Implementing a Simple Programming Language

Functional Programing (CS-210)

EPFL

Simple Untyped Functional Language

Example program:

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

evaluates to:

Simple Untyped Functional Language

Example program:

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

evaluates to: **720**

Simple Untyped Functional Language

Example program:

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

evaluates to: **720**

```
(
  def twice =  (f => x => (f (f x)))
  def square = (x => (* x x))
  (twice square 3)
)
```

evaluates to:

Simple Untyped Functional Language

Example program:

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

evaluates to: **720**

```
(
  def twice =  (f => x => (f (f x)))
  def square = (x => (* x x))
  (twice square 3)
)
```

evaluates to: **81**

Program Representation: Abstract Syntax Trees

```
(def twice =  (f => x => (f (f x)))
 def square = (x => (* x x))
 (twice square 3))
```

≈

```
val defs : DefEnv = Map[String, Expr](
  "twice" -> Fun("f", Fun("x",
                          Call(N("f"), Call(N("f"), N("x"))))),
  "square" -> Fun("x", BinOp(Times, N("x"), N("x"))))
val expr = Call(Call(N("twice"), N("square")), C(3))
```

- We represent a program using ***expression tree*** called Abstract Syntax Tree (AST)
- Our implementation is an ***interpreter***, which traverses AST to produce the result
- We discuss later briefly how to convert an input file into an abstract syntax tree; more on that in the course ***Computer Language Processing (CS-320)*** next year

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

I01. Language of arithmetic and *if* expressions: Trees

Integer constants combined using arithmetic operations and the `if` conditional

```
val expr1 = BinOp(Times, C(6), C(7))          // 6 * 7
val cond1 = BinOp(LessEq, expr1, C(50))       // expr1 <= 50
val expr2 = IfNonzero(cond1, C(10), C(20))    // if (cond1) 10 else 20
```

How to describe such trees?

I01. Language of arithmetic and *if* expressions: Trees

Integer constants combined using arithmetic operations and the `if` conditional

```
val expr1 = BinOp(Times, C(6), C(7))         // 6 * 7
val cond1 = BinOp(LessEq, expr1, C(50))      // expr1 <= 50
val expr2 = IfNonzero(cond1, C(10), C(20))   // if (cond1) 10 else 20
```

How to describe such trees?

```
enum Expr
  case C(c: BigInt)                           // integer constant
  case BinOp(op: BinOps, e1: Expr, e2: Expr) // binary operation
  case IfNonzero(cond: Expr, trueE: Expr, falseE: Expr)

enum BinOps
  case Plus, Minus, Times, Power, LessEq
```

I01. Language of arithmetic and *if* expressions: Printing

```
def str(e: Expr): String = e match
  case C(c) => c.toString
  case BinOp(op, e1, e2) =>
    s"(${strOp(op)} ${str(e1)} ${str(e2)})" // string interpolation
  case IfNonzero(cond, trueE, falseE) =>
    s"(if ${str(cond)} then ${str(trueE)} else ${str(falseE)})"

def strOp(op: BinOps): String = op match
  case Plus   => "+"
  case Minus  => "-"
  case Times  => "*"
  case Power  => "^"
  case LessEq => "<="

> str(IfNonzero(BinOp(LessEq, C(4), C(50)), C(10), C(20)))
(if (<= 4 50) then 10 else 20)
```

I01. Language of arithmetic and *if* expressions: Interpreting

```
def eval(e: Expr): BigInt = e match
  case C(c) => c
  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)

def evalBinOp(op: BinOps)(x: BigInt, y: BigInt): BigInt = op match
  case Plus => x + y
  case Minus => x - y
  case Times => x * y
  case Power => x.pow(y.toInt)
  case LessEq => if (x <= y) 1 else 0

> eval(IfNonzero(BinOp(LessEq, C(4), C(50)), C(10), C(20)))
10
```

I02. Absolute Value and Its ***Desugaring***: Trees

```
enum Expr
  case C(c: BigInt)
  case BinOp(op: BinOps, e1: Expr, e2: Expr)
  case IfNonzero(cond: Expr, trueE: Expr, falseE: Expr)
  case AbsValue(arg: Expr)   // new case
```

How to extend evaluator to work with absolute value as well? Two approaches:

- add a case to the interpreter (exercise)
- transform (desugar) trees to reduce them to previous cases

Syntactic sugar = extra language constructs that are not strictly necessary because they can be expressed in terms of others (they make the language sweeter to use)

Desugaring = automatically eliminating syntactic sugar by expanding constructs

I02. Desugaring Absolute Value: Idea

By definition of absolute value, we would like this equality to hold:

```
abs x   ≡   if (<= x 0) then (- 0 x) else x
```

that is, at the level of AST,

```
AbsValue(x) ⇝
   IfNonzero(BinOp(LessEq, x, C(0)),
             BinOp(Minus, C(0), x),
             x)
```

How to write `desugar` function that eliminates all occurrences of `AbsValue`?

I02. Desugaring Absolute Value: Idea

By definition of absolute value, we would like this equality to hold:

```
abs x   ≡   if (<= x 0) then (- 0 x) else x
```

that is, at the level of AST,

```
AbsValue(x) ⇝
   IfNonzero(BinOp(LessEq, x, C(0)),
             BinOp(Minus, C(0), x),
             x)
```

How to write `desugar` function that eliminates all occurrences of `AbsValue`?

Replace (recursively) each subtree `AbsValue(x)` with its definition.

I02. Desugaring Absolute Value: Code

```
def desugar(e: Expr): Expr = e match
  case C(c) => e
  case BinOp(op, e1, e2) =>
       BinOp(op, desugar(e1), desugar(e2))
  case IfNonzero(cond, trueE, falseE) =>
       IfNonzero(desugar(cond), desugar(trueE), desugar(falseE))
  case AbsValue(arg) =>
    val x = desugar(arg)
    IfNonzero(BinOp(LessEq, x, C(0)),
              BinOp(Minus, C(0), x),
              x)
```

I02. Desugaring Absolute Value: Example Run

```
def show(e: Expr): Unit =
  println("original:")
  println(str(e))
  val de = desugar(e)
  println("desugared:")
  println(str(de))
  println(" ~~> " + eval(de) + "\n")

show(AbsValue(BinOp(Plus,C(10),C(-50))))
```

```
original:
(abs (+ 10 -50))
desugared:
(if (<= (+ 10 -50) 0) then (- 0 (+ 10 -50)) else (+ 10 -50))
  ~~> 40
```

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.

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)
```

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
```

I05: Variable Capture

```
def subst(e: Expr, n: String, r: Expr): Expr = e match
  ...
  case Fun(formal,body) =>
       if formal==n then e  // do not substitute under (n => ...)
       else Fun(formal, subst(body,n,r))
```

The last line exhibits ***variable capture problem***.
If 'formal' occurs free in 'r', then it will be captured by Fun(formal,...) even though that outside occurrence of 'formal' in 'r' has nothing to do with the bound variable in the anonymous function.

```
  FUN: (f => (fact => (f (f fact))))
  ARG: (n => (if n then (* n (fact (- n 1))) else 1))
fact => ((n => (if n then (* n (fact (- n 1))) else 1))
           ((n => (if n then (* n (fact (- n 1))) else 1))  fact))
```

When we supply the integer argument we will also substitute it instead of the name of the factorial function, resulting in run-time error (or, in other cases, wrong result).

I05: We Want to Rename Bound Variable to Avoid Capture

In situation like this:

```
FUN: (f => (fact => (f (f fact))))
ARG: (n => (if n then (* n (fact (- n 1))) else 1))
```

when substituting ARG inside the body of FUN, we first rename bound variable in FUN body into one that does not occur in ARG:

```
FUN: (f => (fact' => (f (f fact'))))
ARG: (n => (if n then (* n (fact (- n 1))) else 1))
```

Instead of fact' we can choose any fresh name for bound variable!
Result then evaluates correctly:

```
fact' => ((n => (if n then (* n (fact (- n 1))) else 1))
          ((n => (if n then (* n (fact (- n 1))) else 1))  fact'))
```

I05: Renaming Bound Variables in Subst

We call our previous substitution ***naive substitution***:

```
def subst0(e: Expr, n: String, r: Expr): Expr = e match ...
  case Fun(formal,body) =>
       if formal==n then e else Fun(formal, subst0(body,n,r))
```

To avoid problems, we use ***capture-avoiding substitution***:

```
def subst(e: Expr, n: String, r: Expr): Expr = e match ...
  case Fun(formal,body) =>
       if formal==n then e else
         val fvs = freeVars(r)
         val (formal1, body1) =
           if fvs.contains(formal) then // rename bound formal
             val formal1 = differentName(formal, fvs)
             (formal1, subst0(body, formal, N(formal1)))
           else (formal, body)
         Fun(formal1, subst(body1,n,r)) // substitute either way
```

l05: Free Variables and Finding a Different Name

```
def differentName(n: String, s: Set[String]): String =
  if s.contains(n) then differentName(n + "'", s)
  else n

def freeVars(e: Expr): Set[String] = e match
  case C(c) => Set()
  case N(s) => Set(s)
  case BinOp(op, e1, e2) => freeVars(e1) ++ freeVars(e2)
  case IfNonzero(cond, trueE, falseE) =>
       freeVars(cond) ++ freeVars(trueE) ++ freeVars(falseE)
  case Call(f, arg) => freeVars(f) ++ freeVars(arg)
  case Fun(formal,body) => freeVars(body) - formal
```

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))
```

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")
      })))
```

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: Write Tail-Recursive Division with Nested Auxiliary Function

I07: Write Tail-Recursive Division with Nested Auxiliary Function

```
val lDefs = List[(String, Expr)](
  "nDiv" -> Fun("xCurrent", Fun("acc",
     IfNonzero(leq(N("y"), N("xCurrent")),
       Call(Call(N("nDiv"), minus(N("xCurrent"), N("y"))),
            plus(C(1), N("acc"))),
       N("acc"))))
)
val gDefs = List[(String, Expr)](
  "tDiv" -> Fun("x", Fun("y",
     Defs(lDefs, Call(Call(N("nDiv"), N("x")), C(0)))))
)
val expr = Defs(gDefs,
                Call(Call(N("tDiv"), C(720)), C(10)))
```

Pretty Printing

```
val lDefs = List(
  "nDiv" -> Fun("xCurrent", Fun("acc",
     IfNonzero(leq(N("y"), N("xCurrent")),
       Call(Call(N("nDiv"), minus(N("xCurrent"), N("y"))),
            plus(C(1), N("acc"))),
       N("acc"))))
)
val tDivExpr =
      Defs(List("tDiv" -> Fun("x", Fun("y",
                          Defs(lDefs, Call(Call(N("nDiv"), N("x")), C(0)))))),
           Call(Call(N("tDiv"), C(720)), C(10)))
```

⇓ **PRINT**

```
(def tDiv = (x => y =>
              (def nDiv = (xCurrent => acc =>
                  (if <= y xCurrent then nDiv (- xCurrent y) (+ 1 acc)
                                    else acc))
              nDiv x 0))
 tDiv 720 10)
```

Parsing

```
val lDefs = List(
  "nDiv" -> Fun("xCurrent", Fun("acc",
     IfNonzero(leq(N("y"), N("xCurrent")),
       Call(Call(N("nDiv"), minus(N("xCurrent"), N("y"))),
            plus(C(1), N("acc"))),
       N("acc"))))
)
val tDivExpr =
      Defs(List("tDiv" -> Fun("x", Fun("y",
                          Defs(lDefs, Call(Call(N("nDiv"), N("x")), C(0)))))),
           Call(Call(N("tDiv"), C(720)), C(10)))
```

⇑ **PARSE**

```
(def tDiv = (x => y =>
              (def nDiv = (xCurrent => acc =>
                  (if <= y xCurrent then nDiv (- xCurrent y) (+ 1 acc)
                                    else acc))
              nDiv x 0))
 tDiv 720 10)
```

Parsing

```
val lDefs = List(
  "nDiv" -> Fun("xCurrent", Fun("acc",
     IfNonzero(leq(N("y"), N("xCurrent")),
       Call(Call(N("nDiv"), minus(N("xCurrent"), N("y"))),
            plus(C(1), N("acc"))),
       N("acc"))))
)
val tDivExpr =
      Defs(List("tDiv" -> Fun("x", Fun("y",
                          Defs(lDefs, Call(Call(N("nDiv"), N("x")), C(0)))))),
           Call(Call(N("tDiv"), C(720)), C(10)))
```

⇑ **PARSE**

```
(def tDiv = (x => y =>
              (def nDiv = (xCurrent => acc =>
                  (if <= y xCurrent then nDiv (- xCurrent y) (+ 1 acc)
                                    else acc))
              nDiv x 0))
 tDiv 720 10)
```

ScaLL1on Library by Romain Edelmann

More about Parsing for Interpreters and Compilers

Computer Language Processing, 3rd year EPFL Course (CS-320)

`https://lara.epfl.ch/w/cc19:top`

Current Interpreter vs Lambda Calculus

```
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)
```

We now make language smaller, but without losing expressive power!

Current Interpreter vs Lambda Calculus

```
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)
```

We now make language smaller, but without losing expressive power!
We show that we only need these three constructs:

```
enum Expr
  case N(name: String)
  case Call(function: Expr, arg: Expr)
  case Fun(param: String, body: Expr)
```

The higher-order language with only these three constructs is called **lambda calculus**.

Encoding Recursion: Dummy Parameter

We show that recursion can be encoded using higher-order functions.
Consider a recursive factorial function definition:

```
(def fact = (n =>
               if n then * n (fact (- n 1)) else 1)
 fact 10)
```

Encoding Recursion: Dummy Parameter

We show that recursion can be encoded using higher-order functions.
Consider a recursive factorial function definition:

```
(def fact = (n =>
               if n then * n (fact (- n 1)) else 1)
 fact 10)
```

Let us add an extra parameter called to factorial which we will call 'self'.

Encoding Recursion: Dummy Parameter

We show that recursion can be encoded using higher-order functions.
Consider a recursive factorial function definition:

```
(def fact = (n =>
               if n then * n (fact (- n 1)) else 1)
 fact 10)
```

Let us add an extra parameter called to factorial which we will call 'self'.
It initially serves no purpose because we just propagate it without ever using it:

```
(def factGen = (self => n =>
                  if n then * n (factGen self (- n 1)) else 1)
 factGen factGen 10)
```

It does not matter what we give as the `self` argument to fact, as it is not used.
Let us use factGen as the first argument. Clearly,
`factGen factGen 10` computes the same thing as `fact 10`

Encoding Recursion: Using Extra Parameter

Starting from:

```
(def factGen = (self => n =>
                  if n then * n (factGen self (- n 1)) else 1)
 factGen factGen 10)
```

let us assume that `factGen` will always be called with itself as the first argument.

Encoding Recursion: Using Extra Parameter

Starting from:

```
(def factGen = (self => n =>
                  if n then * n (factGen self (- n 1)) else 1)
 factGen factGen 10)
```

let us assume that `factGen` will always be called with itself as the first argument. Then `factGen` and `self` are interchangeable, so let us use `self` in the body:

```
(def factGen = (self => n =>
                  if n then * n (self self (- n 1)) else 1)
 factGen factGen 10)
```

Now `factGen` is not recursive any more, it uses higher-order functions instead. Thus our interpreter does not need support for recursive definitions.

Non-Recursive Definitions Using Anonymous Functions

We can always substitute away definitions, instead of:

```
(def factGen = (self => n =>
                  if n then * n (self self (- n 1)) else 1)
 factGen factGen 10)
```

we can write directly:

```
(self => n => if n then * n (self self (- n 1)) else 1) // factGen
  (self => n => if n then * n (self self (- n 1)) else 1) // factGen
   10
```

Non-Recursive Definitions Using Anonymous Functions

We can always substitute away definitions, instead of:

```
(def factGen = (self => n =>
                  if n then * n (self self (- n 1)) else 1)
 factGen factGen 10)
```

we can write directly:

```
(self => n => if n then * n (self self (- n 1)) else 1) // factGen
  (self => n => if n then * n (self self (- n 1)) else 1) // factGen
   10
```

We can also express this by turning `factGen` into a parameter:

```
(factGen => factGen factGen 10)
  (self => n => if n then * n (self self (- n 1)) else 1)
```

that expression reduces to the previous one after one function application.

Demo and Scala Version

This works in environment based-interpreter. Not much slower.

Demo and Scala Version

This works in environment based-interpreter. Not much slower.

It works in substitution-based interpreter, which is nice to observe.

Demo and Scala Version

This works in environment based-interpreter. Not much slower.

It works in substitution-based interpreter, which is nice to observe.

It also works in Scala, we just need to define the recursive type for `self`:

```
case class T(f: T => BigInt => BigInt)

val factGen: T = T(
  (self:T) =>
    (n:BigInt) =>
      if n != 0 then n * self.f(self)(n - 1)
                else 1
)
def factOf10: BigInt = factGen.f(factGen)(10)  // factGen factGen 10
def fact(m: BigInt): BigInt = factGen.f(factGen)(m)
```

Towards a General Form

There is nothing special about the constant 10 in

```
(self => n => if n then * n (self self (- n 1)) else 1)
  (self => n => if n then * n (self self (- n 1)) else 1)  10
```

If we take arbitrary `m`, the expression

Towards a General Form

There is nothing special about the constant 10 in

```
(self => n => if n then * n (self self (- n 1)) else 1)
  (self => n => if n then * n (self self (- n 1)) else 1)  10
```

If we take arbitrary `m`, the expression

```
(self => n => if n then * n (self self (- n 1)) else 1)
  (self => n => if n then * n (self self (- n 1)) else 1)  m
```

computes the factorial of `m`. Thus,

```
(self => n => if n then * n (self self (- n 1)) else 1)
  (self => n => if n then * n (self self (- n 1)) else 1)
```

is the factorial function. Note that it is of the form

```
  (self => body1) (self => body1)
```

where `body1` is the body of the original factorial function but with `self self` instead of the recursive call.

Automating Recursive Function Encoding

```
def mkRecursive(recCallName: String, body: Expr): Expr =
  val body1 = subst(body, recCallName, Call(N("self"), N("self")))
  val selfToBody1 = Fun("self", body1)
  Call(selfToBody1, selfToBody1)
```

For example, if we define the term `factBody` as:

```
n => if n then * n (myself (- n 1)) else 1
```

then evaluating the term

```
Call(mkRecursive("myself", factBody), C(6))
```

gives 720, as desired. We could thus use desugaring to support recursive constructs instead of having a support in the interpreter.

Eliminated Recursion. What about representing numbers?

```
enum Expr
  case C(c: BigInt)     // to eliminate
  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) // OK
```

We now make language smaller, but without losing expressive power!
We wish to show that we only need these three constructs:

```
enum Expr
  case N(name: String)
  case Call(function: Expr, arg: Expr)
  case Fun(param: String, body: Expr)
```

The higher-order language with only these three constructs is called **lambda calculus**.

N-fold Function Application

We defined twice like this:

```
f => x => f (f x)
```

Maybe we can use it to represent number two?
What should we use to represent number three?

N-fold Function Application

We defined twice like this:

```
f => x => f (f x)
```

Maybe we can use it to represent number two?
What should we use to represent number three?

```
f => x => f (f (f x))
```

N-fold Function Application

We defined twice like this:

```
f => x => f (f x)
```

Maybe we can use it to represent number two?
What should we use to represent number three?

```
f => x => f (f (f x))
```

What about zero?

```
f => x => x
```

Such numbers, where *n* becomes *n*-fold function application, are called **Church numerals** according to Alonso Church, inventor of lambda calculus.
Is there a function that computes addition?

N-fold Function Application

We defined twice like this:

```
f => x => f (f x)
```

Maybe we can use it to represent number two?
What should we use to represent number three?

```
f => x => f (f (f x))
```

What about zero?

```
f => x => x
```

Such numbers, where *n* becomes *n*-fold function application, are called **Church numerals** according to Alonso Church, inventor of lambda calculus.
Is there a function that computes addition? A composition of iterations of `f`:

```
m => n => (f => x => m f (n f x))
```

Example of Evaluation of Two Plus Three

```
(m => n => f => x => m f (n f x))      // plus
   (f => x => f (f x))                 // two
      (f => x => f (f (f x)))          // three

~~>

f => x =>
   ((f => (x => (f (f x)))) f)  ((f => x => (f (f (f x)))) f x)
```

If we apply the above term to some concrete F and X we would get call-by-value evaluation corresponding to:

```
      ((f => (x => (f (f x)))) F) ((f => x => (f (f (f x)))) F X)
~~>   (x => (F (F x))) (F (F (F X)))
```

we would evaluate three times F applied to X, then two more times F applied to result.

Eliminated Recursion and Numebrs. What about 'if'?

```
enum Expr
  case C(c: BigInt)      // OK
  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) // OK
```

We wish to show that we only need these three constructs:

```
enum Expr
  case N(name: String)
  case Call(function: Expr, arg: Expr)
  case Fun(param: String, body: Expr)
```

The higher-order language with only these three constructs is called **lambda calculus**.

How To Check If Numeral is Nonzero?

Given a numeral *n*, like one for two:

```
f => x => f (f x)
```

How can we apply it to some expressions to get the effect of

```
if n then eTrue else eFalse
```

We give to numeral a specifically crafted function as `f` and a term as the initial value `x`.

How To Check If Numeral is Nonzero?

Given a numeral *n*, like one for two:

```
f => x => f (f x)
```

How can we apply it to some expressions to get the effect of

```
if n then eTrue else eFalse
```

We give to numeral a specifically crafted function as `f` and a term as the initial value `x`. When `n` is zero (that is, `f ⇒ x ⇒ x`) we want to return `eFalse`.

How To Check If Numeral is Nonzero?

Given a numeral *n*, like one for two:

```
f => x => f (f x)
```

How can we apply it to some expressions to get the effect of

```
if n then eTrue else eFalse
```

We give to numeral a specifically crafted function as `f` and a term as the initial value `x`.
When `n` is zero (that is, `f` ⇒ `x` ⇒ `x`) we want to return `eFalse`.
Let `f` be constant function that ignores its argument and returns `eTrue`.
Thus, we can try:

```
n (arg => eTrue) eFalse
```

How To Check If Numeral is Nonzero?

Given a numeral *n*, like one for two:

```
f => x => f (f x)
```

How can we apply it to some expressions to get the effect of

```
if n then eTrue else eFalse
```

We give to numeral a specifically crafted function as `f` and a term as the initial value `x`.
When `n` is zero (that is, `f ⇒ x ⇒ x`) we want to return `eFalse`.
Let `f` be constant function that ignores its argument and returns `eTrue`.
Thus, we can try:

```
n (arg => eTrue) eFalse
```

Unfortunately, this always evaluates the false branch. To prevent that:

```
(n (arg => foo => eTrue) (foo => eFalse)) bar
```

where `foo` is an arbitrary parameter and `bar` is an arbitrary term (e.g., `(x ⇒ x)`).

Creating Equivalent to IfNonzero

```
def mkIf(n: Expr, eTrue: Expr, eFalse: Expr): Expr =
  Call(
    Call(Call(n, Fun("arg", Fun("foo", eTrue))),
         Fun("foo", eFalse)),
    Fun("x", N("x")))
```

Lambda Calculus

```
enum Expr
  case N(name: String)
  case Call(function: Expr, arg: Expr)
  case Fun(param: String, body: Expr)
```

A general-purpose computation model that can express recursion, numbers, lists and other data types. Standard notation in lambda calculus:

syntax tree	our simple language	lambda calculus
N("x")	x	x
Call(f, e)	f e	f e
Fun(x, e)	x => e	$\lambda x.e$

We have seen it work with **call by value** evaluation. A more common evaluation used in lambda calculus theory (and in Haskell) is call-by-name, which results in more efficient encoding and more terms that are terminating.