The definition and application of functions.

The addition of features definable in the lambda calculus itself.
Some are easily defined: *e.g.*, syntax for numbers and collections.
Others are more challenging: *e.g.*, mutable reference cells.

Variable: `v`
Abstraction: `$\lambda$ x.t`
Application: `t t`

An external language contains derived forms and can be translated to an internal language which uses only core features.

To the left.

The following two are equivalent:

```
f g u
(f g) u
```

As far to the right as possible.

A variable is free if it is not bound by an enclosing abstraction.

Closed terms (aka *combinators*) have no free variables.

A *redex* term:

```
$(\lambda x.t_1) t_2$
```

Evaluation by *beta reduction*: a new term, namely `$t_1$` with each instance of `$x$` in it replaced by `$t2$`.

A rule for determining which redex in a term can be evaluated when.

... *calling convention* which is unrelated, even though some of the evaluation strategies have "call" in their name.

- Full beta-reduction
- Normal order
- Call-by-name
- Call-by-value

Call-by-value is the most common.

An evaluation strategy which allows any redex in a term to be evaluated in the next step.

An evaluation strategy in which only the outermost redex can be evaluated in the next step.

An evaluation strategy that follows the normal order strategy but does not allow reductions inside the body of an abstraction.

A term that cannot be further reduced.

An evaluation strategy that follows the normal order strategy but in which redexes are reduced by first reducing its argument to a value.

In strict evaluation strategies arguments to abstractions are evaluated regardless of whether they are used in the body of a function. In a non-strict (aka *lazy*) evaluation strategy an argument may not be evaluated if it's not used in the body.

Call-by-need, which is call-by-value with memoization. More specifically, after an argument is evaluated all other instances of that argument will be replaced with the value.

Using *currying*, a technique for representing multiple argument functions as higher-order unary functions.
In general:

```
$f = \lambda (x, y).s$$f = \lambda x.\lambda y.s$
```

Techniques for embedding data and operators into the lambda calculus. It is equivalent to *Godel numbering*. The latter is used for natural numbers, the former for lambda abstractions.

They are binary functions. `true` simply returns its first argument, `false` its second.

```
true = $\{lambda}t.\{lambda}f.t$false =
$\{lambda}t.\{lambda}f.f$
```

```
ifelse = $\{lambda}l.\{lambda}m.\{lambda}n .
l m n$
```

Now `ifelse b v w` will return `v` if `b` is `true`, and `w` if `b` is `false`.