Eliminated Recursion! What about representing numbers?

```
enum Expr
  case C(c: BigInt)     // <-- to eliminate next
  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) // Done
```

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 Alonzo 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 Alonzo 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

```
ifNonzero 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

```
ifNonzero 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

```
ifNonzero 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

```
ifNonzero 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, encode `IfNonzero` as:

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

where `_` is an arbitrary parameter and `d` is any lambda term, e.g., `x ⇒ x`

Illustrating encoding of IfNonzero

Take the proposed encoding of `IfNonzero(n,eTrue, eFalse)`:

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

Suppose n is zero, `f ⇒ x ⇒ x`. Then:

```
(f => x => x) (arg => _ => eTrue) (_ => eFalse) d
  ~~> (_ => eFalse) d
  ~~> eFalse
```

Suppose n is one, `f ⇒ x ⇒ f x`. Then:

```
(f => x => f x) (arg => _ => eTrue) (_ => eFalse) d
  ~~> (arg => _ => eTrue) (_ => eFalse) d
  ~~> eTrue
```

Suppose n is, e.g., two, `f ⇒ x ⇒ f (f x)`. Then:

```
(f => x => f (f x)) (arg => _ => eTrue) (_ => eFalse) d
  ~~> (arg => _ => eTrue) ((arg => _ => eTrue) (_ => eFalse)) d
  ~~> (arg => _ => eTrue) (_ => eTrue) d
  ~~> eTrue
```

Automating Encoding of 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")))
```

Reduced to lambda calculus

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

All that is left is:

```
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**.

Lambda Calculus Notation

```
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	common terminology
N("x")	x	x	variable
Call(f, e)	f e	f e	application
Fun(x, e)	x => e	$\lambda x.e$	abstraction

We have seen it work with **call by value** evaluation. Another common evaluation used in lambda calculus theory (and in Haskell) is call-by-name, which terminates on some of the programs for which call by value diverges.