

# Block 1 Project

...by MAKSIM SIMONOV (M00840986)

DIMITRIOS TASIS (M00763303)

AFIF RAHMAN (M00806993)



# Contents

<b>The idea. Chess opening</b>	<b>3</b>
<b>Diagram</b>	<b>4</b>
<b>DrRacket programming</b>	<b>5</b>
<b>Sobs to be observed</b>	<b>10</b>
<b>References</b>	<b>11</b>

# The idea. Chess opening

The first few moves of a chess game can be some of the most important moves you make. In those moves, you will establish your early plans and fight for your place on the board. (chess.com)

But sometimes some players find it boring to make the same opening from time to time, because they want to improve their “after opening” part of the game, in other words middlegame.

Thus, our project will allow players to use their time at its highest standard in the case they have already known basic openings perfectly.

We are going to observe the opening called - **“The Italian Game”**.

The Italian game begins with:

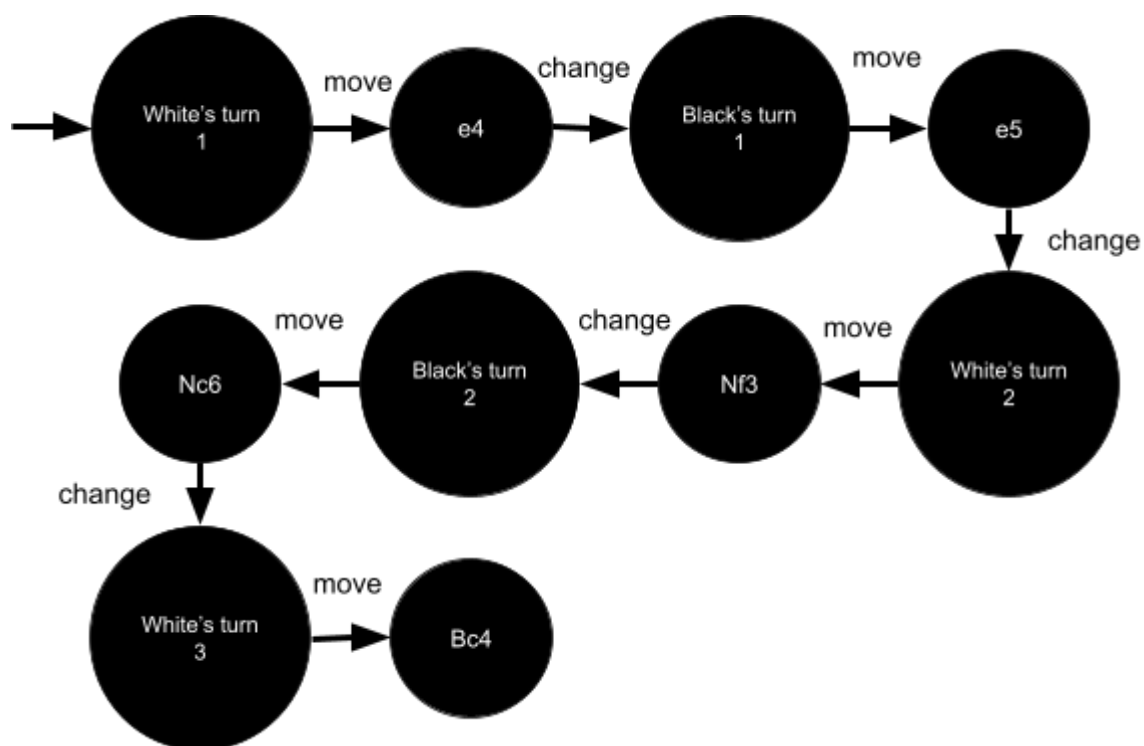
1. e4 (white pawn moves from e2 to e4)
1. e5 (black pawn moves from e7 to e5)
2. Nf3 (white knight moves from g1 to f3)
2. Nc6 (black knight moves from b8 to c6)
3. Bc4 (white bishop moves from f1 to c4)





*-The point is to control the center quickly with your pawn and knight  
and then put your bishop on it's most dangerous square.*

Hopefully, his coded opening will help both players to save time before starting the middle part of a game.

# Diagram



Big circles 	In order to be precise, the next move will be done by white/black pieces, but not vice versa
Small circles 	Making moves to a square

# DrRacket programming

## 1. Making a sequence of states and events.

*Defined as follows: ((“initial state” “event”) “next state”)*

```
→ (define fsm '(  
→ ("whiteturn1" "move" "e4")  
→ ("e4" "change" "blackturn1")  
→ ("blackturn1" "move" "e5")  
→ ("e5" "change" "whiteturn2")  
→ ("whiteturn2" "move" "Nf3")  
→ ("Nf3" "change" "blackturn2")  
→ ("blackturn2" "move" "Nc6")  
→ ("Nc6" "change" "whiteturn3")  
→ ("whiteturn3" "move" "Bc4")  
→ ))
```

It is a list containing states & events.

(Every row has an initial state followed by event, which leads to next state. Bc4 is the final state, which finalizes our “Italian” opening.)

## 2. Making the next state.

By Maksim

;Moving to the next state

```
(define next-state (λ (state event fsm)
  (cond
    ((and (string? state) (string? event))
      (cond
        ((empty? fsm) "error404")
        ((and
          (equal? state (first (first fsm))))
          (equal? event (first (reverse (first fsm))))
          (first (reverse (first fsm))))
        (else
          (next-state state event (rest fsm)))))
    (else "error 505"))))
```

The code above checks whether it is string or not, after the confirmation it takes the state and the event we have put (it can be any state and event (event should relate to the state / correct logic)). Then the code prints the next state from fsm (fsm - state table).

## 2.1 A compact version of next-state

By Dimitrios (We have not put it into Racket)

```
;DIMTRIOS TASIS
(define next-state2 (lambda (a b c)
  (cond
    ((empty? c) "nothing") ;is the finite state machine list empty?

    ((and (equal? a (caaar c)) ;is the input (a) or rather state given equal to
          ;(list (first list (first element(state))?)
          ;c (finite state machine table)))

      (equal? b (cadaar c))) ;is the event b equal to
      ;(list (first list (second element(event) c)
      (cadar c)) ;if both of them come true then cadar = (car (reverse (car c)))

    (else
     (next-state a b (cdr c)))));if a b not applied above then look at
                                ;the rest of the list c
                                ;by recusing the function (a b) with the rest c
```

```
(define next-state (lambda (a b c) ;(init-state event-seq fsm) <--- the arguments for

  (cond

    ((empty? c) "nothing")

    ((and (equal? a (caaar c))

      (equal? b (cadaar c)))

      (cadar c))

    (else (next-state a b (cdr c))))))
```

How does it work? This function first checks if the list/ input of the user's finite state machine is empty. If so, the first condition is fired, where the code outputs "nothing". It then checks if the input a is equal to the "caaar" of c, which is the first (first (first c))). This check is then done with b and c through cadaar and cadar, which is the first and rest function combined together. This is a "else" statement in case a or b is not applied, which looks at the rest of the list through recursion of the function a b with the rest of c.

### 3. Making a run-sequence.

By Maksim Dimitrios Afif

Run-seq for "Italian game opening".

```
(define run-seq (λ (init-state event-seq fsm)
  (cond
    ((null? event-seq) "false")
    (else
     (set! init-state (next-state init-state (first event-seq) fsm))
     (sleep 1)
     (println init-state)
     (run-seq init-state (rest event-seq) fsm )
     (sleep 1)
     ))))
```

The function above, called run-seq, takes 3 arguments (init-state, event seq fsm).

By applying the main sequence of events the function runs from the initial states to the final. It displays all chess moves by printing them one by one with the delay.

Main sequence:

```
(run-seq "whiteturn1" '( "move" "change" "move" "change" "move" "change" "move"
"chang" "move") fsm)
```

The sequence above runs our idea completely.





### 3.1 The main sequence.

By Dimitrios Maksim Afif

```
; DIMITRIOS , AFIF , MAKSIM

(define run-seq1 (lambda (a b c)
  (cond
    ((null? b) #f)
    (else
     (let ([current-state (next-state a (car b) c)])
       (sleep 1)
       (println current-state)
       (run-seq1 current-state (cdr b) c))))))

;let current state = recurse ;
;next-state with a (car b) and c
;(sleep 1) ; delay 1 second
;(displayln a)))) ; print the states
;(run-seq a (cdr b) c)
;repeating the process by looping
;the function and looking at the rest of the events
```

```
(define run-seq1 (lambda (a b c) ;(init-state event-seq fsm) <--- the arguments for run-seq1

  (cond

    ((null? b) #f)

    (else

     (let ([current-state (next-state a (car b) c)])

       (sleep 1)

       (println current-state)

       (run-seq1 current-state (cdr b) c))))))
```

Purpose: This is similar to the next-state function, but it outputs the final state without any of the other states in between. How it works: This function is the same as the run-seq1 function but through the use of the let function, which allows the next-state function inside the current-state equal to c. Sleep() pauses the code for an amount of time, i.e. 1 and then continues executing it after the time has elapsed.

## Sobs to be observed

**10**

**21**

**103**

**110**

**111**

**113**

**144**

**147**

# References

1. <https://www.chess.com/article/view/the-best-chess-openings-for-beginners>