CSE114A lecture 14

last time: let-expressions in our interpreter

this time: - lambda and application expressions in our interpreter
- recursion in our interpreter
- some announcements

```
let c = 42 in
  let cTimes = \x -> c * x in
    let c = 5 in
      cTimes 2
```

This will evaluate to 84 because Haskell has static scope, aka lexical scope.

If we had dynamic scope, then the value of 'c' that is present at the call site of 'cTimes' is the one we'd use. This makes code hard to understand.

So most languages use static scope.

---

example:

```
e =
let c = 42 in
 let cTimes = \x -> c * x in
  let c = 5 in
   cTimes 2
```

interpretExpr [] e

⇒ interpretExpr [(c, 42)] [let cTimes ... cTimes 2]

⇒ interpretExpr [(cTimes, <closure "x" "c*x" [(c,42)]>)
(c,42)]
let c=5 in cTimes 2

⇒ interpretExpr [(c, 5),
(cTimes, <closure "x", "c*x" [(c,42)] >),
(c,42)]
cTimes 2

$\Rightarrow^*$ <closure "x" "c*x" [(c,42)]> 2

⇒ interpretExpr [(x,2) (c,42)] c*x

$\Rightarrow^*$ 84

What is a closure, exactly?

It's some code, together with an environment (that binds free variables in the code).

For me:

$$\begin{tikzcd} \text{LamValue} & \text{Env} & \text{Id} & \text{Expr} \\ & \text{environment} \arrow[u] & & \text{code} \arrow[u] \end{tikzcd}$$

[(c, 42), (x, 2)]

(And what's an enviroment? Anything that tells you the values of variables.)

We chose a list representation, but we could make other choices.