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

I05: Higher-Order Functions Using Substitution

```
def twice =  (f => (x => (f (f x))))
def square = (x => (* x x))

((twice square) 3)
|  (twice square)
|  FUN: (f => (y => (f (f y))))  ARG: (x => (* x x))
|  (y => ((x => (* x x)) ((x => (* x x)) y)))
|  +--> (y => ((x => (* x x)) ((x => (* x x)) y)))
FUN: (y => ((x => (* x x)) ((x => (* x x)) y)))  ARG: 3
((x => (* x x)) ((x => (* x x)) 3))
|  ((x => (* x x)) ((x => (* x x)) 3))
|  |  ((x => (* x x)) 3)
|  |  FUN: (x => (* x x))  ARG: 3
|  |  (* 3 3)
|  |  +--> 9
|  FUN: (x => (* x x))  ARG: 9
|  (* 9 9)
|  +--> 81
+--> 81
```

I05: Trees for Higher-Order Functions

Now we have a case for creating function anywhere in the expression (param => body)
Argument of function call need not be a name but can be an expression
A function has exactly one argument (use currying if needed)

```
enum Expr
  case C(c: BigInt)
  case N(name: String)
  case BinOp(op: BinOps, e1: Expr, e2: Expr)
  case IfNonzero(cond: Expr, trueE: Expr, falseE: Expr)
  case Call(fun: Expr, arg: Expr)     // fun can be expression itself
  case Fun(param: String, body: Expr) // param => body
```

```
Call(Fun("x", BinOp(Times, N("x"), N("x"))),  //  x => (* x x)
     C(3))                                    //  3
```

I05: Eval Using Substitution

Result can be a function, so we return an Expr (not BigInt)

```
def eval(e: Expr): Expr = e match
  case C(c) => e
  case N(n) => eval(defs(n)) // find in global defs, then eval
  case BinOp(op, e1, e2) =>
    evalBinOp(op)(eval(e1), eval(e2))
  case IfNonzero(cond, trueE, falseE) =>
    if eval(cond) != C(0) then eval(trueE)
    else eval(falseE)
  case Fun(_,_) => e  // functions evaluate to themselves
  case Call(fun, arg) =>
    eval(fun) match
      case Fun(n,body) => eval(subst(body, n, eval(arg)))
```

I05: Substitution on Trees for Higher-Order Functions

```
// substitute all n with r in expression e
def subst(e: Expr, n: String, r: Expr): Expr = e match
  case C(c) => e
  case N(s) => if s==n then r else e
  case BinOp(op, e1, e2) =>
       BinOp(op, subst(e1,n,r), subst(e2,n,r))
  case IfNonzero(cond, trueE, falseE) =>
       IfNonzero(subst(cond,n,r), subst(trueE,n,r), subst(falseE,n,r))
  case Call(f, arg) =>
       Call(subst(f,n,r), subst(arg,n,r))
  case Fun(formal,body) =>
       if formal==n then e  // do not substitute under (n => ...)
       else Fun(formal, subst(body,n,r))
```

I05: More Examples: Twice Factorial

```
(def twice =  (f => x => (f (f x)))
 def fact n = (if n then (* n (fact (- n 1))) else 1)
 (twice fact 3))
~~> 720
```

I05: More Examples: Twice Factorial

```
(def twice =  (f => x => (f (f x)))
 def fact n = (if n then (* n (fact (- n 1))) else 1)
 (twice fact 3))
~~> 720

(def twice1 =  (f => fact => (f (f fact)))
 def fact n = (if n then (* n (fact (- n 1))) else 1)
 (twice1 fact 3))
```

I05: More Examples: Twice Factorial

```
(def twice =  (f => x => (f (f x)))
 def fact n = (if n then (* n (fact (- n 1))) else 1)
 (twice fact 3))
~~> 720

(def twice1 =  (f => fact => (f (f fact)))
 def fact n = (if n then (* n (fact (- n 1))) else 1)
 (twice1 fact 3))

  FUN: (f => (fact => (f (f fact))))
  ARG: (n => (if n then (* n (fact (- n 1))) else 1))
  FUN: (fact => ((n => (if n then (* n (fact (- n 1))) else 1))
              ((n => (if n then (* n (fact (- n 1))) else 1)) fact)))
  ARG: 3
  (if 3 then (* 3 (3 (- 3 1))) else 1)
  (3 (- 3 1))
 java.lang.Exception: Cannot apply non-function 3 in a call
```