Current Language (I07) vs Lambda Calculus

We have seen an interpreter for a language with nested recursive definitions:

```
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 Language (l07) vs Lambda Calculus

We have seen an interpreter for a language with nested recursive definitions:

```
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: Extra 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: Extra 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: Extra 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.

First-Class Functions Subsume Recursion

This encoding works in environment based-interpreter. (Not much slower.)

First-Class Functions Subsume Recursion

This encoding works in environment based-interpreter. (Not much slower.)

It also works in substitution-based interpreter, which is instructive to follow.

First-Class Functions Subsume Recursion

This encoding works in environment based-interpreter. (Not much slower.)

It also works in substitution-based interpreter, which is instructive to follow.

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.