Lecture 4:

- Booleans
- Aggregates } structs

A nat number is
either
- 0, or
- 1+k, k nat

→ and that's it

Case n of
0 ⇒
| _ ⇒
... (n-1) ...

A boolean is
either
- true, or
- false

→ and that's it!

case b : bool of
true ⇒ _
| false ⇒ _

```
fun laughs (n: int) : string =
    Case n of
        0 => ""
      | 1 => "a"
      | _ => laughs(n-2) ^ "ha"
```

laughs(4) = "haha"

laughs(6) = "hahaha"

```
fun laughs (n: int) : string =
  case n of
    0 => ""
  | 1 => "a"
  | _ => laughs(n-2) ^ "ha"
```

A nat num is
either
- 0, or
- 1,
- k+2, k nat

—and that's it!

laughs(4)

↦ laughs(2) ^ "ha"

↦ (laughs(0) ^ "ha") ^ "ha"

↦ ("" ^ "ha") ^ "ha"

↦ "haha"

laughs(5)

= "ahaha"

```
fun laughs(n:int):str =
  case n of
    0 => ""
  | _ => (case evenP(n) of
            true => "h" ^ laughs(n-1)
          | false => "a" ^ laughs(n-1))
```

Parens around nested case

laughs(5)

↦ "a" ^ laughs(4)

↦ "a" ^ "haha"

↦ "ahaha"

laughs(4)

↦ "h" ^ laughs(3)

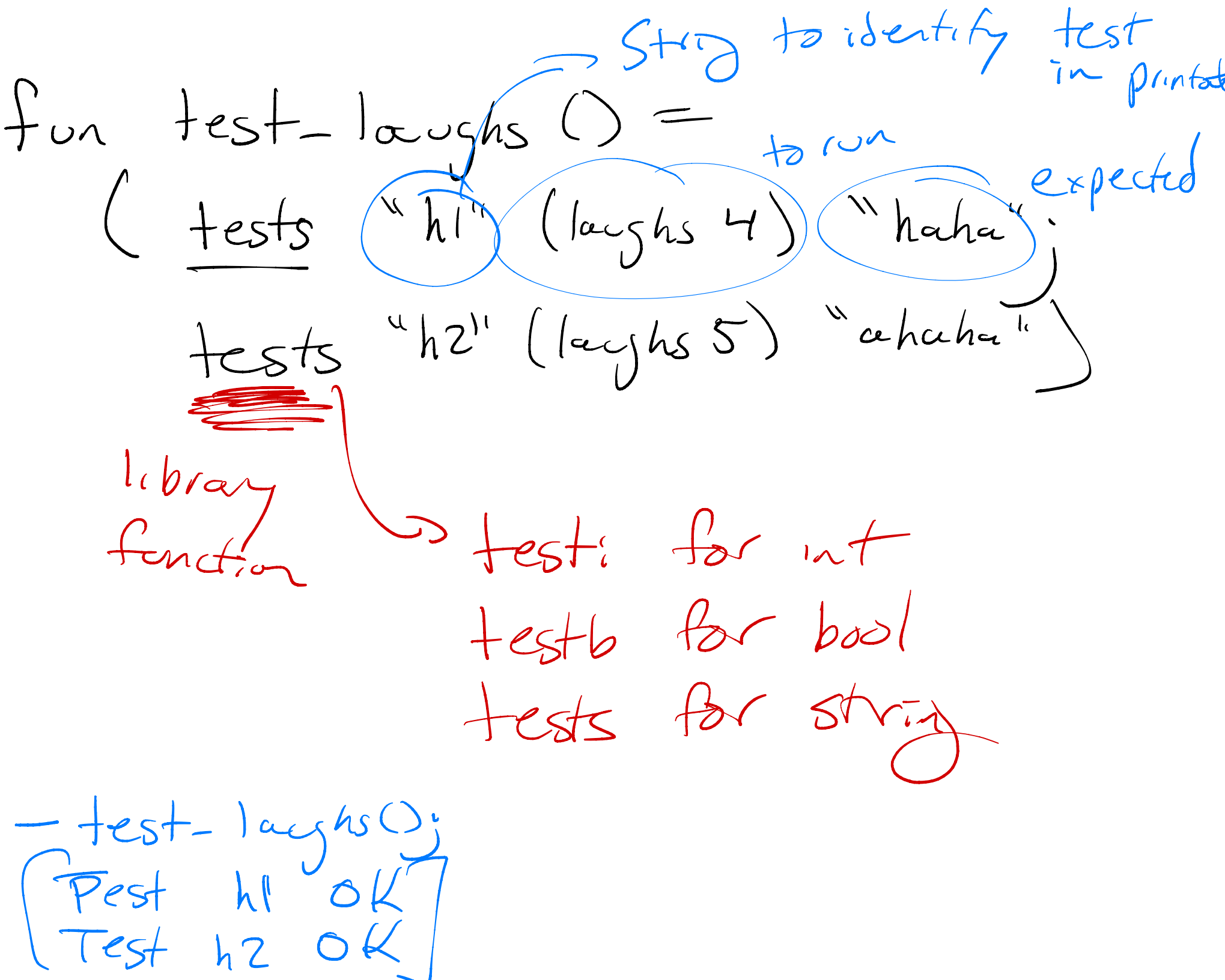
↦ "h" ^ "aha"

↦ "haha"

```
fun laughs(n:int):str =
    case n of
      0 => ""
    | _ => (case even?(n) of
              true => "h"
            | false => "a") ^ laughs(n-1)
```

laughs w/ case recursion on n-1 not n-2

case is an expression!


String to identify test in printout
to run
expected
fun test_laughs () =
(tests "h1" (laughs 4) "haha"
tests "h2" (laughs 5) "ahaha")
library function
test1 for int
test2 for bool
tests for string
- test_laughs();
[Test h1 OK
Test h2 OK]

1. Edit file in Atom / Vs code
2. Save file
3. use "hw02.sml";
4. — test_double();

   or

   — run();

→ make sure it actually loads

Aggregates

fixed-size
heterogeneous
collection of data

pairs

tuples

} structs
records
objects

2 input function → really takes a pair as input

```
fun add(x:int, y:int):int =
    case x of
        0 => y
      | _ => 1 + add(x-1, y)
```

0+y = y

1 + (x-1+y) = x+y

add(3,4) = 7

add(2,1) = 3

type int * int pair type

values (1,2)
(2,3)
(~17, 1001)

(V_1, V_2) V_1 int value
V_2 int value

ops let val (x,y) = p
in e'
end

struct
fields
x: int
y: int

```
fun add(x:int, y:int):int =
    case x of
        0 => y
      | _ => 1 + add(x-1, y)
```

"desugars"

```
fun add (p:int*int):int =
    let val (x, y) = p in
        case x of
            0 => y
          | _ => 1 + add(x-1, y)
    end
```

```
fun add (p:int*int):int =
  let val (x, y) = p in
    case x of
      0 => y
    | _ => 1 + add(x-1, y)
  end
```

add(3,4)

↦ let val (x,y) = (3,4) in ... end

↦ case 3 of 0 => 4 | _ => 1 + add(3-1, 4)

let val $(x, y) = (v_1, v_2)$ in e' end

steps to e' with v_1 substituted for x
v_2 subst. for y

---

let val $(x, y) =$ p : int * int in e' end : T

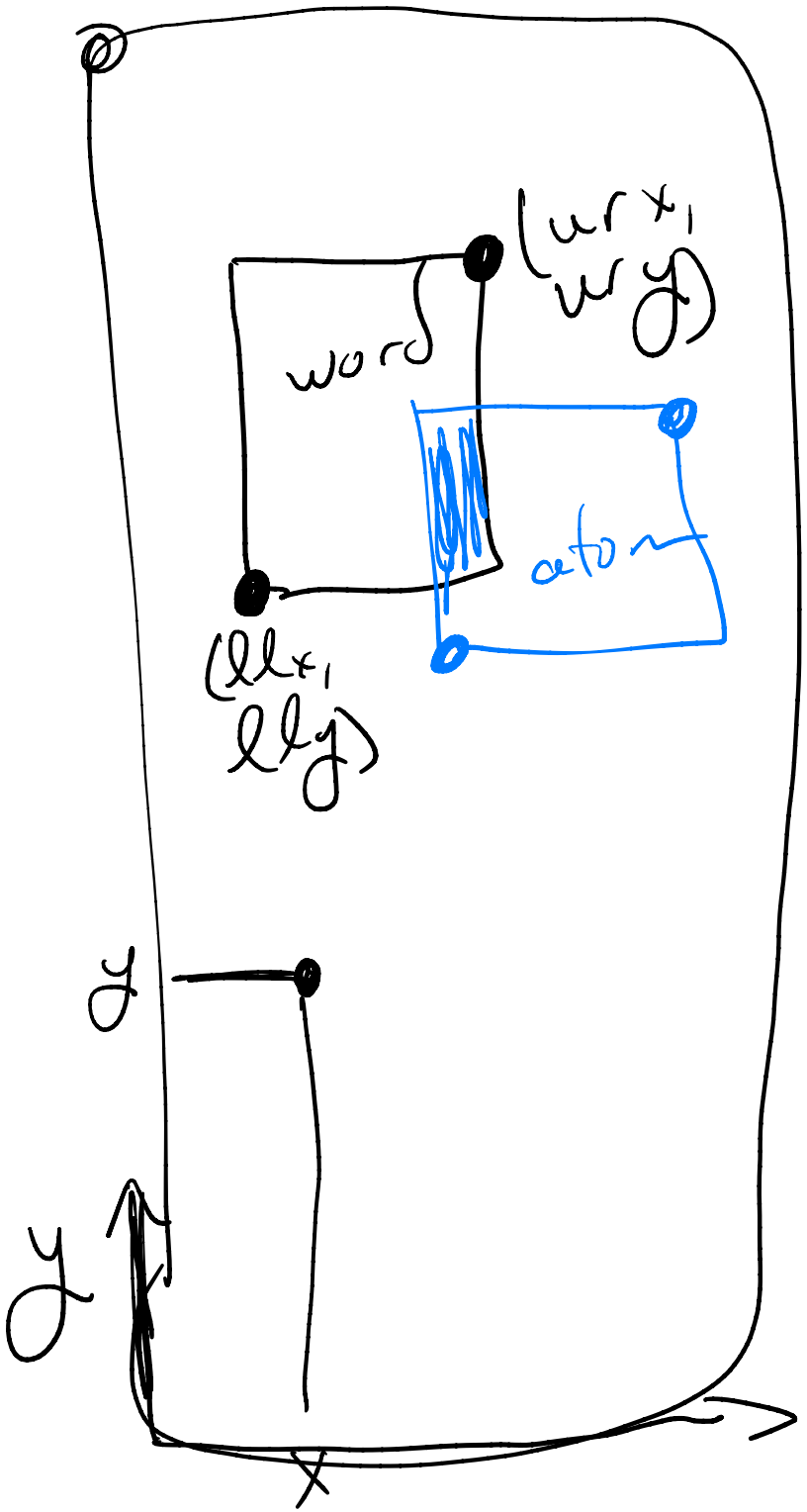
if e' : T assuming

vars x : int
y : int

All works for $T_1 * T_2$

with any types T_1 and T_2

int * string	(4, "ha")
(int * int) * string	((4,2), "ha")
bool * int	(true, 4)

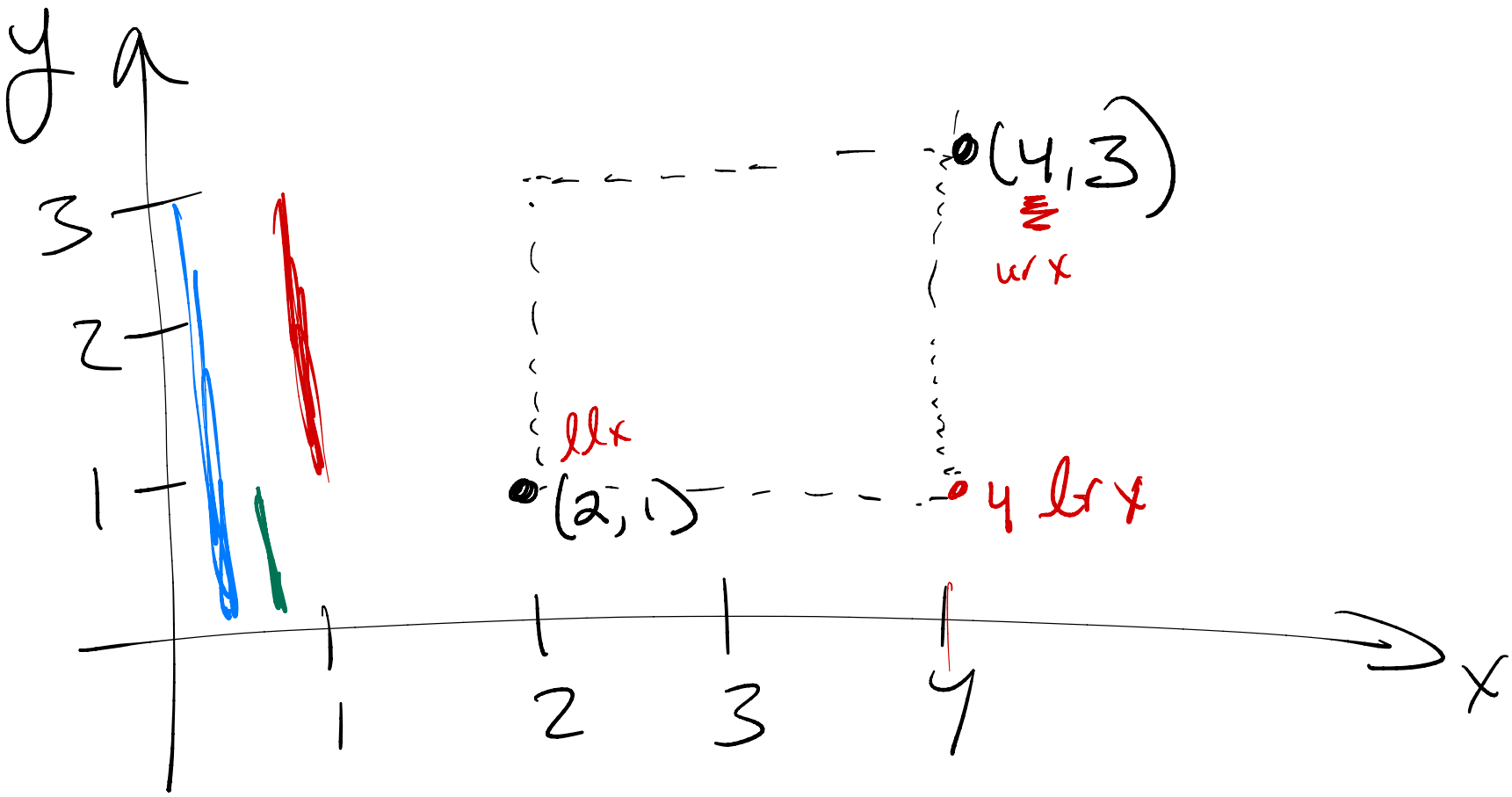


type point = int * int

type rect = point * point

(* (lower left, upper right) *)

(int*int) * (int*int)

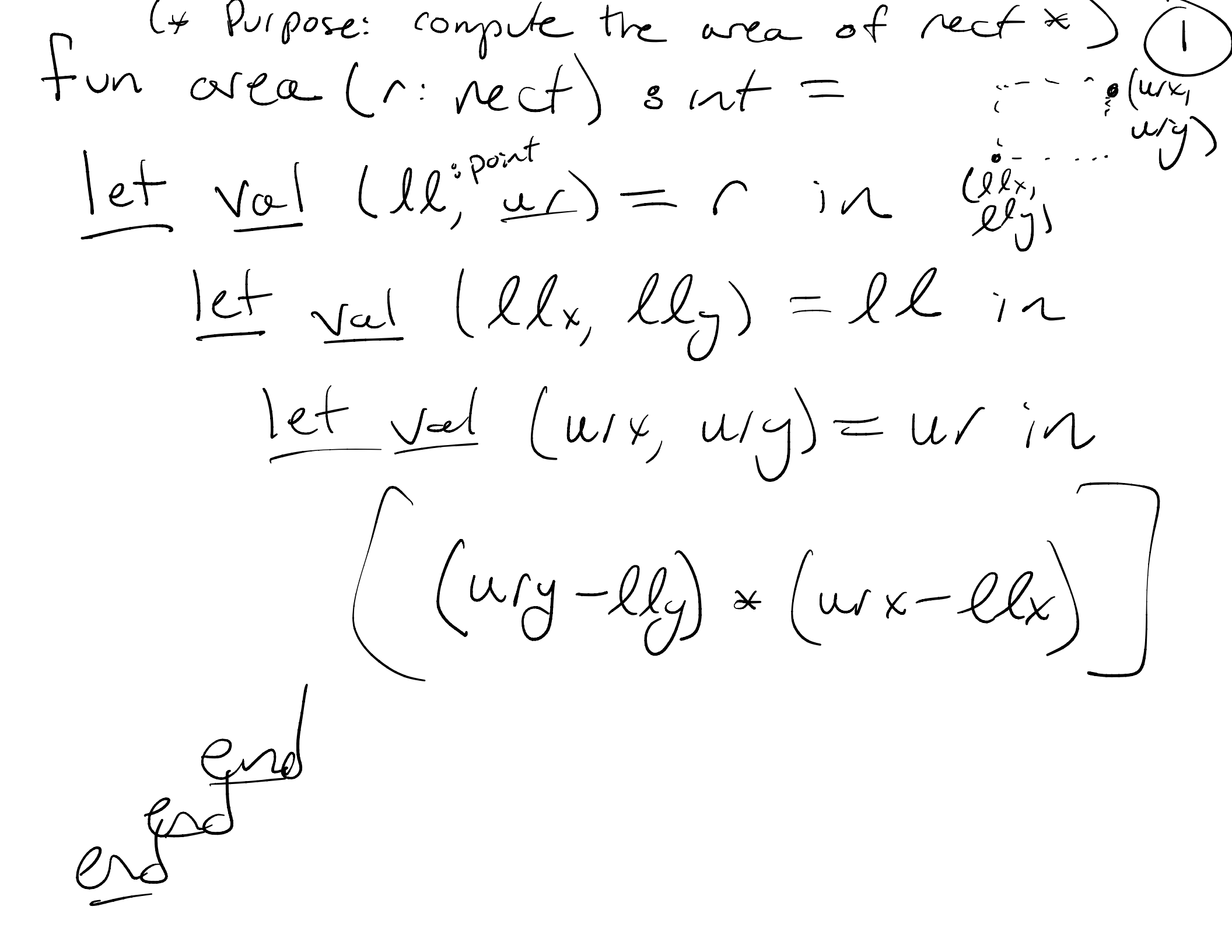


val test: rect = ((2,1), (4,3))

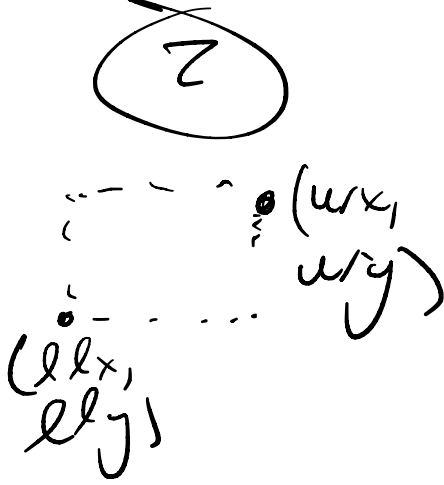
point point

point * point

```
(* Purpose: compute the area of rect *)
fun area (r : rect) : int =
  let val (ll : point, ur) = r in
    let val (llx, lly) = ll in
      let val (urx, ury) = ur in
        ((ury - lly) * (urx - llx))
      end
    end
  end
```



```
fun area (r : rect) : int =
  let val (ll, ur) = r
      val (llx, lly) = ll
      val (urx, ury) = ur in
  (ury - lly) * (urx - llx)
end
```



```
fun area (r : rect) : int =
  let val ((llx, lly), (urx, ury)) = r
  in
    (ury - lly) * (urx - llx)
  end
```

(3)

```
fun area (((llx, lly), (urx, ury)) : rect) : int =
  (ury - lly) * (urx - llx)
```

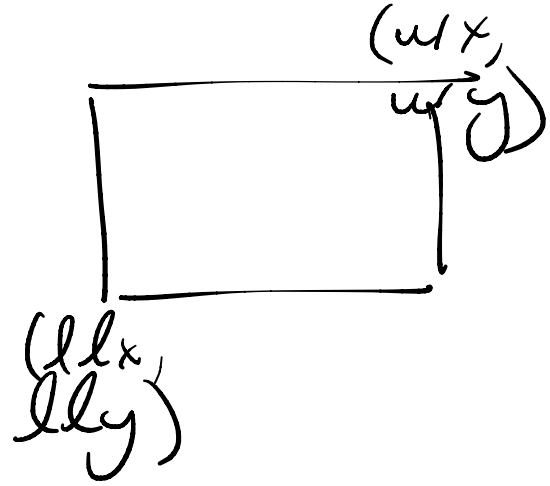
(4)

```
fun area (((llx, lly), (urx, ury)) : rect) : int =
    (ury - lly) * (urx - llx)
```

4

```
fun perim (((llx, lly), (urx, ury)) : rect) : int =
    2 * ((ury - lly) + (urx - llx))
```

4

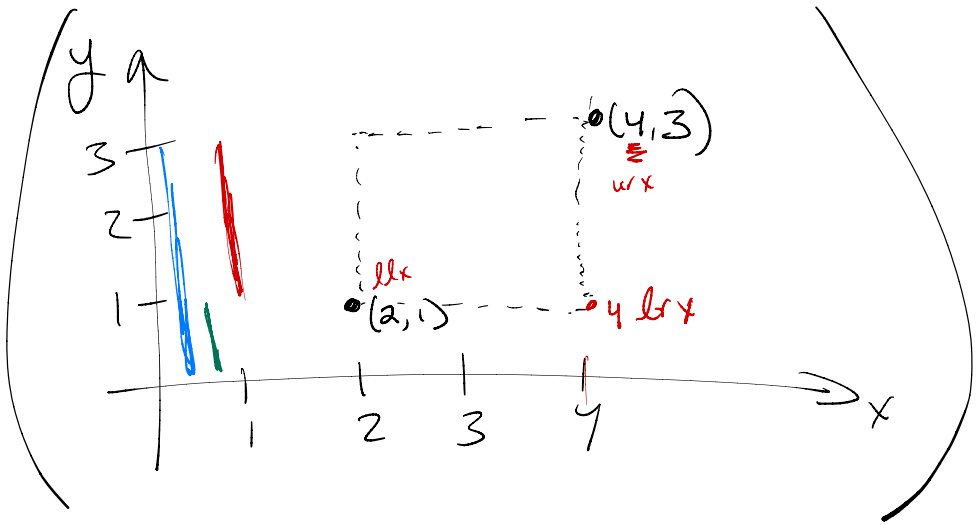


"Helper" function:

make up because it helps with what you wanted to do

```
(* Purpose: compute the (width, height) of r *)
fun sides (((llx, lly), (urx, ury)) : rect) : int * int =
    (urx - llx, ury - lly)
```

width, height

e.g. sides 

$= (2, 2)$

```
fun area ( r : rect ) : int =
    let val (w, h) = sides(r) in
        h * w
    end
```

① helper
② recursive call

```
fun perim ( r : rect ) : int =
    let val (w, h) = sides(r) in
        2 * ( h + w )
    end
```

area ((2,1),(4,3))

↦ let val (w,h) = sides ((2,1),(4,3))
in h * w
end

↦ let val (w,h) = (4-2, 3-1)
in h * w
end

↦ let (w,h) = (2,2)
in h * w
end

↦ 2 * 2 ↦ 4