Growing a Language and Its Interpreter

I01 Language of arithmetic and ***if*** expressions

I02 Absolute value and its ***desugaring***

I03 ***Recursive*** functions implemented using ***substitutions***

I04 ***Environment*** instead of substitutions

I05 ***Higher-order*** functions using substitutions

I06 Higher-order functions using environments

I07 ***Nested recursive*** definitions using environments

I06: Higher-Order Functions Using Environments

Environments are more efficient (and avoid variable capture more easily).
How to use them when parameters can be functions?

Before higher-order functions, environment mapped names to integers.
Now, it maps names to value, which may also be functions:

```
enum Value
  case I(i: BigInt)
  case F(f: Value => Value)

type Env = Map[String, Value]
```

We represent function values of the language we are interpreting using functions in Scala. We say our interpreter is *meta circular* because we use features in meta language in which we write interpreter (Scala) to represent features of the language we are interpreting.

I06: Environment-Based Interpreter is Very Concise!

```
def evalEnv(e: Expr, env: Map[String, Value]): Value = e match
  case C(c) => Value.I(c)
  case N(n) => env.get(n) match
    case Some(v) => v
    case None => evalEnv(defs(n), env)
  case BinOp(op, arg1, arg2) =>
    evalBinOp(op)(evalEnv(arg1,env), evalEnv(arg2,env))
  case IfNonzero(cond, trueE, falseE) =>
    if evalEnv(cond,env) != Value.I(0) then evalEnv(trueE,env)
    else evalEnv(falseE,env)
  case Fun(n,body) => Value.F{(v: Value) =>
    evalEnv(body, env + (n -> v)) } // no danger of capture
  case Call(fun, arg) => evalEnv(fun,env) match
    case Value.F(f) => f(evalEnv(arg,env))
```