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

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

So far we used a special global environment defs to express recursion
We could create locally anonymous functions, but without a way to call them recursively.
In this step of the interpreter, we introduce the Defs expression case for adding (nested, local) mutually recursive functions:

```
enum Expr
  case C(c: BigInt)
  case N(name: String)
  case IfNonzero(cond: Expr, trueE: Expr, falseE: Expr)
  case Call(function: Expr, arg: Expr)
  case Fun(param: String, body: Expr)
  case Defs(defs: List[(String, Expr)], rest: Expr)
```

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

So far we used a special global environment defs to express recursion
We could create locally anonymous functions, but without a way to call them recursively.
In this step of the interpreter, we introduce the Defs expression case for adding (nested, local) mutually recursive functions:

```
enum Expr
  case C(c: BigInt)
  case N(name: String)
  case IfNonzero(cond: Expr, trueE: Expr, falseE: Expr)
  case Call(function: Expr, arg: Expr)
  case Fun(param: String, body: Expr)
  case Defs(defs: List[(String, Expr)], rest: Expr)

type Env = String => Option[Value]
```

I07: Eval for Nested Recursive Definitions: Key Cases

```
def evalEnv(e: Expr, env: Env): Value = e match ...
  case N(n) => env(n) match // no use of defs, only env
    case Some(v) => v
  case Fun(n,body) => Value.F{(v: Value) => // same as before
    val env1: String => Option[Value] =
      (s:String) => if s==n then Some(v) else env(s)
    evalEnv(body, env1) }
  case Call(fun, arg) => evalEnv(fun,env) match // same
    case Value.F(f) => f(evalEnv(arg,env))
  case Defs(defs, rest) => //
    def env1: Env = // extended environment
      (s:String) =>
        lookup(defs, s) match // list lookup in local defs
          case None => env(s) // fall back to outer scope
          case Some(body) => Some(evalEnv(body, env1)) // rec
    evalEnv(rest, env1)
```

I07: What Behavior Would We Get with This Version

```
def evalEnv(e: Expr, env: Env): Value = e match ...
  case N(n) => env(n) match // no use of defs, only env
    case Some(v) => v
  case Fun(n,body) => Value.F{(v: Value) => // same as before
    val env1: String => Option[Value] =
      (s:String) => if s==n then Some(v) else env(s)
    evalEnv(body, env1) }
  case Call(fun, arg) => evalEnv(fun,env) match // same
    case Value.F(f) => f(evalEnv(arg,env))
  case Defs(defs, rest) => //
    def env1: Env = // extended environment
      (s:String) =>
        lookup(defs, s) match // list lookup in local defs
          case None => env(s) // fall back to outer scope
          case Some(body) => Some(evalEnv(body, env)) // nonrec
    evalEnv(rest, env1)
```

I07: What Behavior Would We Get with This Version

```
def evalEnv(e: Expr, env: Env): Value = e match ...
  case Defs(defs, rest) => //
    def env1: Env = // extended environment
      (s:String) =>
        lookup(defs, s) match // list lookup in local defs
          case None => env(s) // fall back to outer scope
          case Some(body) => Some(evalEnv(body, env)) // nonrec
    evalEnv(rest, env1)

(def fact = (n => (if n then (* n (fact (- n 1))) else 1))
 (fact 6))

java.lang.Exception: Unknown name 'fact' in top-level environment
```

I07: Must Make env1 Recursive

```
def evalEnv(e: Expr, env: Env): Value = e match ...
  case Defs(defs, rest) => //
    def env1: Env = // extended environment
      (s:String) =>
        lookup(defs, s) match // list lookup in local defs
          case None => env(s) // fall back to outer scope
          case Some(body) => Some(evalEnv(body, env1)) // rec
    evalEnv(rest, env1)

(def fact = (n => (if n then (* n (fact (- n 1))) else 1))
 (fact 6))

~~> 720
```

I07: Initial Environment Replaces BinOp-s

We start evaluation in the initial environment:

```
evalEnv(e, initEnv)
```

```
val initEnv: Env =
  (s:String) => s match
    case "+"  => lift2int(_ + _)
    case "-"  => lift2int(_ - _)
    case "*"  => lift2int(_ * _)
    case "^"  => lift2int((x:BigInt,y:BigInt) => x.pow(y.toInt))
    case "<=" => lift2int(
      (x:BigInt,y:BigInt) => if x <= y then 1 else 0)
    case _ => error(s"Unknown name '$s' in initial environment")
```

We no longer need BinOp as special expression form

I07: Lifting Binary Functions to Work on Values

```
def lift2int(f: (BigInt, BigInt) => BigInt): Option[Value] =
  import Value._
  Some(F(
    (v1:Value) => F(
      (v2:Value) => {
        (v1,v2) match
          case (I(i1),I(i2)) => I(f(i1,i2))
          case _ => error("Wrong operator type")
      })))
```