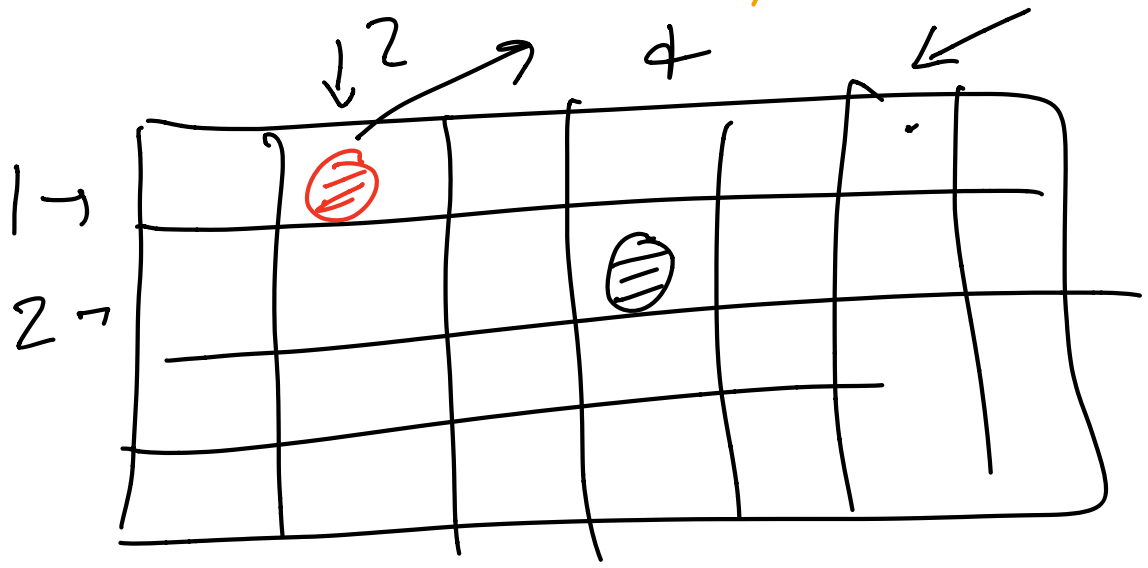
```
(define-struct Piece
  [color : Color]
  [loc   : Loc])
```

store info about one pice

```
(define-struct Loc
  [row : Integer]
  [col : Integer])
```

```
(define-struct Checkers
  ([pieces : (Listof Piece)]
   [turn : Color]))
```



The Red piece will have
a structure
color - 'red
(Loc 1 2)

(List (Piece 'red (Loc 1 2))

(Piece 'black (Loc 2 +1))

turn - (Whose turn is it?
Is it the black piece
or red piece)

Part a)

Modify the checkers structure

clicked-piece

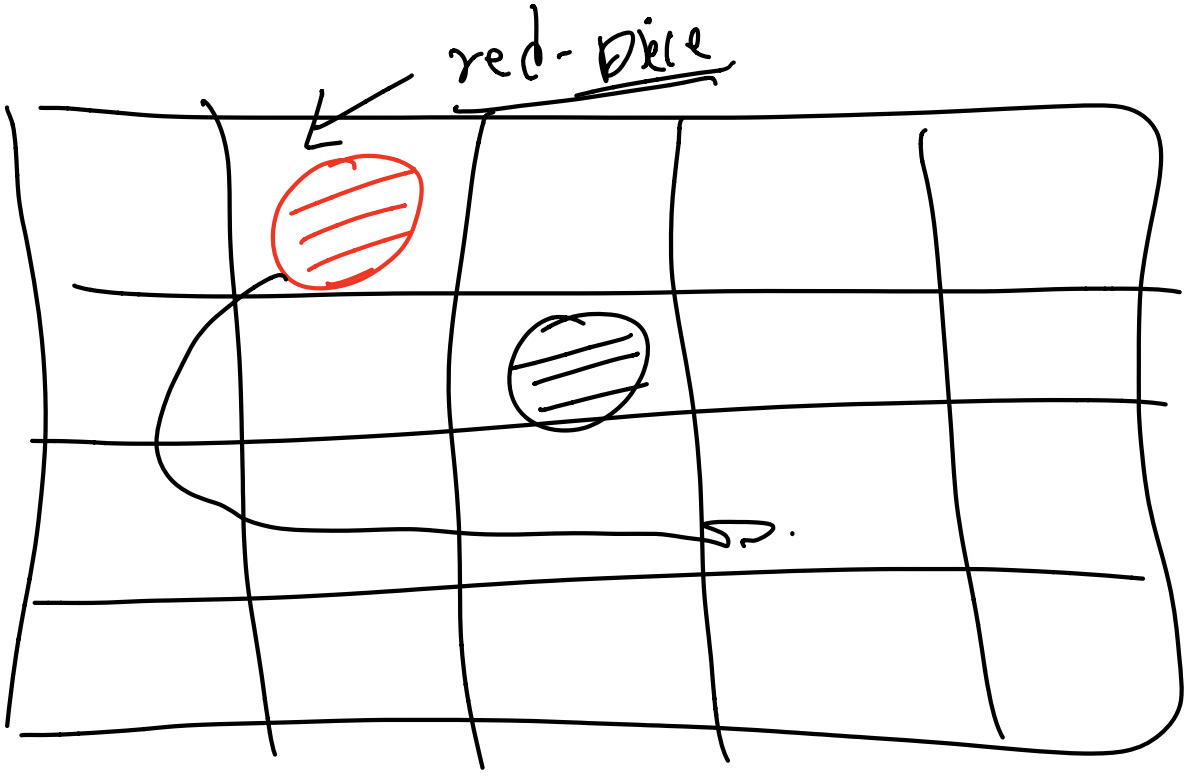
```
(define-struct Checkers
 ( [pieces : (Listof Piece)]
   [turn : Color ]
   [clicked-piece : (U 'none Piece)]))
```

↳ modification being asked

Part b)

click-piece

(: click-piece : Pice Checkers
→ Checkers)



Checkers

pieces - [Piece1 Piece2]

turn -

→ Remove it from this lv

clicked-piece ← None

Checker

pieces - [Piece 2]

turn -

clicked-piece - Piece 2

Given a piece and list of pieces Remove this piece from the list of pieces

Update the clicked-piece-field

Part c)

place-piece

(: place-piece : Loc Checkers
→ Checkers)

Clicked-piece - has a piece
(not 'none)
Update the old location of the
piece. with the new
location which is given in the
input.

Update the clicked-piece
field - 'none'

Add the piece which was in the clicked-piece field to the List Checkers pieces.

ch ⇒ pieces

(Checkers (List

('black 4 7))

'black ← turn

('red 3 5)) ←

↖ clicked piece

(place-piece ch (Loc 6 5))

(Checker (List ('red 6 5) ('black 4 7))

'none) .

Part d :

get-piec-or-none

(: get-piece-or-none : Loc
Checker → (U 'none Piece))

It will "iterate" (use recursion)
to check there is a piece
in the location Loc.

Pieces- Listof Piece

Porte

C: click-board: Checker Integer
Integer Mouse-Event → Checkers)

(define (click-board game-state
x y event)
(match event
("button-up" . [ ])

Porte ↓ Modifier.

[_ game-state] ← Point
Modify this

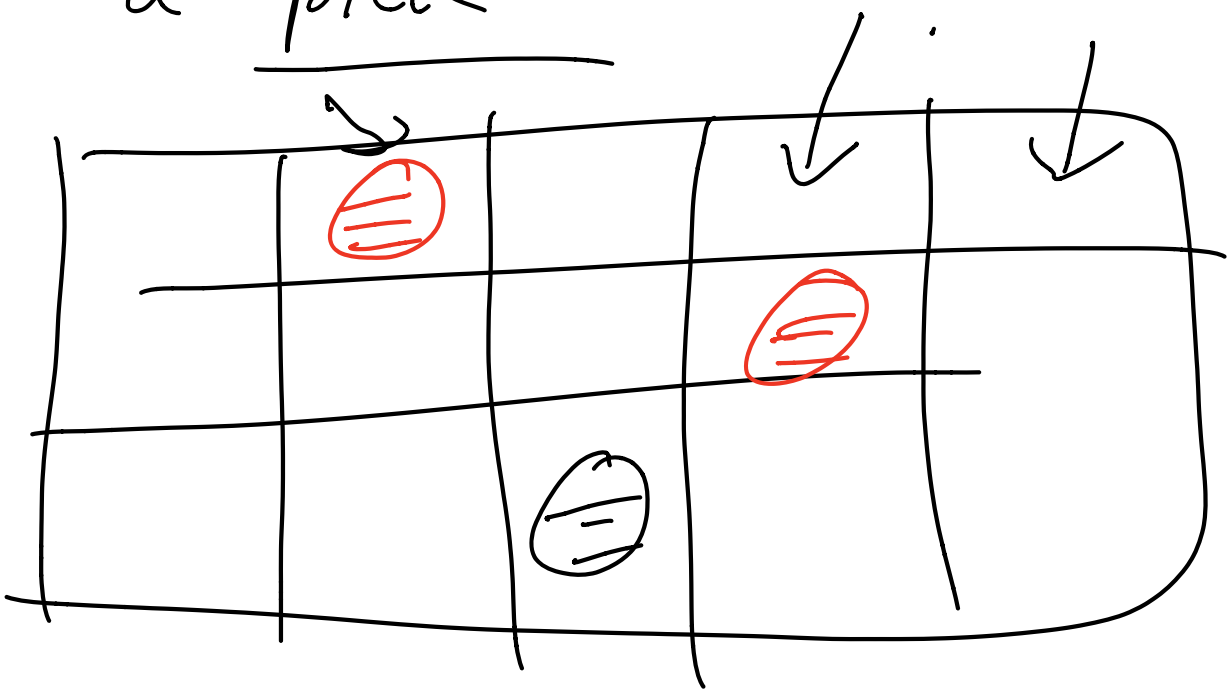
If clicked-piece is not

then you want to call
Note: Use xy→loc function

- (place-piece (xy->loc x y) game-state)

- clicked-piece is none

You are trying to select a piece



call get-piece-none

to check whether your clicked location has a piece

If there is a piece (
get-piece-or-none output
is not none)
call click-piece
input1 → piece - which is output
of get-piece-or-none.

input 2 → game-state

What happens if get-piece
-or-none outputs none?

Output should be the
input game-state.

Anonymous Functions

Functions are data.

```
( : int : Integer )
(define int 2)
```

```
(: my-func : Integer -> Integer)
(define my-func (lamda
                 ↑
                 Keyword
([n : Integer])
  (remainder n 75)))
```

(1 3 4)

```
(def three 3)
(def four 4)
(1 three four)
```

Example.

Append Write a expression which appends "!" to every string in the list of a string.

["no" "go"]
→ ["no!" "go!"]

```
(map (lambda (s : String)
       (string-append s "!"))
     (list "no" "go"))
```

Earlier we had to define your function before hand before using it with map, now you don't have to do that using lambdas.

Example 2

Comput the sum of squares of a (Listof Real)

[1 2 3 4]

→ $1^2 + 2^2 + 3^2 + 4^2 =$

```
(: sum-sq : (Listof Real)
            -> Real)
(define (sum-sq num)
  (foldr (lambda ([x : Real] [acc : Real])
           (+ (* x x) acc)) 0 num))
```

Currying

Example

Suppose you have a list of integers you want to add 1 to every

element of the list.

```
(: add-one : Integer -> Integer)
(define (add-one n)
  (+ n 1)))
```

```
(map add-one (list 1 2 3 4 1))
```

$\Rightarrow$ 2 3 4 5.

Write a function which adds two to every element in a list

```
(: add-two : Integer -> Integer)
(define (add-two n)
  (+ n 2)))
```

(map add-two (list 2 3 4))

⇒ 3 4 5 6

Is there a function which takes a integer as input output add_one version of that version.

Input

1 → add_one.

2 → add_two

900 → "add_hundred"

Answer is yes

```
(: add : Integer -> (Integer
                     -> Integer))
(define (add x)
  ;; We will write a lambda function
  ;; which takes an integer as
  ;; input and outputs input+x
  (lambda ([n : Integer]) (+ n x)))

(map (add 2) (list 1 2 3 4))
```

⇒ 3 4 5 6

(map (add 100) (list 1 2 3 4))

⇒ 101 102 103 104.

The name lambda

lambda calculus 1930s

↓

"The first computer"

≈

"The first programming language"

↳ Turing Machine.

We are looking at functional programming.