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

I04: Environment instead of substitutions

Environments are often more efficient alternative to substitutions.
Instead of copying body of function definition and replacing parameter names with argument constants, we do replacement lazily:

- leave the body as is (no copying!)
- record map from names to argument constants in the environment
- when we find a name, look it up in the environment

I04: Factorial Using Environments

```
fact(3)
|  env: Map(n -> 3)
|  (if n then (* n (fact (- n 1))) else 1)        // body as declared
|  fact(2)
|  |  env: Map(n -> 2)
|  |  (if n then (* n (fact (- n 1))) else 1)     // same
|  |  fact(1)
|  |  |  env: Map(n -> 1)
|  |  |  (if n then (* n (fact (- n 1))) else 1)  // still same
|  |  |  fact(0)
|  |  |  |  env: Map(n -> 0)
|  |  |  |  (if n then (* n (fact (- n 1))) else 1)  // again same!
|  |  |  +--> 1
|  |  +--> 1
|  +--> 2
+--> 6
```

I03: Evaluation Using Environment

```
def evalE(e: Expr, env: Map[String, BigInt]): BigInt = e match
  case C(c) => c
  case N(n) => env(n)  // look up name in the environment
  case BinOp(op, e1, e2) =>
    evalBinOp(op)(evalE(e1, env), evalE(e2, env))
  case IfNonzero(cond, trueE, falseE) =>
    if evalE(cond,env) != 0 then evalE(trueE,env)
    else evalE(falseE,env)
  case Call(fName, args) =>
    defs.get(fName) match
      case Some(f) =>
        val evaledArgs = args.map((e: Expr) => evalE(e,env))
        // newEnv additionally maps parameters to arguments
        val newEnv = env ++ f.params.zip(evaledArgs)
        evalE(f.body, newEnv)
```