Comp 212 Lecture 2

Computing by Calculation

Functional Programming
Standard ML (SML)

Calculational view of programs

VS

Machine-centric
Memory-centric (C)

State of a computation is just a program

Step the program until it is done/

↓

sequential
parallel

answer/
value

Expressions/Programs	Types	Values/answers
2	int	2
1+1	int	2
(1+2)*(3+4)	int	21
"a"	string	"a"
"a" ^ "b"	string	"ab"
intToString 5	string	"5"
"a"+1	no type	no value
5 div 0 (5 div 0 ✗) integer division	int	no value

$(1+2)*(3+4)$

$\longmapsto$ $3*(3+4)$

"steps"

$\longmapsto$ $3*7$

$\longmapsto$ 21

Sequential

3 steps

$(1+2)*(3+4)$

$\Longmapsto$ $3*7$

"parallel steps"

$\Longmapsto$ 21

Parallel

2 steps

Every type has a collection of values and operations

Int

values 0, 1, 2, ~1, ~2, ...

(negative)

operations + * div intToString

String

values "a" "b" "ab"

ops ^

Type checking:

1. each of the values has the indicated type

2. each operation is well-typed when the subexpressions are well-typed (and have the right types)

$(3+7)*5$: int because

"has type"

$(3+7)$: int b/c

3 : int b/c value

5 : int b/c value

(5) : int b/c value

"a" + 1 : ? int b/c

"a" : int false ✗

1 : int ✓

SML checks types → "compile-time"

before running a program → "run-time"

type error if inconsistant

5 div 0 : int b/c

5 : int ✓

0 : int ✓

---

raises an "exception" (Div)

(well-defined run-time error)

real

Values 1.0 3.14 ~2.17
2.0

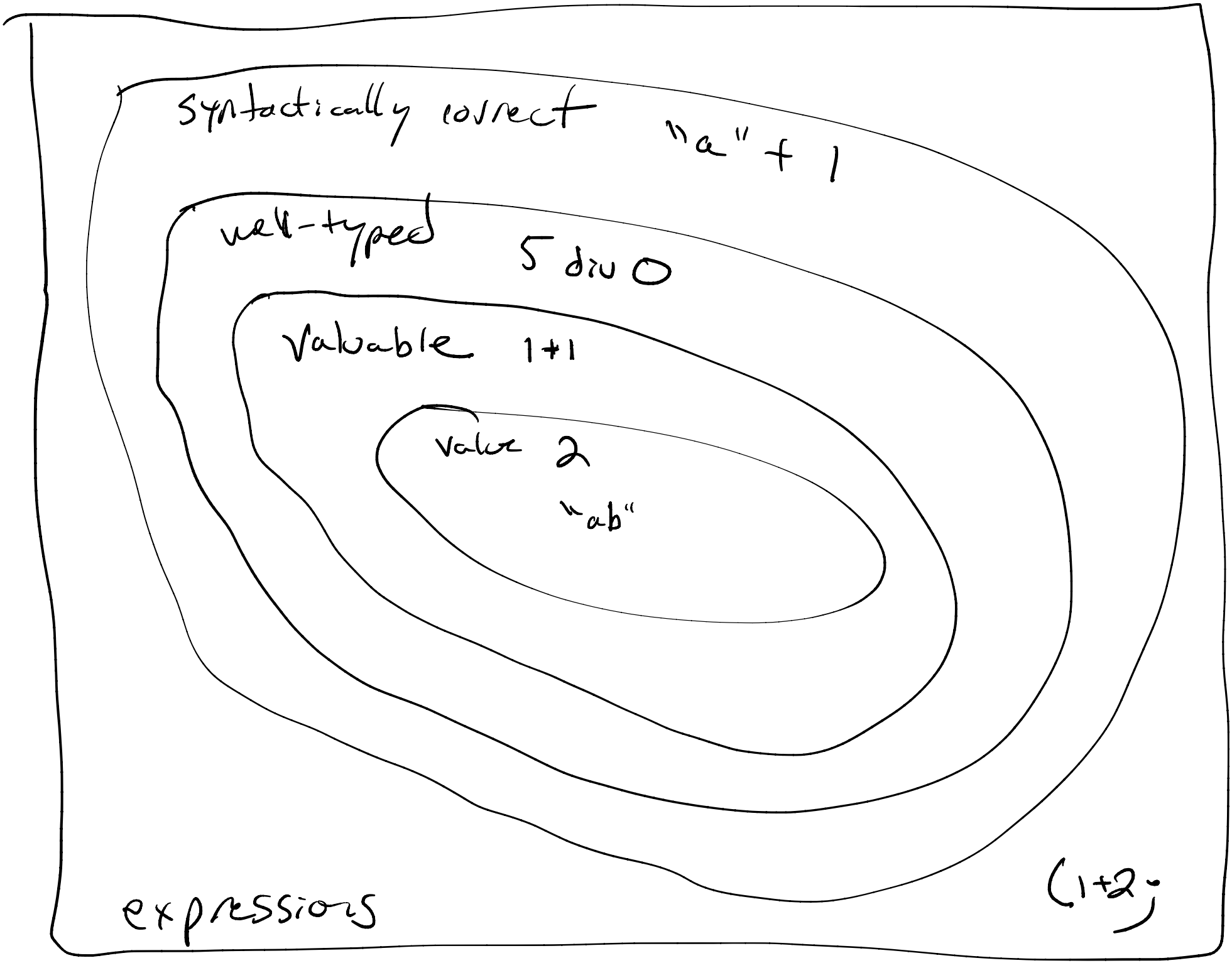
ops + * /

5 div 2 = 2

5.0 / 2 ill-typed

5.0 / 20 = 2.5

real 2
2.0


syntactically correct
"a" + 1
well-typed
5 div 0
valuable
1+1
value
2
"ab"
(1+2j
expressions

Variables

a variable is

a placeholder

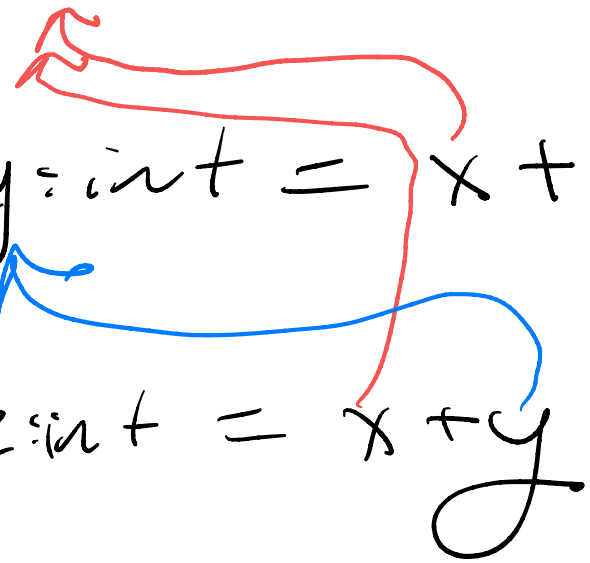
for a value.

```
val x : int = 2+3

val y : int = x + 1

val z : int = x + y

:
:
:
```



type checking declarations:

1. body has the indicated type (RHS)
2. assume the variable has the indicated type in code below

```
val x:int = 2+3
val y:int = x+1
val z:int = x+y
```

↦

```
val x = 5
val y = x+1
val z = x+y
```

substitution

↦

```
val x = 5
val y = 5+1
val z = 5+y
```

↦

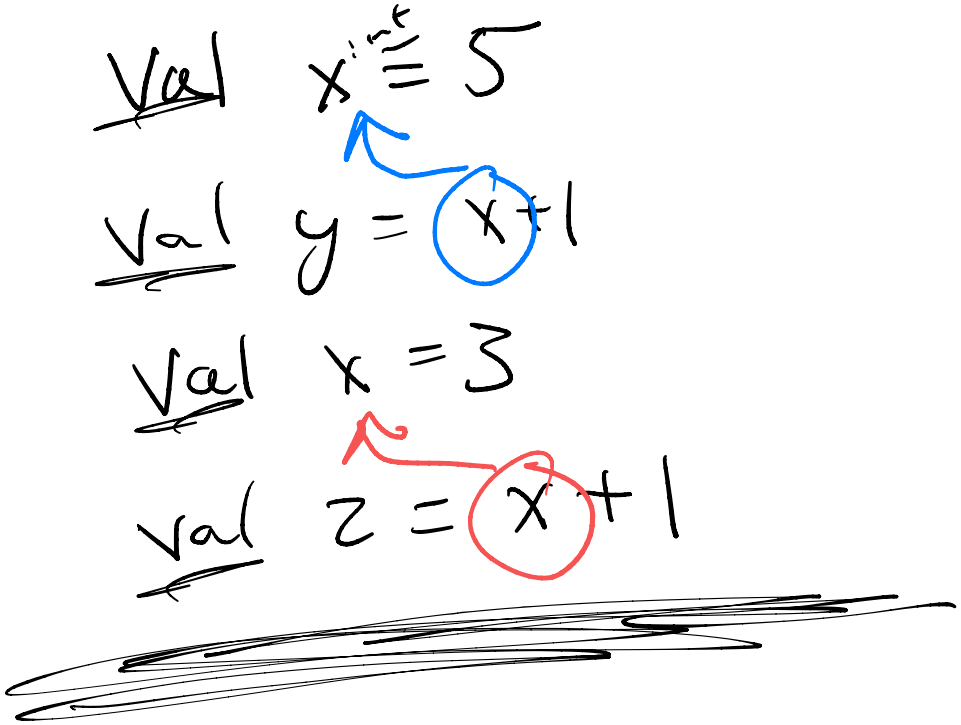
```
val x = 5
val y = 6
val z = 5+y
```

↦

```
val x = 5
val y = 6
val z = 5+6
```

↦

```
val x = 5
val y = 6
val z = 11
```



Variables refer to the nearest enclosing binder (declaration)

"shadowing"

```
val x = 5
val y = x+1
val w = 3
val z = w+1
```

"rename variables consistently"

Functions captures a pattern of computation

Math $f(x) = 2x + 6$

SML `fun f(x:int):int = 2*x+6`

- `f` — name of the function
- `x:int` — variable standing for the input; type of the input
- `int` — type of the output
- `2*x+6` — body

① var is assumed to have that type

② body must have the output type

fun f(x) = 2*x+6 to run it:

f(3)

↦ (2*3) +6

↦ ....

f(12)

↦ (2*12) +6

↦ ....

Substitute the actual input for variable and continues

f(1+1)

↦ ... ?

Lecture 3:

1. Functions
2. Aggregates
3. Design Recipe

Functions capture a pattern of computation

$$f(x) = 2x + 6$$

math

```
fun f (x: int): int = (2 * x) + 6
```

fun — keyword

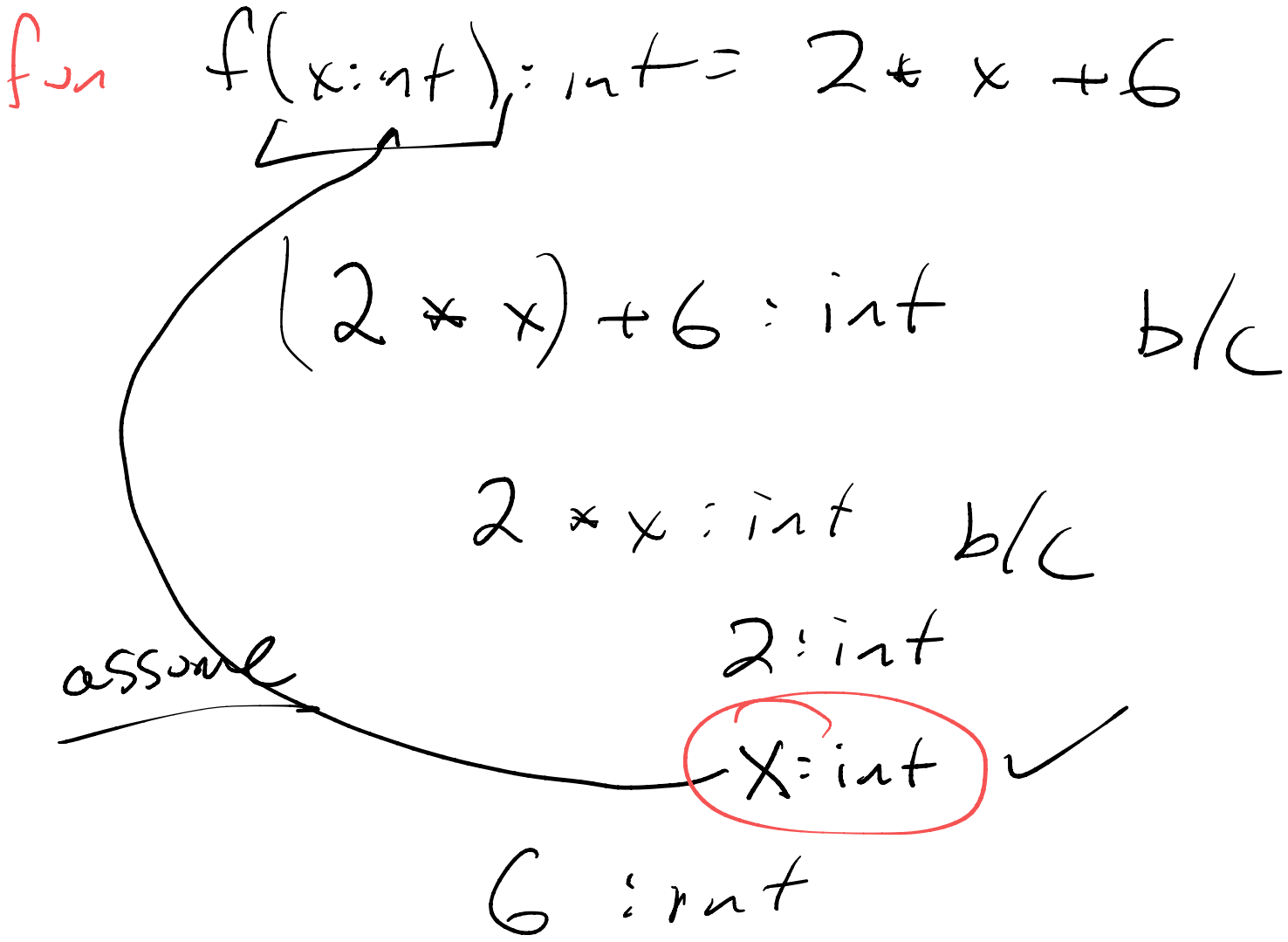
f — name

x — variable for input

int — input type

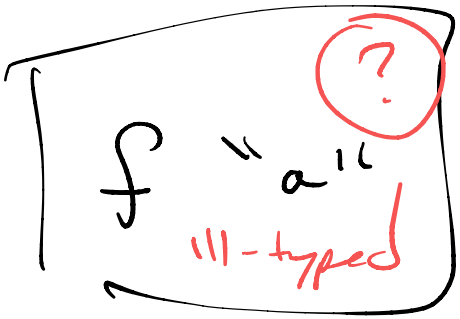
int — output type

(2 * x) + 6 — body


fun f(x:int):int = 2 * x + 6
(2 * x) + 6 : int b/c
2 * x : int b/c
2 : int
assume
x : int
6 : int

Function Application

f (3) f 3



If f is a function from int to int

then f 3 has type int

because 3 has type int

```
fun f(x) = 2*x+6
```

f 3

↦ (2*3)+6

↦ 6+6

↦ 12

f 4

↦ (2*4)+6

↦ 8+6

↦ 14

f(2+2)

↦ f(4)

↦ ....

① run the "argument" until it is a value

② substitute that value into the body

"call-by-value"

f(2+2)

↦ f(4)

↦ (2*4)+6

↦ ⋮

call by value

SML

f(2+2)

↦ 2*(2+2)+6

↦ 2*4+6

↦ ⋮

"call-by-name"

Haskell

f ((2+2)+(3+3))

↦ f (4+(3+3)) (1)

↦ f (4+6) (2)

↦ f 10 (3)

↦ 2*10+6 (4)

fun f(x) = 2*x+6

f ((2+2)+(3+3))

↦ 2*((2+2)+(3+3))+6 (1)

↦ 2*(4+(3+3))+6 (2)

↦ 2*(4+6)+6 (3)

↦ (2*10)+6 (4)

fun g(x:int) : int = x * x

g((2+2)+(3+3))

↦ g(4+3+3) ①

↦ g(4+6) ②

↦ g 10 ③

↦ 10 * 10

CBV

g((2+2)+(3+3))

↦ ((2+2)+(3+3)) * ((2+2)+(3+3))

① ↦ (4 + (3+3)) * ((2+2)+(3+3))

② ↦ (4 + 6) * ((2+2)+(3+3))

③ ↦ 10 * ((2+2)+(3+3))

CBN

```
val a :int = 2+2
val y = f a
```

$\longmapsto$ `val a = 4`
`val y = f a`

$\longmapsto$ `val a = 4`
`val y = f 4`

`f 4`

$\longmapsto$ `2*4+6`

$\longmapsto$ ⋮

CBN

$\longmapsto$ `val a = 2+2`
`val fy = f(2+2)`

`f (2+2)`

$\longmapsto$ `2*(2+2)+6`

$\longmapsto$ `(2*4)+6`

$\longmapsto$ `8+6`

$\longmapsto$ `14`

fun h(x:int) = 7

h((2+2)+(3+3))

↦ h(4+(3+3))

↦ h(4+6)

↦ h 10

↦ 7

CBV

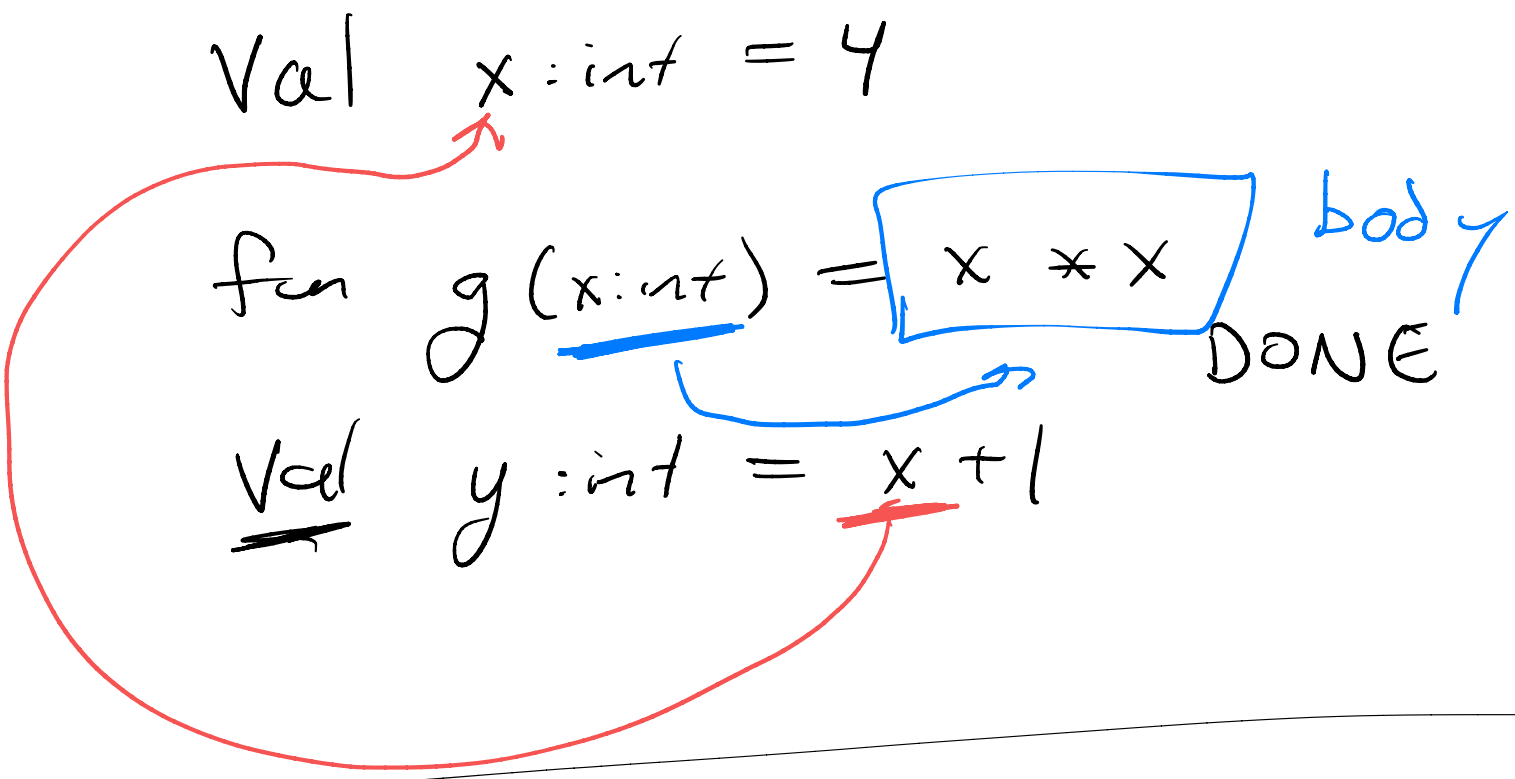
h((2+2)+(3+3))

↦ 7

call-by-need

CBN

nearest "enclosing" binder




```
val x:int= 4
val y: int = x+1
fun g (x:int) = x*y
```

```
fun g (x:int) : int =
    let val y = x+1
    in
        x * y
    end
```

body

```
let
    [decls]
in
    [expression]
end
```

1. y has indicated type in body
2. Right-hand side must have indicated type

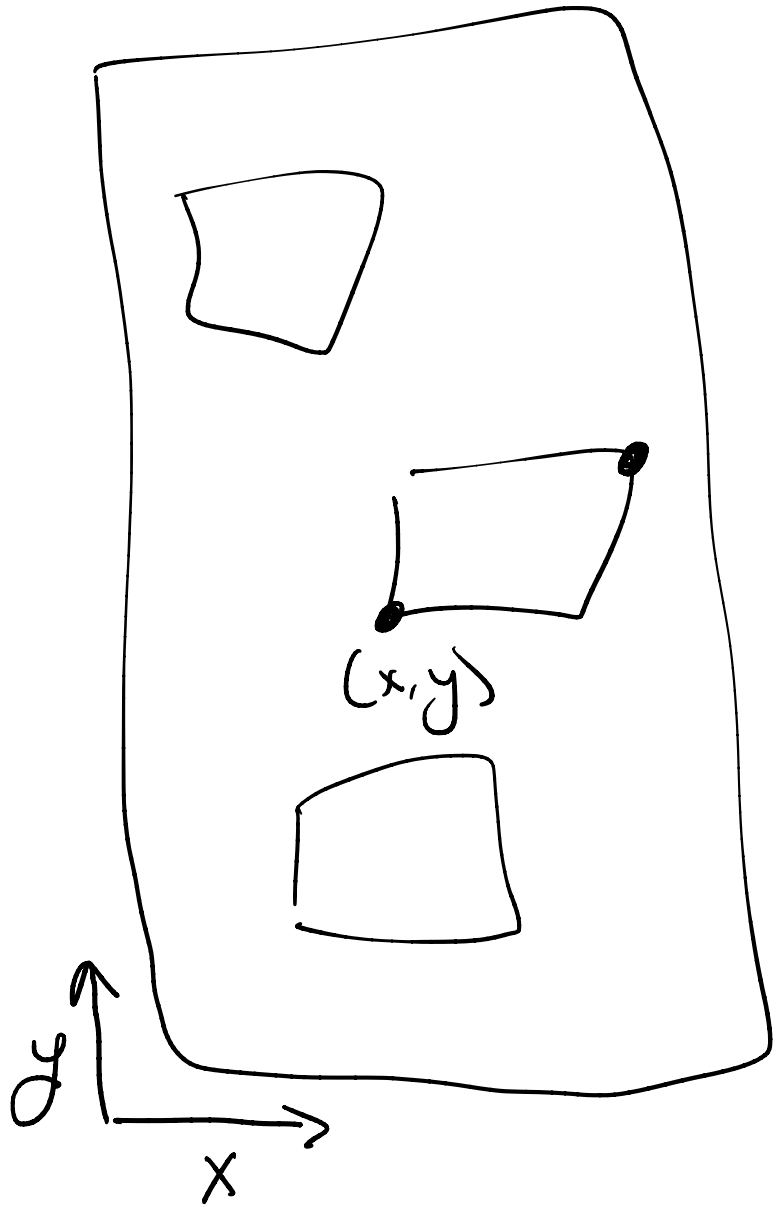
Aggregates → heterogeneous

but

- "structs"
- tuples
- pairs
- ⋮

fixed size collection of data

Rectangle



pos of at least 2 corners

① lower-left

② upper-right

type point = int * int

type rect = point * point

(int * int) * (int * int)

Values int * int

(v_1, v_2) $v_1 : int$

$v_2 : int$

e.g. (1,2) (11, 17) ....

(int * int) * (int * int)

Values ((1,2) , (3,4))

int * string

values (1, "a")

String * int

values ("a", 1)

---

operation let val (x, y) = [int] [int] [have type int * int]

in

[body]

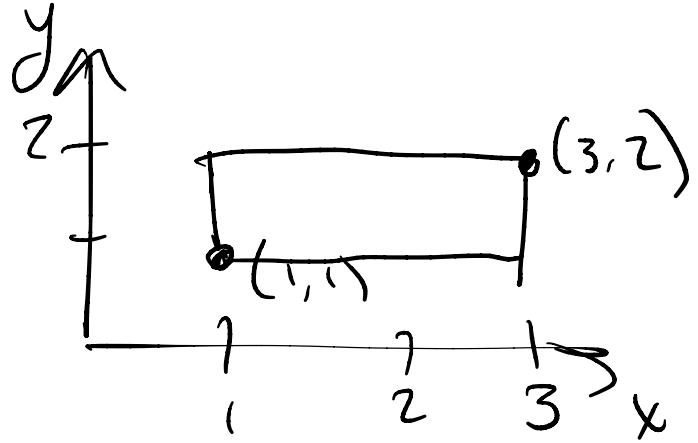
end

type point = int * int (* x y *) type rect = point * point

(* Purpose: calculate the area of the rect *)

fun area(r : rect) : int =

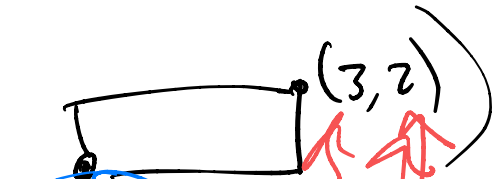
(* e.g.



((1,1), (3,2))
ll ur

area ((1,0),(4,2)) should be 6
area ((1,1),(3,2)) should be 2 *)
area ((1,1),(4,2)) should be 3

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



To step

let val $(a, b) = e_1$ in e_2

① step e_1 until it is (v_1, v_2) v_1 v_2 values

② substitute v_1 for a
v_2 for b
in e_2

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

↦ let val (ll, ur) = ((1,1),(3,2)) in ----

let val (llx, lly) = ll in

let val (urx, ury) = ur in ----

↦ let val (llx, lly) = (1,1)

in let val (urx, ury) = (3,2)

in (urx − llx) * (ury − lly) end end

↦ let val (urx, ury) = (3,2)

in (urx − 1, ury − 1) end

↦ (3−1, 2−1) ↦ --

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

multiple declarations

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

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

Methodology -

1. Purpose
2. Examples
3. Code
4. Test