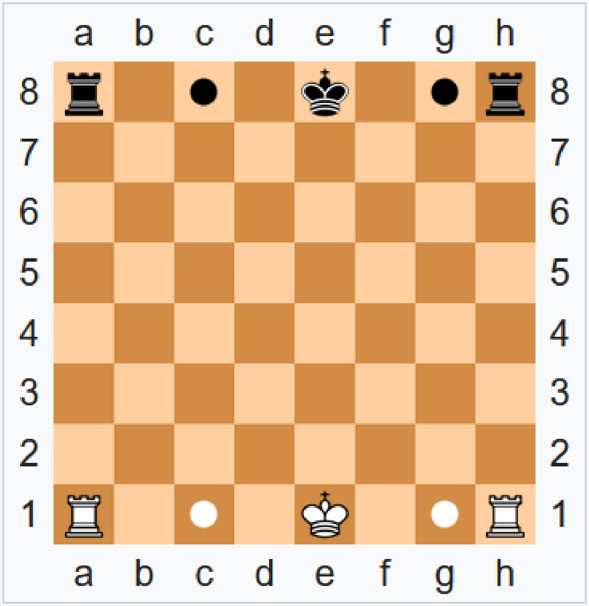
δ will be expressed with a Python implementation and resultant diagram:

```
from graphviz import Digraph

dot = Digraph()
q0 = ‘Move\nStart’
q1 = ‘Pawn\nMove’
q2 = ‘End\nRank’
q3 = ‘Check\nor\nMate’
q4 = ‘Piece’
q5 = ‘Piece\nfile’
q6 = ‘Rank\nDisamb.’
Q7 = ‘Piece\nCapture’
dot.node(q0, q0)
dot.node(q1, q1)
dot.node(q2, q2, shape=’doublecircle’)
dot.node(q3, q3, shape=’doublecircle’)
dot.node(q4, q4)
dot.node(q5, q5)
dot.node(q6, q6)
dot.node(q7, q7)

dot.edge(q0, q1, label=’F, $→F’)
dot.edge(q1, q2, label=’r, F→r\nx, F→x’)
dot.edge(q2, q2, label=’r, F→r\nr, x→r\nε, r→ε\nε, ε→ε’)
dot.edge(q2, q3, label=’#, r→#\n+, r→+’)
dot.edge(q3, q3, label=’ε, #→ε\nε, +→ε\nε, ε→ε’)

dot.edge(q0, q4, label=’P, $→P’)
dot.edge(q4, q5, label=’F, P→F’)
dot.edge(q5, q2, label=’r, P→r\Nf, F→F’)
dot.edge(q4, q6, label=’r, P→r’)
dot.edge(q5, q6, label=’r, F→r’)
dot.edge(q6, q2, label=’F, r→F’)
dot.edge(q6, q7, label=’x, r→x’)
dot.edge(q7, q2, label=’F, x→F’)

dot.render(‘pda_diagram’, format=’png’)
```

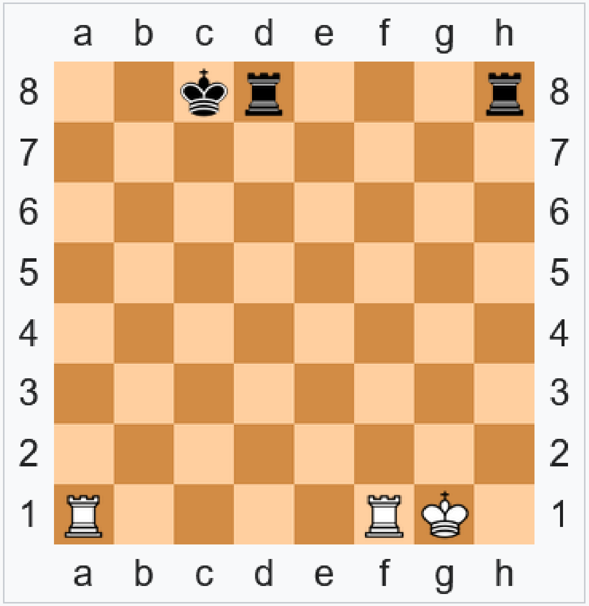

Move Start
F, $⟶F
P, $⟶P
Piece
Pawn Move
F, P⟶F
Piece file
r, P⟶r
r, F⟶r
Rank Disamb.
r, F⟶r
x, F⟶x
r, P⟶r
F, F⟶F
x, r⟶x
F, r⟶F
Piece Capture
F, x⟶F
End Rank
r, F⟶r
r, x⟶r
ε, r⟶ε
ε, ε⟶ε
#, r⟶#
+, r⟶+
Check or Mate
ε, #⟶ε
ε, +⟶ε
ε, ε⟶ε

**Caveats**

For transparency's sake, there are two moves our machine doesn't cover. Castling:

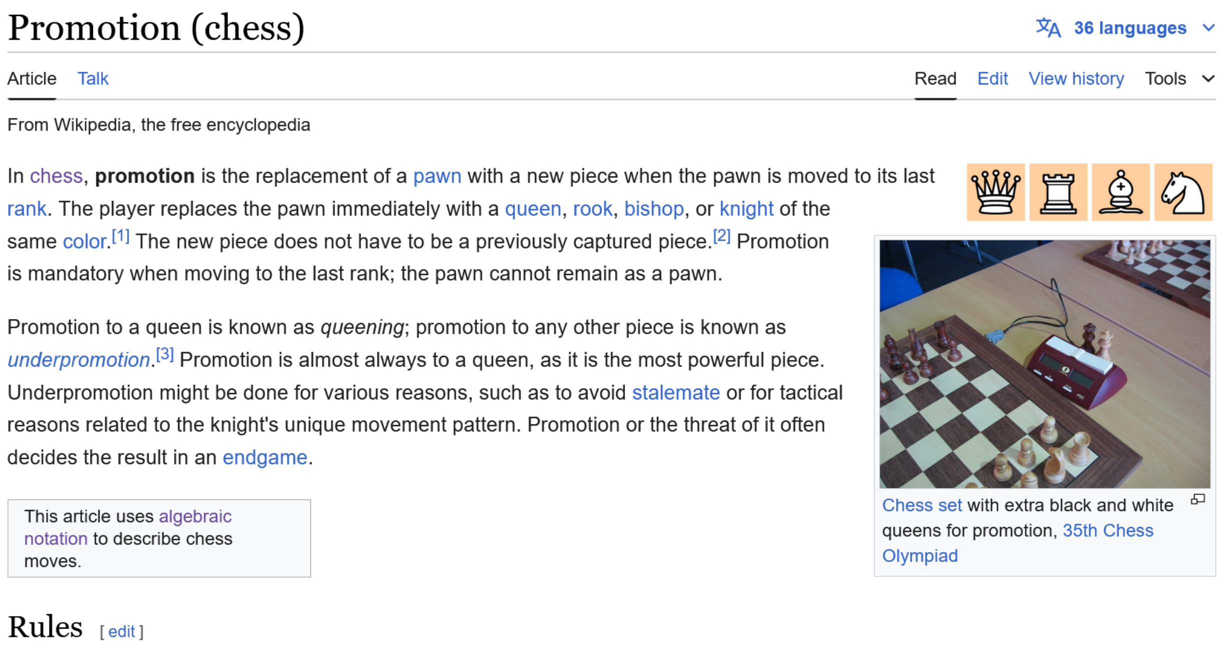


Initial positions of kings and rooks. Kings may castle to the indicated squares.



White has castled kingside; Black has castled queenside.

And promotion:



Promotion (chess)

36 languages

Article Talk Read Edit View history Tools

From Wikipedia, the free encyclopedia

In chess, **promotion** is the replacement of a pawn with a new piece when the pawn is moved to its last rank. The player replaces the pawn immediately with a queen, rook, bishop, or knight of the same color.[1] The new piece does not have to be a previously captured piece.[2] Promotion is mandatory when moving to the last rank; the pawn cannot remain as a pawn.

Promotion to a queen is known as *queening*; promotion to any other piece is known as *underpromotion*.[3] Promotion is almost always to a queen, as it is the most powerful piece. Underpromotion might be done for various reasons, such as to avoid stalemate or for tactical reasons related to the knight's unique movement pattern. Promotion or the threat of it often decides the result in an endgame.

This article uses algebraic notation to describe chess moves.

Chess set with extra black and white queens for promotion, 35th Chess Olympiad

Rules [ edit ]

Castling was skipped because it requires logic derived from processing the entire board/game state. A Turing Machine would be required.

- There must be no pieces between the king and rook.
- There must be no square that is under attack (ex. Castling can't put the king in check, even if the square it is moving to isn't the final square it will end up at)
- Neither the king nor the rook must have moved even one square (moving back to their original squares still invalidates eligibility to castle)

Promotion was skipped to simplify the presentation of the project, but technically the only logic you need to confirm it is valid (assuming the move is legal of course) is to check if a white pawn is moving to Rank 8 or a black pawn is moving to Rank 1. A Simplified Algebraic Chess Notation PA is incapable of this, but an Algebraic Chess Notation PA could easily do this.

**Grammar implementing all moves (no validation for Castling and Promotion)**

$$\begin{aligned}
MOVE\,START &\rightarrow Q_1 \cup Q_2 \cup Q_3 \cup Q_4 \cup Q_5 \cup Q_6 \cup Q_7 \cup Q_8 \\
Q_1 &\rightarrow \text{Pawn Move} \\
Q_2 &\rightarrow \text{Piece Move} \\
Q_3 &\rightarrow \text{File Disamb. Piece Move} \\
Q_4 &\rightarrow \text{Rank Disamb. Piece Move} \\
Q_5 &\rightarrow \text{Doubly Disamb. Piece Move} \\
Q_6 &\rightarrow \text{Promotion} \\
Q_7 &\rightarrow \text{Kingside Castle} \\
Q_8 &\rightarrow \text{Queenside Castle} \\
\text{Pawn Move} &\rightarrow FXr[+|\#] \\
\text{Piece Move} &\rightarrow PFXr[+|\#] \\
\text{File Disamb. Piece Move} &\rightarrow PFFr[+|\#] \\
\text{Rank Disamb. Piece Move} &\rightarrow PrXFr[+|\#] \\
\text{Doubly Disamb. Piece Move} &\rightarrow PFrXFr[+|\#] \\
\text{Promotion} &\rightarrow FR_P[+|\#] \\
\text{Kingside Castle} &\rightarrow \text{O-O} \\
\text{Queenside Castle} &\rightarrow \text{O-O-O} \\
X &\rightarrow \epsilon \mid x \\
F &\rightarrow a|b|c|d|e|f|g \\
r &\rightarrow 1|2|3|4|5|6|7|8 \\
P &\rightarrow \text{♖}|\text{♕}|\text{♗}|\text{♘}|\text{♔}|\text{♜}|\text{♛}|\text{♝}|\text{♞}|\text{♚} \\
R_P &\rightarrow 1|8
\end{aligned}$$

**Conclusion**

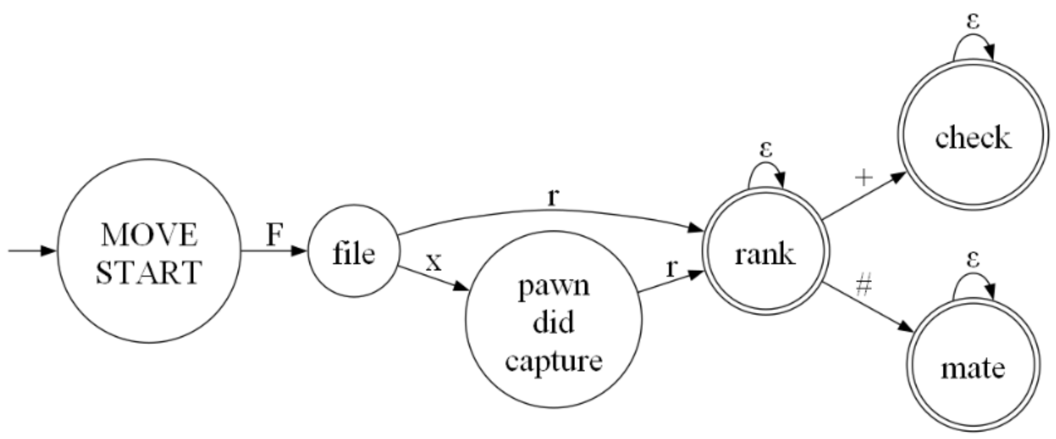
The project started out as an excuse to eventually use a Raspberry Pi Pico and have a formally proven system to use, but it ended up becoming the crux of my CSCI 4890 study, and I ended becoming much more interested in the process of converting from NFA→RE→CFG→PA. Because I was able to heavily rely on my knowledge of the moves, I was able to spot bugs in implementation, making it much easier to make correct examples of the relevant abstractions (Grammars, Expressions, the PA).

**Appendix**

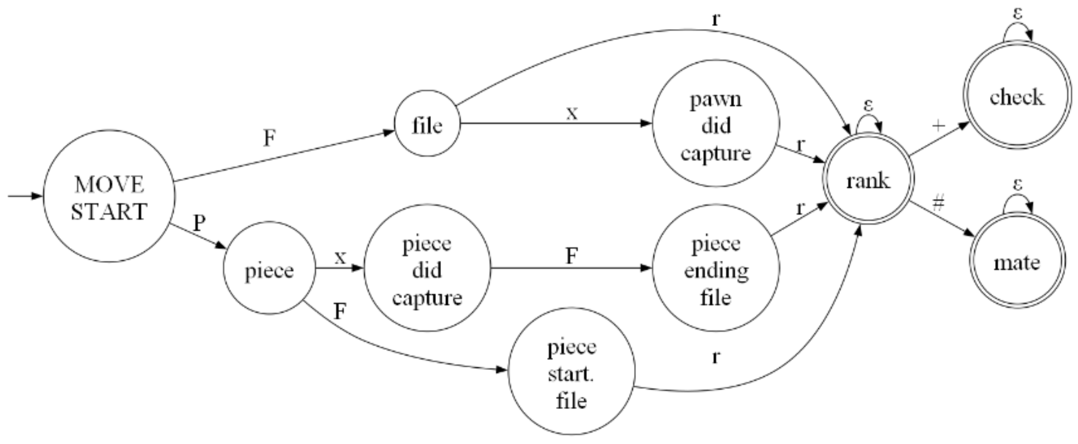
The notion of green, light green, yellow, red, and dark_red machine is explained in this code: https://github.com/IntegralWorks/CSCI_4890_Wall_Chess . Specifically, within https://github.com/IntegralWorks/CSCI_4890_Wall_Chess/blob/main/view_machines.ipynb one can see that each machine can be combined via OOP Inheritance. Ex.:

```
from automathon import NFA
from PIL import Image
from PIL import ImageDraw
from PIL import ImageFont
from PIL import ImageOps

s = ''
s+='|------------------------------------------------------------------------|\n'
s+='|                                 Legend                                 |\n'
s+='|F: File (a|b|c|d|e|f|g|h) r: Rank (1|2|3|4|5|6|7|8) P: Piece (R|Q|N|B|K)|\n'
s+='|x: Capture (x)            +: Check (+)              #: Mate (#)         |\n'
s+='|------------------------------------------------------------------------|\n'

#univeral to all machines
piece    = 'P'
file     = 'F'
capture  = 'x'
rank     = 'r'
check    = '+'
mate     = '#'
epsilon  = 'ε'
alphabet = {piece, file, capture, rank, check, mate, epsilon}

class M:
    def __init__(self):
        self.q  = {'MOVE\nSTART'} #always states
        self.q0 = 'MOVE\nSTART'   #always states
        self.f  = {'check','mate'} #check and mate are always ending characters
        self.sigma = alphabet
        self.legend = s
        self.delta = dict()
        self.automata = None
    def add_states_to_q(self, lst : list):
        for qn in lst:
            self.q.add(qn)

    def update_transistions_to_delta(self, qn : str, transistions : dict): #transistions must
be a dict of key:value is str:set
        if qn not in self.delta.keys():
            self.delta[qn] = transistions

        if qn in self.delta.keys():
            for k,v in transistions.items():
                    self.delta[qn][k] = v

    #pawn only moves
    def green_machine(self):
        self.add_states_to_q(['file','pawn\ndid\ncapture','rank','check','mate'])
        self.f.add('rank')
        self.update_transistions_to_delta('MOVE\nSTART'      , {file  : {'file'}})
        self.update_transistions_to_delta('file'             , {rank  : {'rank'} , capture :
{'pawn\ndid\ncapture'}})
        self.update_transistions_to_delta('pawn\ndid\ncapture', {rank  : {'rank'}})
        self.update_transistions_to_delta('rank'             , {check : {'check'}, mate    :
{'mate'}, epsilon : {'rank'}})
        self.update_transistions_to_delta('check'            , {epsilon : {'check'}})
        self.update_transistions_to_delta('mate'             , {epsilon : {'mate'}})


    #identical to pawn only moves, but with a piece
    def light_green_machine(self):
        self.green_machine()
        self.add_states_to_q(['piece', 'piece\nstart.\nfile', 'piece\ndid\ncapture'    ,
'piece\nending\nfile'])
        self.update_transistions_to_delta('MOVE\nSTART'        , {piece : {'piece'}})
        self.update_transistions_to_delta('piece'              , {file  :
{'piece\nstart.\nfile'}, capture : {'piece\ndid\ncapture'}})
        self.update_transistions_to_delta('piece\ndid\ncapture' , {file  :
{'piece\nending\nfile'}})
        self.update_transistions_to_delta('piece\nstart.\nfile' , {rank  : {'rank'}})
        self.update_transistions_to_delta('piece\nending\nfile' , {rank  : {'rank'}})
```

```
In [1]: import machine_definitions
        from IPython.display import Image as Show

In [2]: m = machine_definitions.M()

In [3]: m.green_machine()

In [4]: m.process_automata()

        Validity: True
        Rejected:  PFr PFr+ PFr# PxFr PxFr+ PxFr# PFFr PFFr+ PFFr# PFxFr PFxFr+ PFxFr# PrFr PrFr+ PrFr# PrxFr PrxFr+ PrxFr# PFrFr PFrFr+
        PFrFr# PFrxFr PFrxFr+ PFrxFr#

In [5]: Show(filename='automata.gv.png')

Out[5]:
```



```
In [6]: m.light_green_machine()
        m.process_automata('lightgreen')
        Show(filename='lightgreen.gv.png')

        Validity: True
        Rejected:  PFFr PFFr+ PFFr# PFxFr PFxFr+ PFxFr# PrFr PrFr+ PrFr# PrxFr PrxFr+ PrxFr# PFrFr PFrFr+ PFrFr# PFrxFr PFrxFr+ PFrxFr#

Out[6]:
```



```
In [7]: m.yellow_machine()
```

And so on.

Within https://github.com/IntegralWorks/CSCI_4890_Wall_Chess/tree/main, there are a couple of .tex files that were compiled with Overleaf. The chess symbols were found here: https://ctan.mirrors.hoobly.com/info/symbols/comprehensive/symbols-a4.pdf