A better title than "functions" for this chapter would be "procedures".

"Top-level functions" are basically fcs which are "global" - they can be called anywhere in the program

[10-15-18]

Recall:

For us, an "interpreter" has 3 main parts:
- abstract syntax
- values
- evaluation ("eval") (thing that takes a thing and returns you a value)

"Formal parameters" ("parameters")

```
int foo(char c, int x) = {...}
int main() = {...
  foo('A', 4);
...}
```

"actual arguments" ("actuals"/"arguments")

} VOCAB ★

Remember, a "first-class function" is a fc which can be treated just like a variable.

◎ Closures

We need a way of defining Functional values

- "A closure combines a functional expression w an environment. It is a value expressed by a functional expression."

Called a "closure" bc usually, in an expression, you have "open", "free", or "unbound" variables. With a closure, you use an environment to "close off" those variables so that they are no longer free or unbound.

Adding an environment to \x = x+y makes a closure

To call a closure, see lecture slides
You don't need to know about environment closure diagrams

◎ Parameter Passing Styles

Call-by-Value (CBV) ("eager evaluation") + Call-by-Name (CBN) ("lazy evaluation")

- a parameter passing convention describes how arguments to functions are evaluated OR describes the order of evaluation of an application and its arguments

ex: (\x -> e) (2+3)

("eager") → CBV would evaluate (2+3)=5 right away (before (\x->e))
("lazy") → CBN would evaluate (2+3)=5 only when its value is needed

⊙ Eager (CBV) Langs vs. Lazy (CBN) Langs.

ex:

```
bomb : Int -> Int
fun bomb n = bomb n
fun one x = 1
```

↳ if we evaluate "one (bomb 99)", :
- eager : goes forever evaluating (bomb 99)
- lazy : evaluates to 1 & never bothers evaluating (bomb 99)

↳ implementation of lazy evaluation; it's lazy ≠ Haskell's lazy.