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

I03: Recursive Functions

We would like to handle examples like this one:

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

What do we need to add to our abstract syntax trees?

I03: Recursive Functions

We would like to handle examples like this one:

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

What do we need to add to our abstract syntax trees?

- names inside expressions to refer to parameters (`n`)
- calls to user-defined functions (`fact 6`)
- definitions (map function names to parameters and function bodies)

I03: Recursive Function Definitions: Trees and Factorial Example

```
enum Expr
  case C(c: BigInt)
  case N(name: String)   // immutable variable
  case BinOp(op: BinOps, e1: Expr, e2: Expr)
  case IfNonzero(cond: Expr, trueE: Expr, falseE: Expr)
  case Call(function: String, args: List[Expr])   // function call

case class Function(params: List[String], body: Expr)
type DefEnv = Map[String, Function]  // function names to definitions

val defs : DefEnv = Map[String, Function](
  "fact" -> Function(List("n"), // formal parameter "n", body:
    IfNonzero(N("n"),
              BinOp(Times, N("n"),
                    Call("fact", List(BinOp(Minus,(N("n"), C(1)))))),
              C(1)))
) // if n then (* n (fact (- n 1))) else 1
```

I03: Idea of Evaluation Based Using Substitution

- evaluate arguments so they become constants
- look up function body, replace formal parameters with constants
- evaluate replaced function body

```
def fact n = (if n then (* n (fact (- n 1))) else 1)
(fact 3)
(if 3 then (* 3 (fact (- 3 1))) else 1)
|  (fact 2)
|  (if 2 then (* 2 (fact (- 2 1))) else 1)
|  |  (fact 1)
|  |  (if 1 then (* 1 (fact (- 1 1))) else 1)
|  |  |  (fact 0)
|  |  |  (if 0 then (* 0 (fact (- 0 1))) else 1)
|  |  |  +--> 1
|  |  +--> 1
|  +--> 2
+--> 6
```

I03: eval Using Substitution

```
def eval(e: Expr): BigInt = e match
  case C(c) => c
  case N(n) => error(s"Unknown name '$n'") // should never occur
  case BinOp(op, e1, e2) =>
    evalBinOp(op)(eval(e1), eval(e2))
  case IfNonzero(cond, trueE, falseE) =>
    if eval(cond) != 0 then eval(trueE)
    else eval(falseE)
  case Call(fName, args) => // the only new case we handle
    defs.get(fName) match // defs is a global map with all functions
      case Some(f) => // f has body:Expr and params:List[String]
        val evaledArgs = args.map((e: Expr) => C(eval(e)))
        val bodySub = substAll(f.body, f.params, evaledArgs)
        eval(bodySub)   // may contain further recursive calls
                        // bodySub should no longer have N(...)
```

I03: Substitution

```
// 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(c, trueE, falseE) =>
       IfNonzero(subst(c,n,r), subst(trueE,n,r), subst(falseE,n,r))
  case Call(f, args) =>
       Call(f, args.map(subst(_,n,r)))

def substAll(e: Expr, names: List[String],
                replacements: List[Expr]): Expr  =
  (names, replacements) match
     case (n :: ns, r :: rs) => substAll(subst(e,n,r), ns, rs)
     case _ => e
```

I03: Division Example and Wrap Up

```
def div x y =
  (if (<= y x) then (+ 1 (div (- x y) y)) else 0)

(div 15 6)
(if (<= 6 15) then (+ 1 (div (- 15 6) 6)) else 0)
|  (div 9 6)
|  (if (<= 6 9) then (+ 1 (div (- 9 6) 6)) else 0)
|  |  (div 3 6)
|  |  (if (<= 6 3) then (+ 1 (div (- 3 6) 6)) else 0)
|  |  +--> 0
|  +--> 1
+--> 2
```

I03: Division Example and Wrap Up

```
def div x y =
  (if (<= y x) then (+ 1 (div (- x y) y)) else 0)

(div 15 6)
(if (<= 6 15) then (+ 1 (div (- 15 6) 6)) else 0)
|  (div 9 6)
|  (if (<= 6 9) then (+ 1 (div (- 9 6) 6)) else 0)
|  |  (div 3 6)
|  |  (if (<= 6 3) then (+ 1 (div (- 3 6) 6)) else 0)
|  |  +--> 0
|  +--> 1
+--> 2
```

This completes the interpreter for recursive computable functions. Every computable function that maps an n-tuple of integers into an integer can be described in it and our interpreter can execute it! We can even encode data structures as large integers.