DOCUMENTAZIONE TECNICA CHESSCOMPILER

Linguaggi Formali e Compilatori A.A. 2023-2024

ALESSANDRO COLOMBO 1066001

GIONATHA PIROLA 1066011

FILOSOFIA GENERALE DI SVILUPPO

ChessChecker è un compilatore di partite di scacchi che si basa sull'idea di poter controllare la correttezza ed in seguito visualizzare una partita di scacchi partendo da una notazione classica. In ambienti professionali, le mosse sono solitamente specificate seguendo una notazione ben precisa che non lascia alcuna ambiguità riguardo la mossa che viene eseguita. Questo tipo di scrittura viene usato per riassumere partite nei tornei o come linee guida in diversi libri, al fine di evitare di dover rappresentare graficamente tutta la scacchiera e per avere un linguaggio comune a tutti i giocatori.

I potenziali utilizzatori sono quindi due: nel primo caso, chi deve registrare le mosse si può servire di questo tool per verificare che siano scritte correttamente e che non siano presenti mosse non concesse da regolamento. Nel secondo caso, chiunque legga una partita può inserire le mosse descritte nel compilatore per vedere una rappresentazione grafica, più chiara e veloce, della situazione analizzata.

Questo richiederebbe ovviamente degli adattamenti da parte della *community globale* di scacchi, rendendo eventuali partite o istruzioni disponibili come file di testo: un libro potrebbe inserire qr code da scannerizzare per poter copiare il contenuto del file in pochi secondi, la pagina web di un torneo potrebbe permettere di scaricare il file già pronto per essere lanciato, correttamente compilato in fase di stesura.

TECNOLOGIE UTILIZZATE

Per lo sviluppo del progetto sono stati utilizzati i seguenti tool:

- **Eclipse** -> IDE di sviluppo;
- **Antlr** -> Generatore di parser;
- **Java** -> Linguaggio di sviluppo (jdk-17);
- **StarUML** -> diagrammi UML.

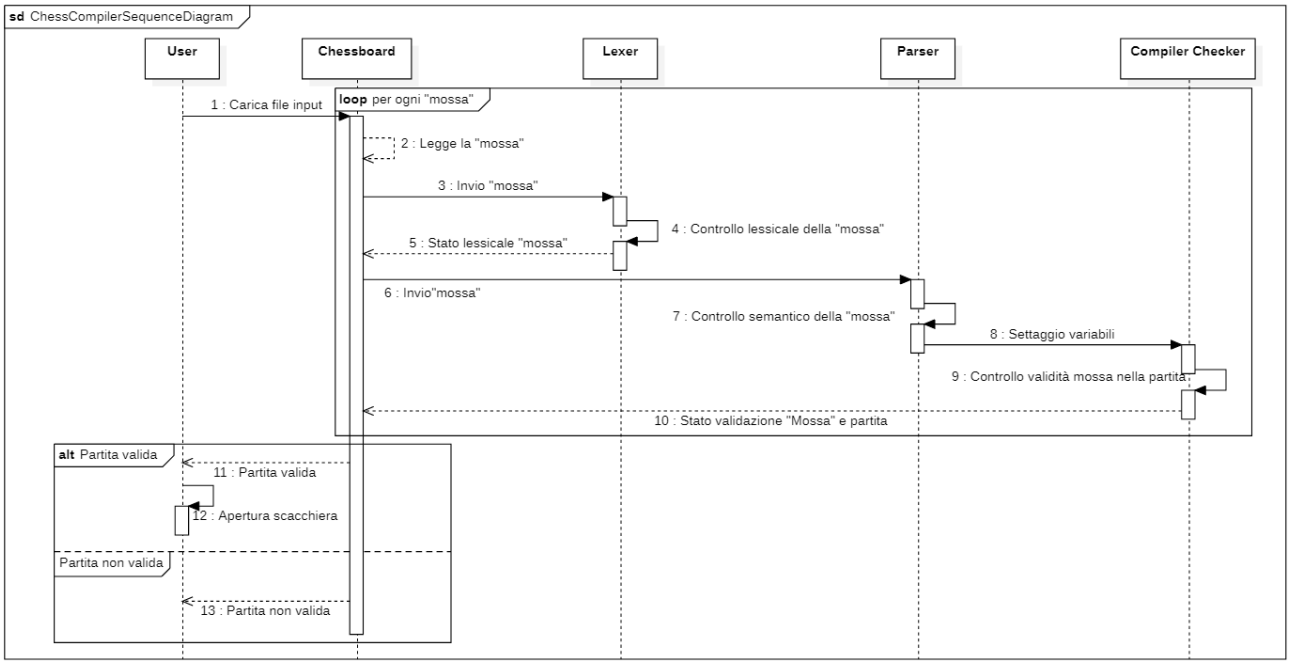
FUNZIONAMENTO GENERALE

Preso un input un file di mosse scacchistiche questo viene compilato e nel caso in cui non siano presenti errori (sia lessicali sia semantici) viene avviata una scacchiera grafica dove è possibile scorrere tra le mosse inserite.
Per far ciò è stato necessario sviluppare un **compilatore** (tramite ANTLR) per validare le mosse, un **semantichandler** per la gestione dei possibili errori presenti nel file di input ed infine una parte

puramente java per occuparsi dell'**interfaccia utente** del programma (sia per l'input che per l'output).

Nel diagramma che segue è possibile vedere il funzionamento generale del programma.

Sequence Diagram



STRUTTURA DEL PROGETTO

Tutto il codice si trova sotto la cartella **Code->src**, all'interno si trovano i seguenti package:

testerCompilerChecker

Qui si trovano le seguenti classi:

- **Main**: main class del progetto, ciò che viene effettivamente lanciato in ChessCompiler.jar;
- **Main_Tester**: main creato in fase di sviluppo per testare il funzionamento del progetto senza necessità di inserire il file di input da interfaccia grafica;
- **ChessGrammarLexerTester**: chiama il lexer creato da ANTLR stampando un output dei risultati ottenuti ed eventuali errori;
- **ChessGrammarParserTester**: chiama il parser creato da ANTLR stampando un output dei risultati ottenuti ed eventuali errori.

compilerPackage

All'interno si trovano tutte le classi relative alla parte di gestione del compilatore.
Nello specifico :

- **compilerChecker**: file per il controllo semantico della grammatica.
  Al suo interno sono presenti tutti i metodi chiamati dal compilatore per il controllo semantico delle istruzioni prese in input dal programma.
  Estende la classe Checker in quanto alcuni metodi utilizzati nel controllo semantico sono gli stessi utilizzati per capire il pezzo che deve muoversi;
- **semantichandler**: file per la gestione degli errori/warning della grammatica.
  Al suo interno sono presenti tutti gli errori/warning gestiti in fase di compilazione;
- **chessGrammar.g**: file utilizzato da ANTLR, la grammatica del programma;
- **chessGrammar.tokens**: file contenente i token generati da ANTLR;
- **chessGrammarLexer**: file java del lexer, generato da ANTLR;
- **chessGrammarParser**: file java del parser, generato da ANTLR;

chessPackage

Una volta che il compilatore termina senza errore vengono utilizzate le classi presenti in questo package per la gestione della parte grafica del progetto, in particolare:

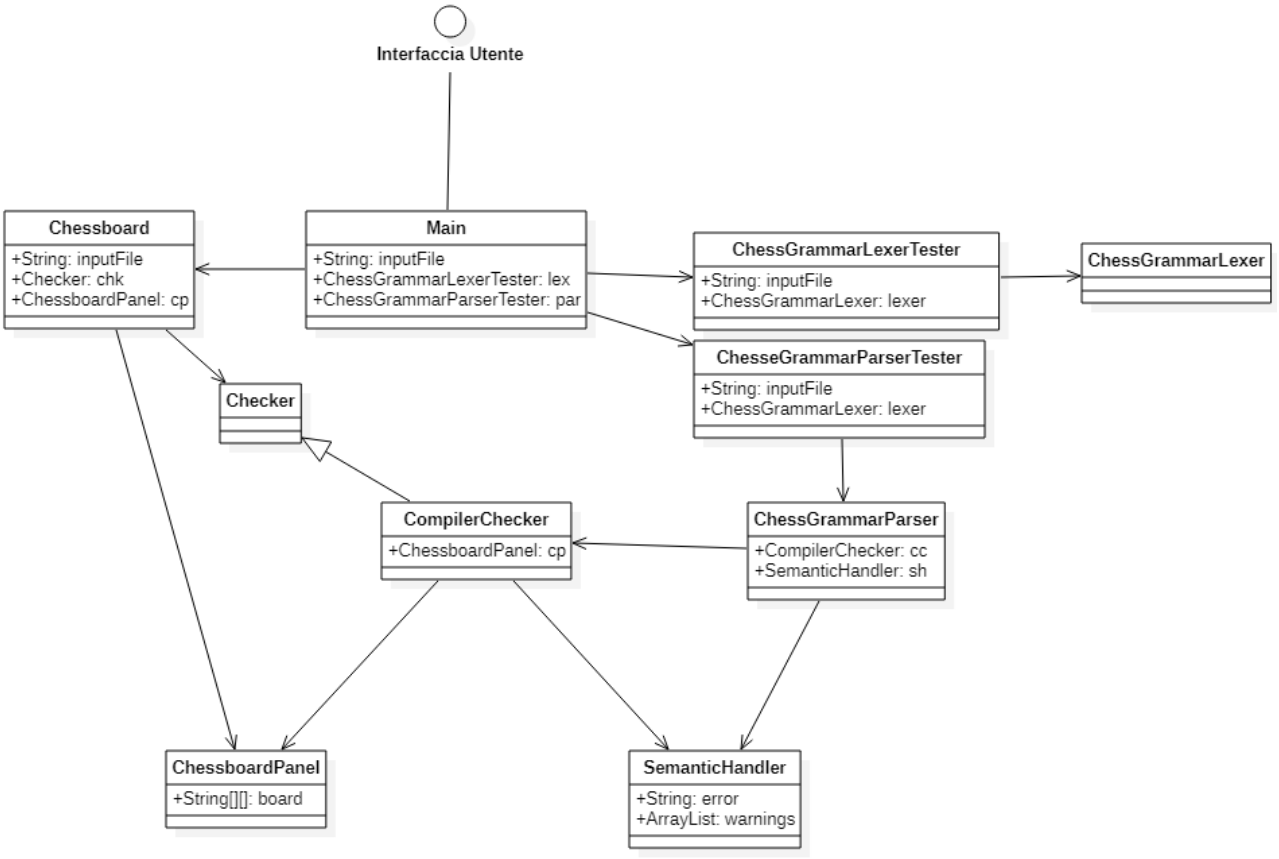
- **Checker**: classe utilizzata per capire quale pezzo dovrà muoversi data una determinata mossa (già validata dal compilatore).
  Una volta stabilite le modifiche da effettuare sulla scacchiera si occuperà del cambio;
- **Chessboard**: si occupa di leggere dal file di input le varie mosse passandole alla classe Checker;
- **ChessboardPanel**: classe che rappresenta la scacchiera.

pieces

All'interno si trovano semplicemente le immagini dei pezzi scacchistici utilizzati dal programma.

Nel diagramma che segue sono presenti le relazioni tra le classi descritte in precedenza.

Class Diagram



CONTROLLI SEMANTICI/ERRORI GESTITI

I controlli semantici effettuati sulla grammatica (presenti nel file compilerChecker) e quindi di conseguenza gli errori che il programma è in grado di gestire (tramite semanticHandler) sono i seguenti:

- **TAKE_NOT_CORRECT_ERROR**: nella mossa devono essere inseriti indicatori di mangiata (‘x’ o ‘:’) quando si è effettuata una mangiata.
  È gestito anche il caso in cui non viene effettuata una mangiata ma viene messo l’indicatore nella mossa;
- **MOVE_NOT_UNIQUE_ERROR**: la mossa inserita deve poter essere effettuata da un solo pezzo;
- **KING_IN_CHECK_ERROR**: la mossa eseguita non deve lasciare il proprio re sotto scacco;
- **PROMOTION_ERROR**: nella mossa devono essere inseriti gli indicatori corretti di promozioni quando si effettua una promozione.
  È gestito anche il caso in cui un pedone arrivi nell’ultima casella senza effettuare una promozione;

- **CHECK_NOT_CORRECT_ERROR**: se la mossa mette il re avversario sotto scacco deve essere specificato nella mossa tramite un '+'. Se lascia l'avversario sotto scacco da due pezzi allora deve essere indicato tramite '++'.
  Viene gestito anche il caso in cui '+' e '++' sono inseriti in una mossa che non mette il re avversario sotto scacco.
- **CHECKMATE_NOT_CORRECT_ERROR**: se la mossa mette il re avversario sotto scacco matto deve essere specificato nella mossa tramite un '#'.
  Viene gestito anche il caso in cui '#' sono inseriti in una mossa che non mette il re avversario sotto scacco matto.
- **ENPASSANT_ERROR**: l'indicatore ' ep' non deve essere inserito in mosse diverse dall'en-passant;
- **CASTLE_ERROR**: se la mossa è 'O-O' oppure 'O-O-O' l'arrocco corto o lungo deve essere possibile;
- **TURN_NUMBER_ERROR**: il numero del turno deve essere inserito correttamente in ordine crescente (a partire dall'uno);
- **PREAMBLE_NOT_POSSIBLE_ERROR**: vengono segnale possibili errori nel preambolo, nello specifico:
  - I pezzi non devono essere inseriti dove sono già presenti altri pezzi;
  - Le caselle devono avere righe e colonne comprese tra 1-8 e a-h;
  - Deve esserci uno ed un solo re per colore;
  - I pedoni non devono essere inseriti nella prima e nella ultima riga della scacchiera;
  - Deve esserci un solo preambolo per giocatore.
- **PREAMBLE_DRAW_ERROR**: La partita non deve partire da una situazione di patta;
- **STARTING_KING_CHECK_ERROR**: Il re di chi non parte non deve essere sotto scacco.
- **STARTING_TURN_ERROR**: il colore di chi dovrebbe iniziare deve essere in linea in quanto scritto nel preambolo (il primo preambolo inserito indica il giocatore che parte);
- **IMPOSSIBLE_MOVE_ERROR**: la mossa deve poter essere eseguita da almeno un pezzo;
- **LAST_TURN_ERROR**: se si ha un turno con solo la mossa del bianco si presuppone che la partita sia finita e che quindi non devono esserci altri turni.

Oltre a questi errori è stato inserito anche un warning (**NOTATION_WARNING**) il quale indica all'utente che una mossa inserita poteva essere abbreviata, es. Pa4a5 -> Pa5.

(Per il funzionamento di ogni controllo fare riferimento direttamente al codice dove ogni metodo è stato commentato opportunamente).

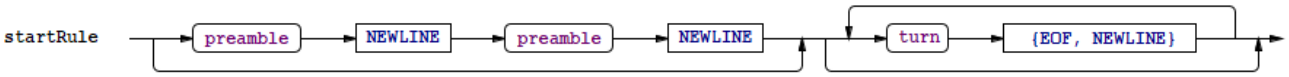
GRAMMATICA ANTLR

CONTROLLO LESSICALE

I token utilizzati nella grammatica sono i seguenti:

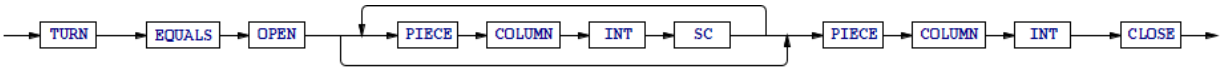
```
PIECE : ('R' | 'B' | 'N' | 'Q' | 'K' | 'P');
COLUMN  : 'a'..'h';
INT   : '0'..'9'+;
MINUS : '-';
PLUS : '+';
EQUALS : '=';
HASH : '#';
TAKE : ('x' | ':');
CASTLE : 'O';
EP : ' ep';
TAB : '\t';
POINT : '.';
NEWLINE : '\r'? '\n';
OPEN    : '[';
CLOSE   : ']';
TURN : ('white' | 'black');
SC : ';';
```

Tramite l'utilizzo di questi token sono state create le regole della grammatica, nello specifico si ha la **startRule**:

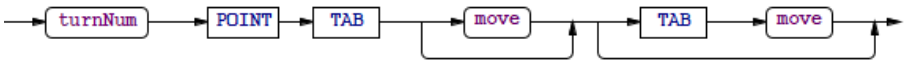


Questa è composta da:

- **Preamboli** (due ma non obbligatori)

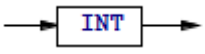


- **Turni** (Ciclo, composti da: numero turno + tab + mossa "bianca" + mossa "nera")

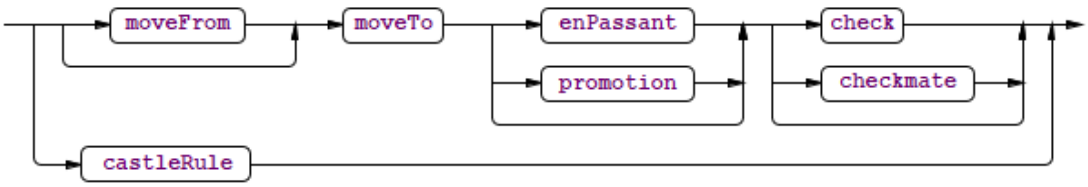


  A sua volta i turni sono composti da:

  - **Numero turno**

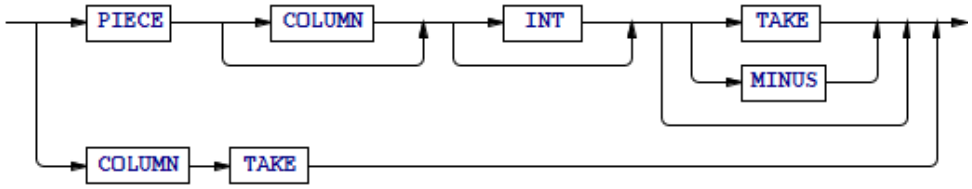


  - **Mossa:** qui vengono gestiti tutti i caratteri possibili in una mossa

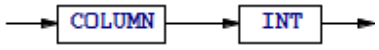


Per la gestione delle mosse sono state create le seguenti regole:

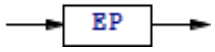
- **MoveFrom**



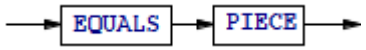
- **MoveTo**



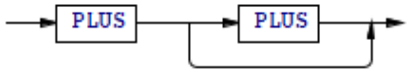
- **EnPassant**



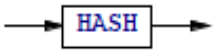
- **Promotion**



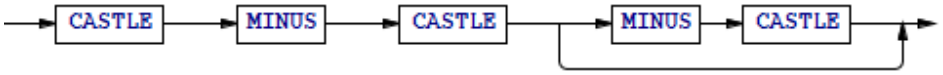
- **Check**



- **CheckMate**



- **CastleRule**



CONTROLLO SEMANTICO

Di seguito è riportata la grammatica comprensiva dei comandi eseguiti nella classe "**compilerChecker**" per il controllo semantico delle mosse/partita.

La maggior parte di questi metodi sono utilizzati per settare dei parametri che vengono utilizzati durante i controlli semantici successivi.

Si effettuano quindi controlli su:

1. Il preambolo assicurandosi che non generi gli errori visti nella sezione precedente;
2. La riga, controllando il numero di turno;
3. La mossa, utilizzando le variabili settate in precedenza per analizzarla ed invocare se necessario uno dei molteplici errori che possono essere stati commessi.

```
startRule
    : (preamble NEWLINE preamble NEWLINE)? {cc.checkChessboard();}
    (turn (NEWLINE | EOF) {cc.checkCorrectStartingTurn();})*
    ;

preamble
        :
        t=TURN {cc.setPrambleStartTurn($t);}
        EQUALS
        OPEN
        (p=PIECE c=COLUMN r=INT SC {cc.checkPreamblePlacement($p,$t,$r,$c);})*
        (p1=PIECE c1=COLUMN r1=INT {cc.checkPreamblePlacement($p1,$t,$r1,$c1);})
        CLOSE
        ;

turnNum :
        v=INT {
                cc.setTurnNumber($v);
                cc.isTurnCorrect();
                }
        ;

moveFrom :
        (p=PIECE {cc.setPiece($p);}
        (c=COLUMN {cc.setColFrom($c);})?
        (r=INT {cc.setRowFrom($r);})?
        ((t=TAKE | t=MINUS) {cc.setTake($t);})?)  |
        ((c=COLUMN t=TAKE)      {cc.setTake($t);
                                 cc.setColFrom($c);})
;
moveTo  :
        c=COLUMN {cc.setColTo($c);}
        r=INT {cc.setRowTo($r);
                cc.setLastToken($r); }
;

enPassant: EP   {cc.setEnpassant();};

check   : PLUS {cc.setChecks();}
         (PLUS {cc.setChecks();})?
          ;

checkmate : HASH {cc.setCheckMate();};

promotion : EQUALS p=PIECE {cc.setPromotion($p);};

castleRule:
        t=CASTLE {cc.setLastToken($t);}
        MINUS {int i=1;}
        CASTLE
        (MINUS CASTLE {i = 2;})?

        {cc.setCastle(i);};

turn
    : turnNum
        POINT
        TAB
        (move {cc.processMove();})?
                {cc.nextTurn();}
        (TAB
        move {cc.processMove();})?
                {cc.nextTurn();};

move
    : ((moveFrom?
                moveTo
                (enPassant | promotion)?
                (check | checkmate)?)
        | castleRule) ;
```