
5x-2=13
then
x = 3
But professor, you said yesterday that x was equal to 2!

Functions

EFC // Lecture 4

2017

Similar expressions

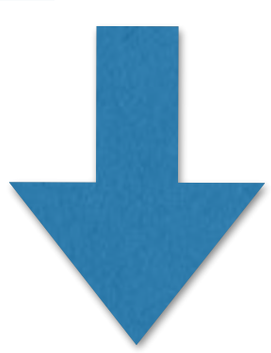
```
# let x = 5;;
val x : int = 5
# let square_x = x * x;;
val square_x : int = 25
# let x = 16;;
val x : int = 16
# let square_x = x * x;;
val square_x : int = 256
# let x = 8;;
val x : int = 8
# let square_x = x * x;;
val square_x : int = 64
```

doesn't change

changes!

If only we could '*parameterise*' expressions

```
# let square_x = x * x;;
val square_x : int = 64
```



```
# let square x = x * x;;
val square : int -> int = <fun>
```

FUNCTION!

How do we use functions?

"*Application*"

```
# let square (x) = x * x;;
val square : int -> int = <fun>
# let x = 5;;
val x : int = 5
# square (x);;
- : int = 25
# let x = 16;;
val x : int = 16
# square x;;
- : int = 256
# let x = 8;;
val x : int = 8
# square x;;
- : int = 64
```

Functions and *evaluation*

```
let square x = x * x
```

Think of *function application*

```
square 5
```

as the *definition of argument* x*:*

```
let x = 5 in x * x
```

Step-by-step

```
let square x = x * x
```

square (square 5)

Step-by-step

```
square x = x * x
```

```
square (square 5)
```

Step-by-step

```
square x = x * x
```

square (**square 5**)

Step-by-step

```
square x = x * x
```

square (**let x = 5 in x * x**)

Step-by-step

```
square x = x * x
```

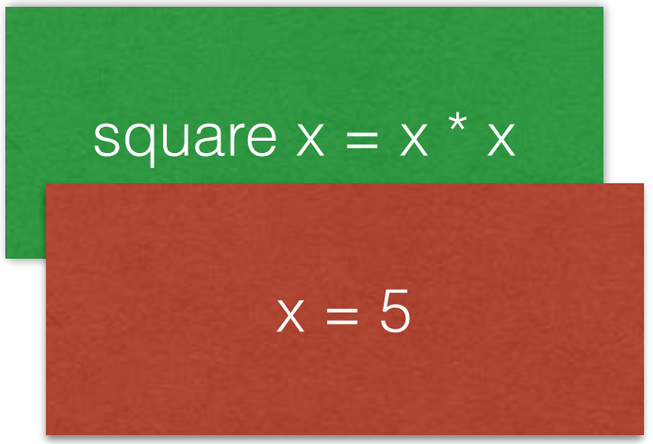
```
square (let x = 5 in x * x)
```

Step-by-step

```
square x = x * x
```

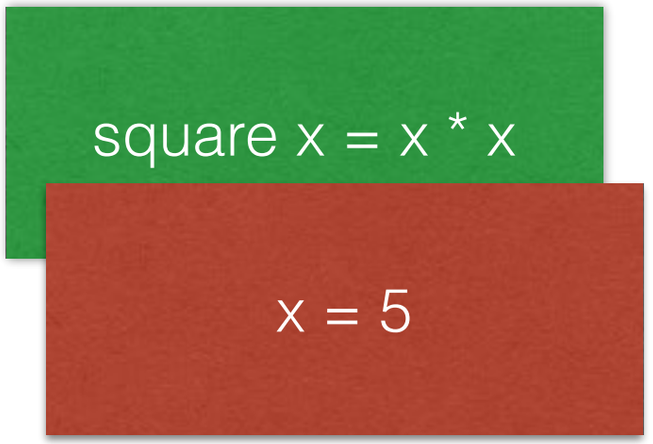
square (**let x = 5 in** x * x)

Step-by-step



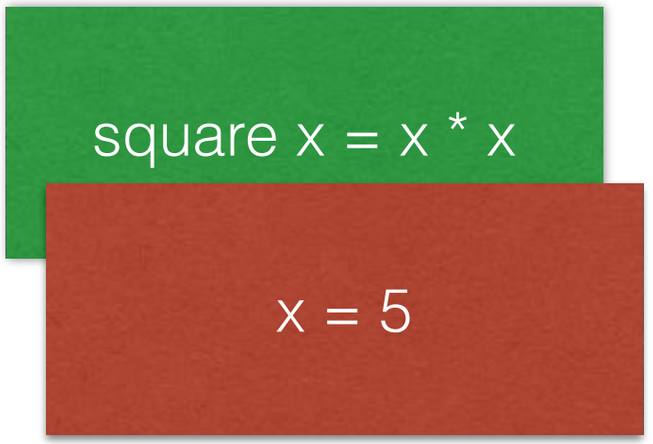
square (x * x)

Step-by-step



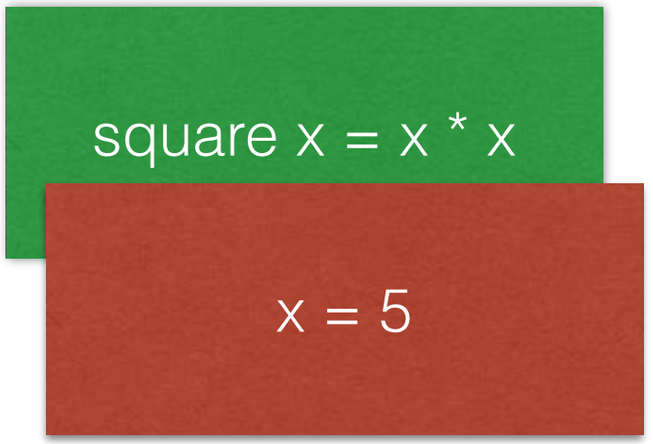
square (**x** * x)

Step-by-step



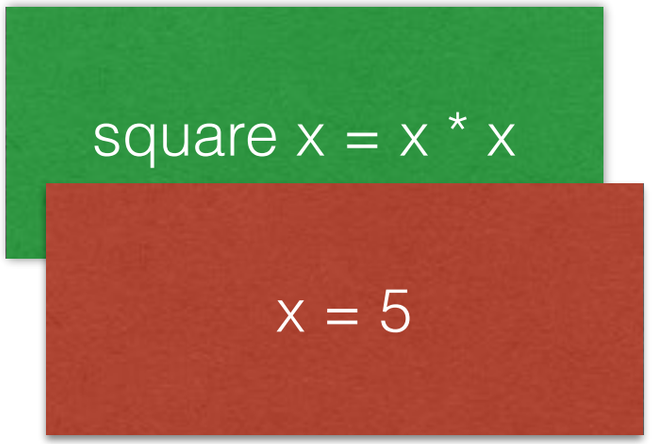
square (5 * x)

Step-by-step



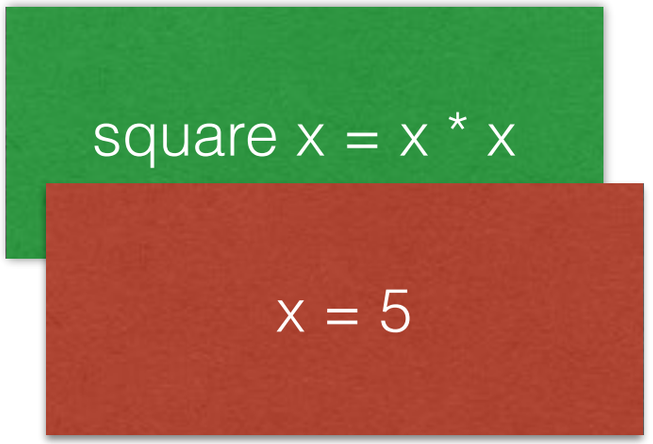
square (5 * **x**)

Step-by-step



square (5 * 5)

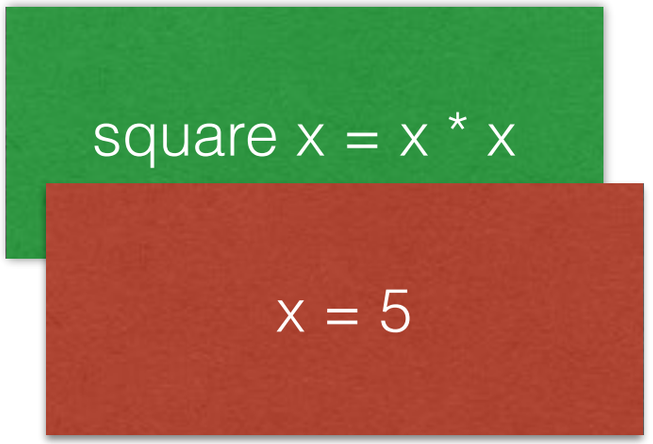
Step-by-step



square (**5 * 5**)

Step-by-step

square (**25**)



Step-by-step

```
square x = x * x
```

```
square 25
```

Step-by-step

```
square x = x * x
```

square 25

Step-by-step

```
square x = x * x
```

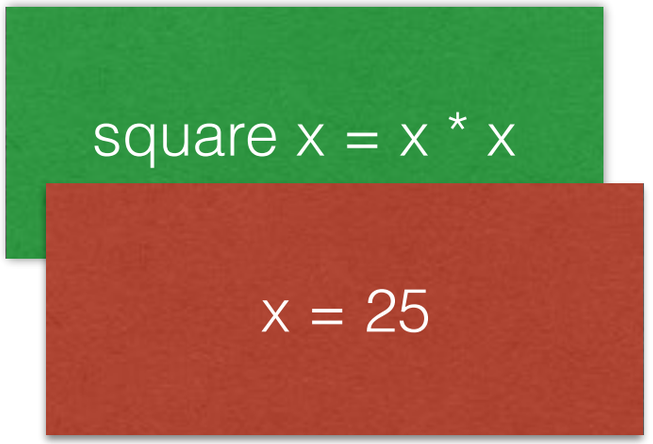
```
let x = 25 in x * x
```

Step-by-step

```
square x = x * x
```

let x = 25 in x * x

Step-by-step



x * x

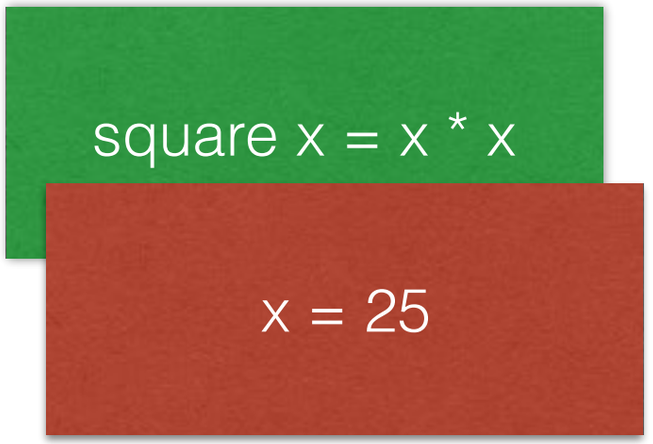
Step-by-step

square x = x * x

x = 25

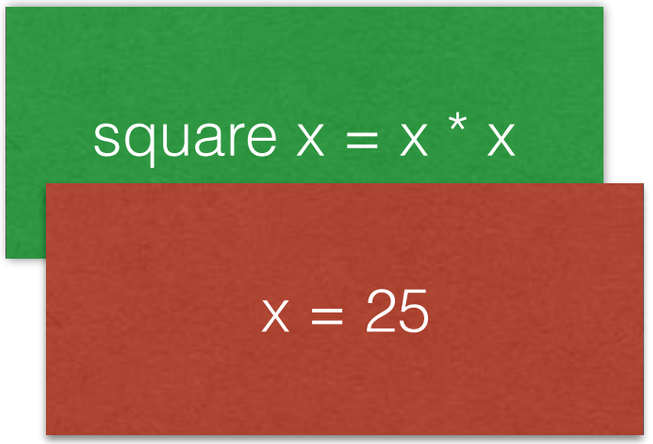
x * x

Step-by-step



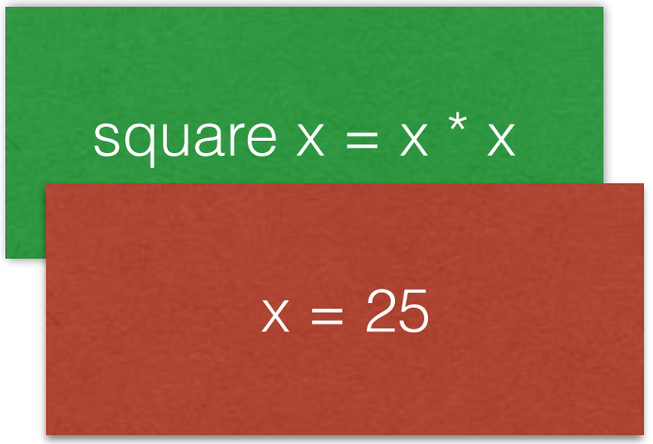
25 * x

Step-by-step



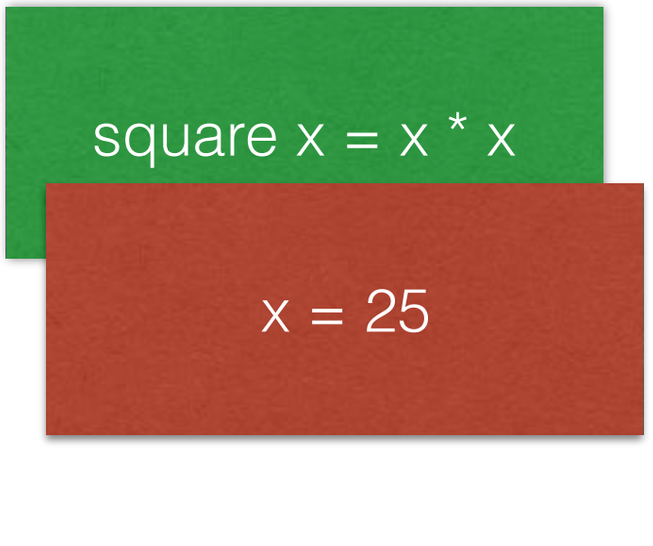
25 * **x**

Step-by-step



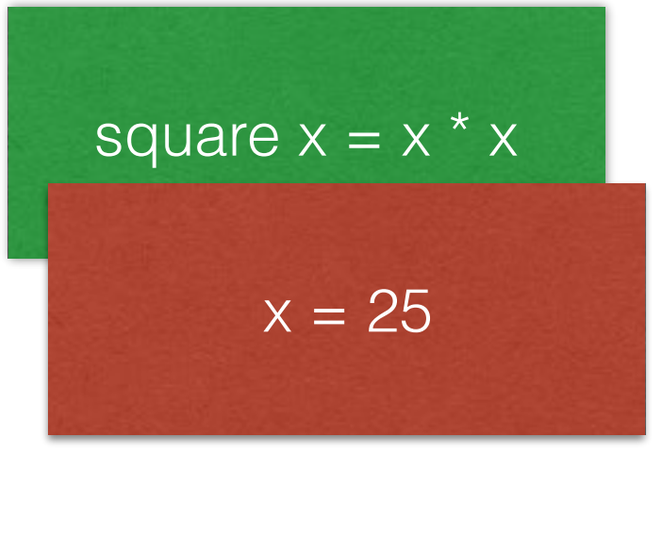
25 * 25

Step-by-step



25 * 25

Step-by-step



625

Test your understanding

http://bit.ly/focs04a

Answers

Functions are (parameterised) expressions (just like math functions)

Test your understanding

bit.ly/focs04b

Answers

```
let f x = x + x in let g h = h (h 1) in g f
let g h = h (h 1) in g f                          [ f(x)=x+x ]
g f                                             [ g(h)=h(h(1)) ]
let h = f in h(h(1))
f(f(1))
f(let x = 1 in x+x)                                  [ x=1 ]
f(x+x)
f(1+1)
f(2)
let x = 2 in x+x                                     [ x=2 ]
x+x
2+2
4
```

"*Anonymous*" functions

```
(fun x → x * x) 25
```

compare with

```
let square x = x * x

square 25
```

Functions are (really) values.

compare with

```
let square = fun x → x * x

square 25
```

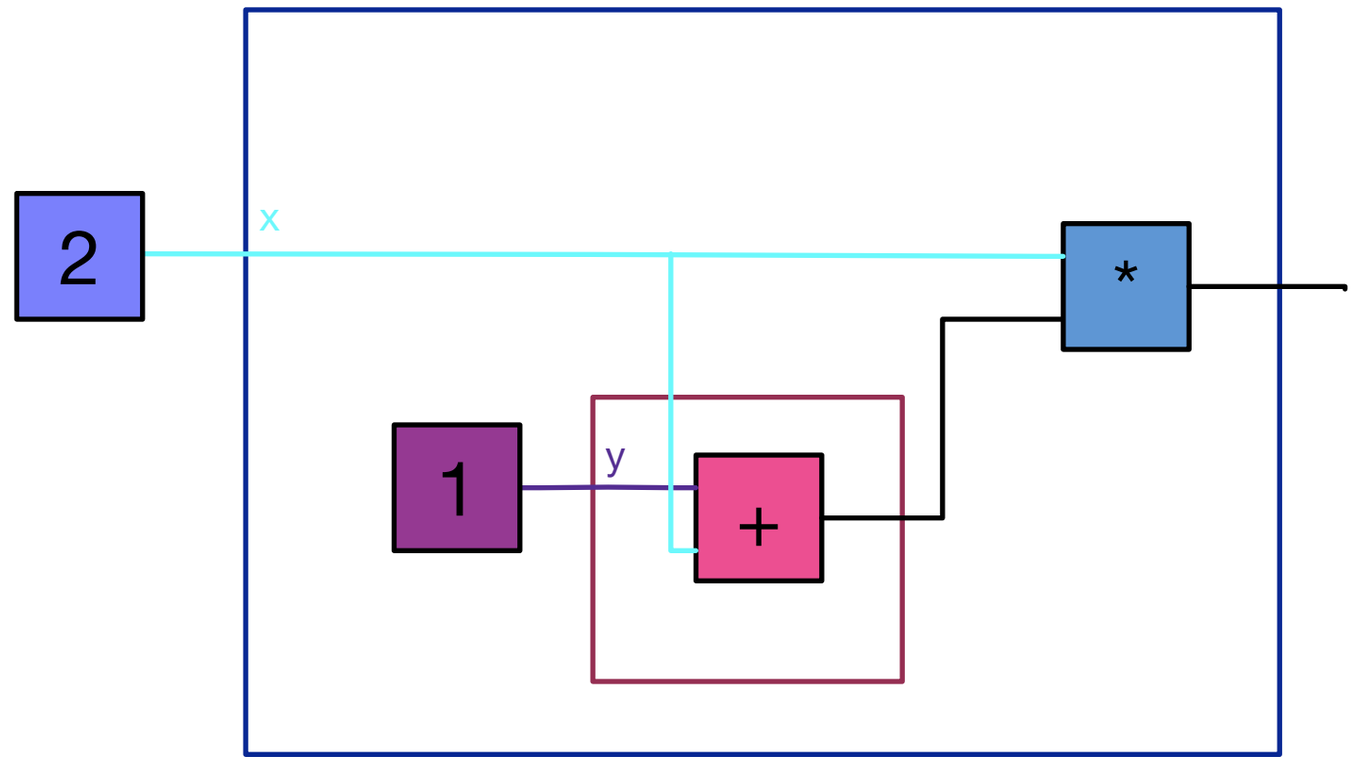
```
(fun x → x + ((fun x → x + x) 1)) 2
(let x = 2 in x + ((fun x → x + x) 1))
(x + ((fun x → x + x) 1))
(2 + ((fun x → x + x) 1))
```

x = 2

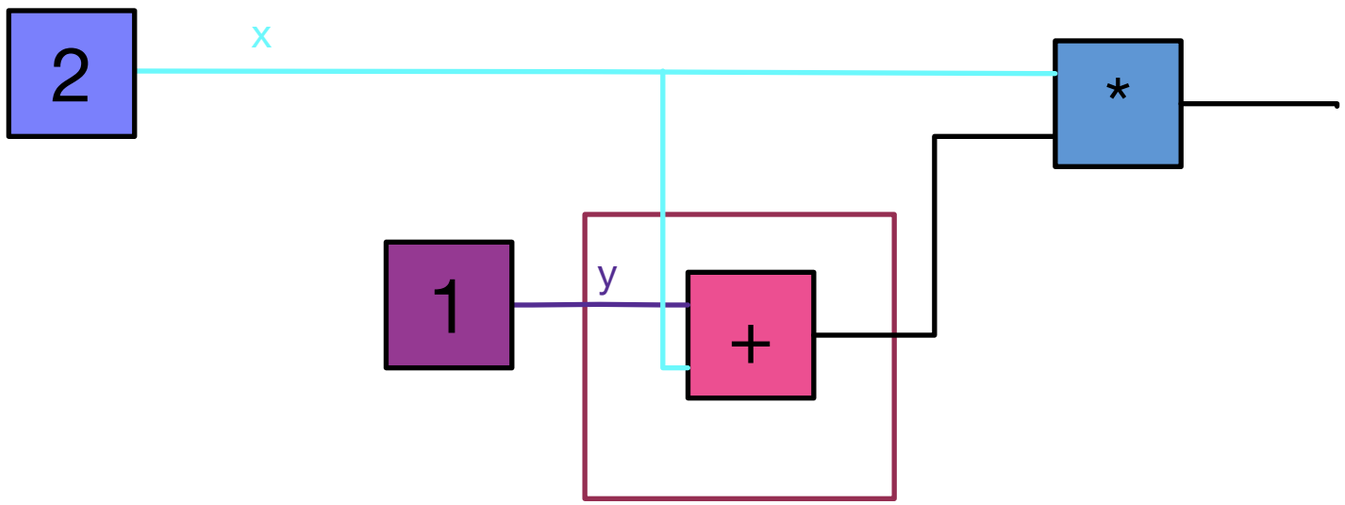
```
(2 + (let x = 1 in x + x))
(2 + (x + x))
(2 + (1 + 1))
4
```

x = 1

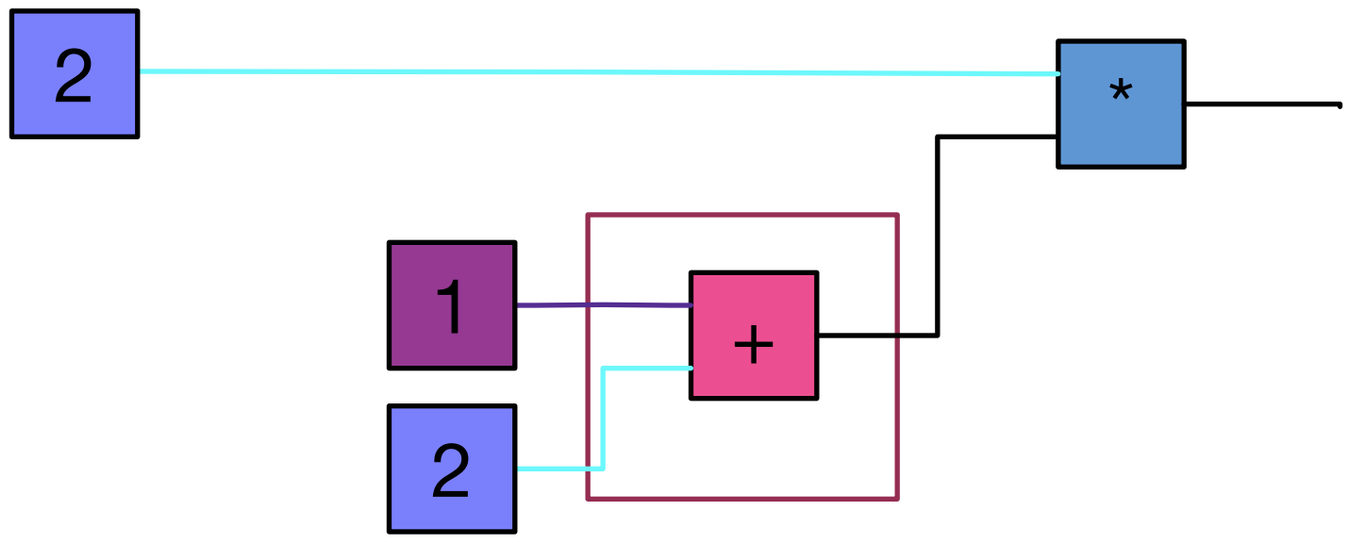
```
(fun x → x * ((fun y → y + x) 1)) 2
```



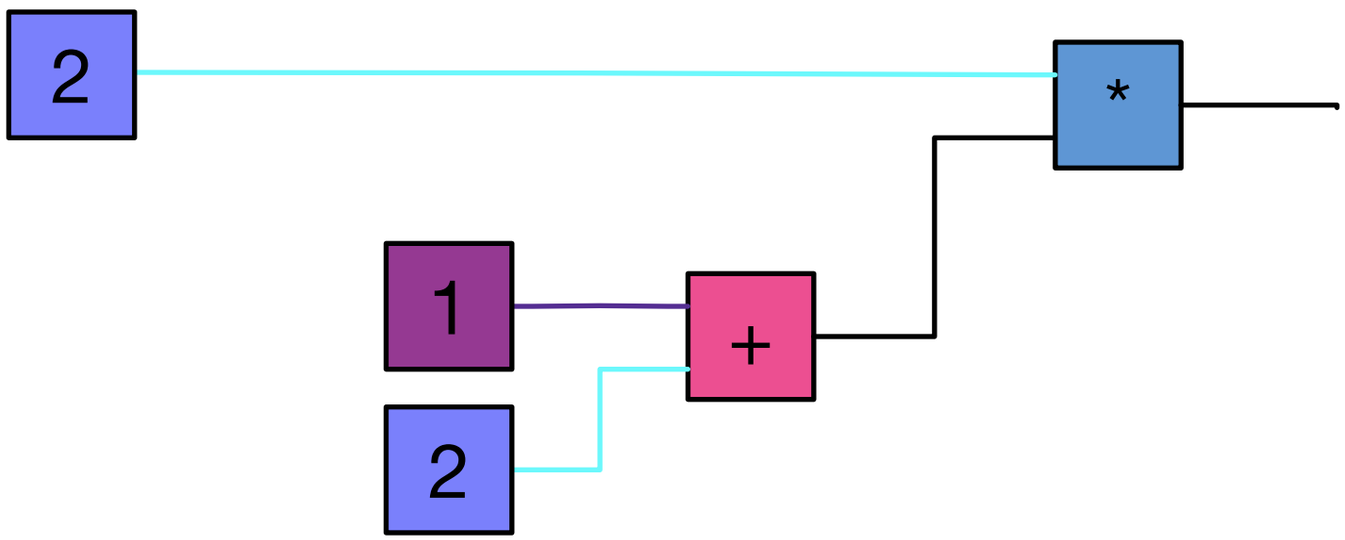
```
(fun x → x * ((fun y → y + x) 1)) 2
```



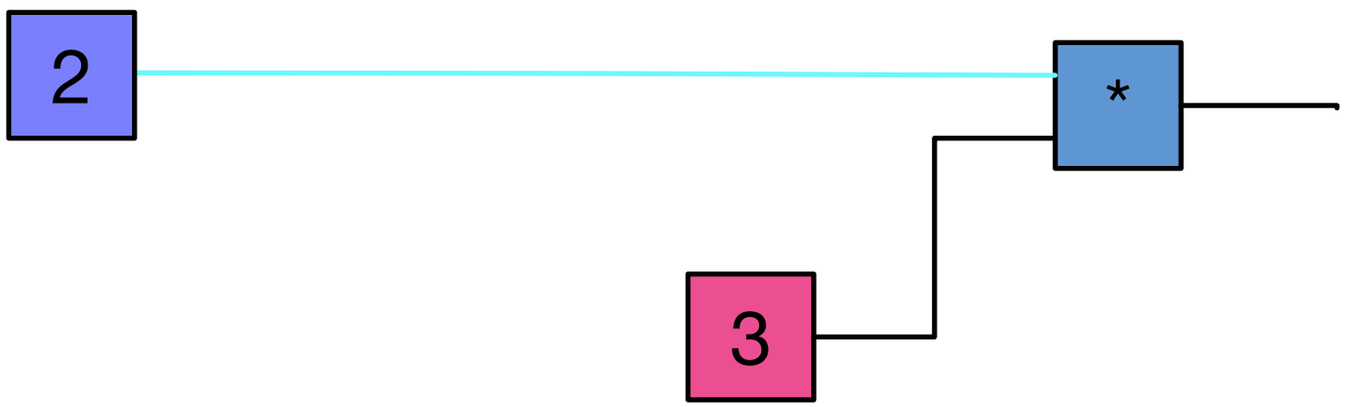
```
(fun x → x * ((fun y → y + x) 1)) 2
```



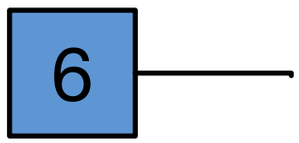
```
(fun x → x * ((fun y → y + x) 1)) 2
```



```
(fun x → x * ((fun y → y + x) 1)) 2
```



```
(fun x → x * ((fun y → y + x) 1)) 2
```



More arguments

```
let sum_sq (x, y) = x * x + y * y;;
let sum_sq x y = x * x + y * y;;
let sum_sq = fun x y -> x*x + y*y;;
let sum_sq = fun x -> (fun y -> x*x + y*y);;
```

Main concepts

- **Evaluation:** “redex” + “substitution”
- **Scope:** where a variable is defined:

```
x + (let y = x + 1 in x + y) + y
```

- **Visibility**: what instance is actually used:

```
x + (let x = x + 3 in x + x) + x
```

same as

```
x + (let z = x + 3 in z + z) + x
```

Note

- these are rather dry and bureaucratic syntactic issues
- quite easy to get wrong!
    - LISP got it wrong!
- we need it more when we *evaluate* programs than when we *write* programs
    - multiples, sometimes nested copies of expressions

Week 2 survey

http://bit.ly/focs04d

Answers