The Lambda Calculus

Alonzo Church

Alan Turing
The lambda-calculus

If our previous language of arithmetic expressions was the simplest nontrivial programming language, then the lambda-calculus is the simplest interesting programming language...

- Turing complete
- higher order (functions as data)
- main new feature: variable binding and lexical scope

- The e. coli of programming language research

- The foundation of many real-world programming language designs (including ML, Haskell, Scheme, Lisp, ...)

CIS 500, 22 September
**Intuitions**

Suppose we want to describe a function that adds three to any number we pass it. We might write

\[
\text{plus3 } x \ = \ \text{succ } (\text{succ } (\text{succ } x))
\]

That is, “\text{plus3 } x \text{ is succ } (\text{succ } (\text{succ } x)).”
Intuitions

Suppose we want to describe a function that adds three to any number we pass it. We might write

\[ \text{plus3 } x \ = \ \text{succ} \ (\text{succ} \ (\text{succ } x)) \]

That is, “\text{plus3 } x \text{ is succ (succ (succ } x))\text{.”}"

Q: What is \text{plus3} itself?
Suppose we want to describe a function that adds three to any number we pass it. We might write

\[
\text{plus3 } x \ = \ \text{succ} \ (\text{succ} \ (\text{succ} \ x))
\]

That is, “plus3 \(x\) is succ (succ (succ \(x\))).”

Q: What is plus3 itself?

A: plus3 is the function that, given \(x\), yields succ (succ (succ \(x\))).
Suppose we want to describe a function that adds three to any number we pass it. We might write

\[
\text{plus3 } x = \text{succ (succ (succ } x))
\]

That is, "\text{plus3 } x\text{ is succ (succ (succ } x))\text{.}"

Q: What is \text{plus3} itself?

A: \text{plus3} is the function that, given \( x \), yields \text{succ (succ (succ } x))\.

\[
\text{plus3 } = \lambda x. \text{succ (succ (succ } x))
\]

This function exists independent of the name \text{plus3}.\)
Intuitions

Suppose we want to describe a function that adds three to any number we pass it. We might write

\[ \text{plus3} \ x \ = \ \text{succ} \ (\text{succ} \ (\text{succ} \ x)) \]

That is, “\text{plus3} \ x \ \text{is succ} \ (\text{succ} \ (\text{succ} \ x))”.

Q: What is \text{plus3} itself?

A: \text{plus3} is the function that, given \( x \), yields \( \text{succ} \ (\text{succ} \ (\text{succ} \ x)) \).

\[ \text{plus3} \ = \ \lambda x. \ \text{succ} \ (\text{succ} \ (\text{succ} \ x)) \]

This function exists independent of the name \text{plus3}.

On this view, \( \text{plus3} \ (\text{succ} \ 0) \) is just a convenient shorthand for “the function that, given \( x \), yields \( \text{succ} \ (\text{succ} \ (\text{succ} \ x)) \), applied to \( \text{succ} \ 0 \)”.

\[ \text{plus3} \ (\text{succ} \ 0) \ = \ (\lambda x. \ \text{succ} \ (\text{succ} \ (\text{succ} \ x))) \ (\text{succ} \ 0) \]
Essentials

We have introduced two primitive syntactic forms:

♦ abstraction of a term $t$ on some subterm $x$:

   \[ \lambda x. t \]

   “The function that, when applied to a value $v$, yields $t$ with $v$ in place of $x$.”

♦ application of a function to an argument:

   $t_1 ~ t_2$

   “the function $t_1$ applied to the argument $t_2$”

Recall that we wrote anonymous functions “\texttt{fun x \rightarrow t}” in OCaml.
Abstractions over Functions

Consider the \( \lambda \)-abstraction

\[
g = \lambda f. f (f \ (\text{succ} \ 0))
\]

Note that the parameter variable \( f \) is used in the function position in the body of \( g \). Terms like \( g \) are called higher-order functions.

If we apply \( g \) to an argument like \text{plus3}, the “substitution rule” yields a nontrivial computation:

\[
g \ \text{plus3} \ = \ (\lambda f. f (f \ (\text{succ} \ 0))) \ (\lambda x. \ \text{succ} \ (\text{succ} \ (\text{succ} \ x)))
\]

i.e. \( (\lambda x. \ \text{succ} \ (\text{succ} \ (\text{succ} \ x))) \)

\[
((\lambda x. \ \text{succ} \ (\text{succ} \ (\text{succ} \ x))) \ \text{succ} \ 0))
\]

i.e. \( (\lambda x. \ \text{succ} \ (\text{succ} \ (\text{succ} \ x))) \)

\[
(\text{succ} \ (\text{succ} \ (\text{succ} \ (\text{succ} \ 0))))
\]

i.e. \( \text{succ} \ (\text{succ} \ (\text{succ} \ (\text{succ} \ (\text{succ} \ (\text{succ} \ 0))))) \)
Abstractions Returning Functions

Consider the following variant of $g$:

$$\text{double} \ = \ \lambda f. \ \lambda y. \ f \ (f \ y)$$

I.e., $\text{double}$ is the function that, when applied to a function $f$, yields a function that, when applied to an argument $y$, yields $f \ (f \ y)$.

Prelude> let g = \f -> \y -> f (f y)
Prelude> g (+ 2) 3
7
Example

double plus3 0
=  (λf. λy. f (f y))
   (λx. succ (succ (succ x)))
   0
i.e.  (λy. (λx. succ (succ (succ x)))
      ((λx. succ (succ (succ x))) y))
     0
i.e.  (λx. succ (succ (succ x)))
     ((λx. succ (succ (succ x))) 0)
i.e.  (λx. succ (succ (succ x)))
     (succ (succ (succ 0)))
i.e.  succ (succ (succ (succ (succ 0)))))
The Pure Lambda-Calculus

As the preceding examples suggest, once we have \( \lambda \)-abstraction and application, we can throw away all the other language primitives and still have left a rich and powerful programming language.

In this language — the “pure lambda-calculus” — everything is a function.

- Variables always denote functions
- Functions always take other functions as parameters
- The result of a function is always a function
Formalities
Syntax

\[
  t ::= \quad \text{terms}
  \]
  \[
    x \quad \text{variable}
  \]
  \[
    \lambda x.t \quad \text{abstraction}
  \]
  \[
    t t \quad \text{application}
  \]

Terminology:

- terms in the pure \( \lambda \)-calculus are often called \( \lambda \)-terms
- terms of the form \( \lambda x. \ t \) are called \( \lambda \)-abstractions or just abstractions
**Scope**

The \( \lambda \)-abstraction term \( \lambda x. t \) binds the variable \( x \).

The *scope* of this binding is the body \( t \).

Occurrences of \( x \) inside \( t \) are said to be *bound* by the abstraction.

Occurrences of \( x \) that are *not* within the scope of an abstraction binding \( x \) are said to be *free*.

\[
\lambda x. \lambda y. x y z
\]
Scope

The $\lambda$-abstraction term $\lambda x. t$ binds the variable $x$.

The scope of this binding is the body $t$.

Occurrences of $x$ inside $t$ are said to be bound by the abstraction.

Occurrences of $x$ that are not within the scope of an abstraction binding $x$ are said to be free.

\[
\lambda x. \lambda y. x y z \\
\lambda x. (\lambda y. z y) y
\]
Values

\[ v ::= \]
\[ \lambda x.t \quad \text{abstraction value} \]

\[ t ::= \]
\[ x \quad \text{variable} \]
\[ \lambda x.t \quad \text{abstraction} \]
\[ t t \quad \text{application} \]

values
Operational Semantics

Computation rule:

\[(\lambda x.t_{12}) v_2 \longrightarrow [x \mapsto v_2] t_{12}\]  \hspace{1cm} (E-APPABS)

Notation: \([x \mapsto v_2] t_{12}\) is “the term that results from substituting free occurrences of \(x\) in \(t_{12}\) with \(v_{12}\).”
Operational Semantics

Computation rule:

$$(\lambda x. t_{12}) v_2 \rightarrow [x \mapsto v_2] t_{12} \quad \text{(E-APPABS)}$$

**Notation:** $[x \mapsto v_2] t_{12}$ is “the term that results from substituting free occurrences of $x$ in $t_{12}$ with $v_{12}$.”

**Congruence rules:**

call by name:

$$
\begin{array}{c}
t_1 \rightarrow t'_1 \\
t_1 t_2 \rightarrow t'_1 t_2
\end{array}
\quad
\begin{array}{c}
t_1 \rightarrow t'_1 \\
t_1 t_2 \rightarrow t'_1 t_2
\end{array} \quad \text{(E-APP1)}$$

big-step semantics

$$
\begin{array}{c}
\lambda x. t \downarrow \lambda x. t \\
t_1 \downarrow \lambda x. t_{12} \\
t_2 \downarrow v_2 \quad [x \mapsto v_2] t_{12} \downarrow t' \\
t_1, t_2 \downarrow t'
\end{array}
\quad
\begin{array}{c}
t_2 \downarrow t'_2 \\
v_1 t_2 \downarrow v_1 t'_2
\end{array} \quad \text{(E-APP2)}$$
A term of the form \((\lambda x. t) v\) — that is, a \(\lambda\)-abstraction applied to a value — is called a redex (short for “reducible expression”).
Alternative evaluation strategies

Strictly speaking, the language we have defined is called the pure, call-by-value lambda-calculus.

The evaluation strategy we have chosen — call by value — reflects standard conventions found in most mainstream languages.

Some other common ones:

- Call by name (cf. Haskell)
- Normal order (leftmost/outermost)
- Full (non-deterministic) beta-reduction
Programming in the Lambda-Calculus
Multiple arguments

Above, we wrote a function `double` that returns a function as an argument.

\[\text{double} = \lambda f. \lambda y. f (f y)\]

This idiom — a \(\lambda\)-abstraction that does nothing but immediately yield another abstraction — is very common in the \(\lambda\)-calculus.

In general, \(\lambda x. \lambda y. t\) is a function that, given a value \(v\) for \(x\), yields a function that, given a value \(u\) for \(y\), yields \(t\) with \(v\) in place of \(x\) and \(u\) in place of \(y\).

That is, \(\lambda x. \lambda y. t\) is a two-argument function.

(Recall the discussion of currying in OCaml.)
Syntactic conventions

Since λ-calculus provides only one-argument functions, all multi-argument functions must be written in curried style.

The following conventions make the linear forms of terms easier to read and write:

- Application associates to the left
  E.g., \( t \ u \ v \) means \( (t \ u) \ v \), not \( t \ (u \ v) \)

- Bodies of \( \lambda \) abstractions extend as far to the right as possible
  E.g., \( \lambda x. \ \lambda y. \ x \ y \) means \( \lambda x. \ (\lambda y. \ x \ y) \), not \( \lambda x. \ (\lambda y. \ x) \ y \)
The “Church Booleans”

\[ \text{tru} = \lambda t. \lambda f. t \]
\[ \text{fls} = \lambda t. \lambda f. f \]

\[ \text{tru} \ v \ w \]
\[ = (\lambda t. \lambda f. t) \ v \ w \quad \text{by definition} \]
\[ \rightarrow (\lambda f. \ v) \ w \quad \text{reducing the underlined redex} \]
\[ \rightarrow v \quad \text{reducing the underlined redex} \]

\[ \text{fls} \ v \ w \]
\[ = (\lambda t. \lambda f. f) \ v \ w \quad \text{by definition} \]
\[ \rightarrow (\lambda f. \ f) \ w \quad \text{reducing the underlined redex} \]
\[ \rightarrow w \quad \text{reducing the underlined redex} \]
Functions on Booleans

\[ \text{not} = \lambda b. \text{if } b \text{ then } \text{fls} \text{ else } \text{tru} \]

That is, \text{not} is a function that, given a boolean value \( v \), returns \text{fls} if \( v \) is \text{tru} and \text{tru} if \( v \) is \text{fls}.
Functions on Booleans

\[ \text{and} = \lambda b. \lambda c. b \ c \ \text{fls} \]

That is, \text{and} is a function that, given two boolean values \( v \) and \( w \), returns \( w \) if \( v \) is \text{tr}u and \text{fls} if \( v \) is \text{fl}s (short-circuit ?)

Thus \text{and} \( v \ w \) yields \text{tr}u if both \( v \) and \( w \) are \text{tr}u and \text{fls} if either \( v \) or \( w \) is \text{fl}s.

what about \text{or}?
Pairs

\begin{align*}
\text{pair} &= \lambda f. \lambda s. \lambda b. \, b \, f \, s \\
\text{fst} &= \lambda p. \, p \, \text{tru} \\
\text{snd} &= \lambda p. \, p \, \text{fls}
\end{align*}

That is, \text{pair} \, v \, w \text{ is a function that, when applied to a boolean value } b, \text{ applies } b \text{ to } v \text{ and } w.

By the definition of booleans, this application yields \( v \) if \( b \) is \( \text{tru} \) and \( w \) if \( b \) is \( \text{fls} \), so the first and second projection functions \( \text{fst} \) and \( \text{snd} \) can be implemented simply by supplying the appropriate boolean.

\begin{align*}
\text{fst} \, \text{pair} \, v \, w \\
= &\; \text{fst} \, (\lambda f. \lambda s. \lambda b. \, b \, f \, s) \, v \, w \\
\rightarrow &\; \text{fst} \, (\lambda s. \lambda b. \, b \, v \, s) \, w \\
\rightarrow &\; \text{fst} \, (\lambda b. \, b \, v \, w) \\
= &\; (\lambda p. \, p \, \text{tru}) \, (\lambda b. \, b \, v \, w) \\
\rightarrow &\; (\lambda b. \, b \, v \, w) \, \text{tru} \\
\rightarrow &\; \text{tru} \, v \, w \\
\rightarrow^* &\; v
\end{align*}

by definition

reducing the underlined redex

reducing the underlined redex

by definition

reducing the underlined redex

reducing the underlined redex

as before.
\[
\text{fst\ (pair\ } v \ w) \\
= \text{fst\ ((}\lambda f.\ \lambda s.\ \lambda b.\ b\ f\ s)\ v\ w) \quad \text{by definition} \\
\rightarrow \text{fst\ ((}\lambda s.\ \lambda b.\ b\ v\ s)\ w) \quad \text{reducing the underlined redex} \\
\rightarrow \text{fst\ (}\lambda b.\ b\ v\ w) \quad \text{reducing the underlined redex} \\
= \text{(}\lambda p.\ p\ \text{tru}\text{)}\ (}\lambda b.\ b\ v\ w) \quad \text{by definition} \\
\rightarrow \text{(}\lambda b.\ b\ v\ w)\ \text{tru} \quad \text{reducing the underlined redex} \\
\rightarrow \text{tru\ v\ w} \quad \text{reducing the underlined redex} \\
\rightarrow^* v \quad \text{as before.}
\]
Church numerals

Idea: represent the number \( n \) by a function that “repeats some action \( n \) times.”

\[
\begin{align*}
c_0 &= \lambda s. \lambda z. z & \text{what about “fls”? maybe C is right...} \\
c_1 &= \lambda s. \lambda z. s\ z \\
c_2 &= \lambda s. \lambda z. s\ (s\ z) \\
c_3 &= \lambda s. \lambda z. s\ (s\ (s\ z)) \\
\end{align*}
\]

That is, each number \( n \) is represented by a term \( c_n \) that takes two arguments, \( s \) and \( z \) (for “successor” and “zero”), and applies \( s \), \( n \) times, to \( z \).
Functions on Church Numerals

Successor:
Functions on Church Numerals

**Successor:**

\[ \text{sc}c = \lambda n. \lambda s. \lambda z. s (n \ s \ z) \]

another solution?

\[ \text{sc}c2 = \lambda n. \lambda s. \lambda z. n \ s \ (s \ z); \]

\[
\begin{align*}
c_0 &= \lambda s. \lambda z. z \\
c_1 &= \lambda s. \lambda z. s \ z \\
c_2 &= \lambda s. \lambda z. s \ (s \ z) \\
c_3 &= \lambda s. \lambda z. s \ (s \ (s \ z))
\end{align*}
\]
Functions on Church Numerals

Successor:

\[
\text{succ} = \lambda n. \lambda s. \lambda z. s (n s z)
\]
\[
\text{succ2} = \lambda n. \lambda s. \lambda z. n s (s z);
\]

Addition:

\[
c_0 = \lambda s. \lambda z. z
\]
\[
c_1 = \lambda s. \lambda z. s z
\]
\[
c_2 = \lambda s. \lambda z. s (s z)
\]
\[
c_3 = \lambda s. \lambda z. s (s (s z))
\]
Functions on Church Numerals

Successor:

\[
\text{successor} = \lambda n. \, \lambda s. \, \lambda z. \, s \, (n \, s \, z) \\
\text{successor} = \lambda n. \, \lambda s. \, \lambda z. \, n \, s \, (s \, z);
\]

Addition:

\[
\text{addition} = \lambda m. \, \lambda n. \, \lambda s. \, \lambda z. \, m \, s \, (n \, s \, z)
\]

\[
c_0 = \lambda s. \, \lambda z. \, z \\
c_1 = \lambda s. \, \lambda z. \, s \, z \\
c_2 = \lambda s. \, \lambda z. \, s \, (s \, z) \\
c_3 = \lambda s. \, \lambda z. \, s \, (s \, (s \, z))
\]
Functions on Church Numerals

Successor:

\[
\text{scc} = \lambda n. \lambda s. \lambda z. s (n \ s \ z) \\
\text{scc2} = \lambda n. \lambda s. \lambda z. n \ s \ (s \ z);
\]

Addition:

\[
\text{plus} = \lambda m. \lambda n. \lambda s. \lambda z. m \ s \ (n \ s \ z)
\]

Multiplication:

\[
c_0 = \lambda s. \lambda z. z \\
c_1 = \lambda s. \lambda z. s \ z \\
c_2 = \lambda s. \lambda z. s \ (s \ z) \\
c_3 = \lambda s. \lambda z. s \ (s \ (s \ z))
\]
Functions on Church Numerals

Successor:
\[
\text{scc } = \lambda n. \lambda s. \lambda z. s \ (n \ s \ z) \\
\text{scc2 } = \lambda n. \lambda s. \lambda z. n \ s \ (s \ z);
\]

Addition:
\[
\text{plus } = \lambda m. \lambda n. \lambda s. \lambda z. m \ s \ (n \ s \ z)
\]

Multiplication:
\[
\text{times } = \lambda m. \lambda n. m \ (\text{plus} \ n) \ c_0
\]
Functions on Church Numerals

**Successor:**
\[
\operatorname{scc} = \lambda n. \lambda s. \lambda z. s \ (n \ s \ z) \\
\operatorname{scc2} = \lambda n. \lambda s. \lambda z. n \ s \ (s \ z);
\]

**Addition:**
\[
\operatorname{plus} = \lambda m. \lambda n. \lambda s. \lambda z. m \ s \ (n \ s \ z)
\]

**Multiplication:**
\[
\operatorname{times} = \lambda m. \lambda n. m \ (\operatorname{plus} \ n) \ c_0
\]

**Zero test:**
\[
c_0 = \lambda s. \lambda z. z \\
c_1 = \lambda s. \lambda z. s \ z \\
c_2 = \lambda s. \lambda z. s \ (s \ z) \\
c_3 = \lambda s. \lambda z. s \ (s \ (s \ z))
\]
Functions on Church Numerals

Successor:

\[
\text{succ} = \lambda n. \lambda s. \lambda z. s \ (n \ s \ z)
\]
\[
\text{succ2} = \lambda n. \lambda s. \lambda z. n \ s \ (s \ z);
\]

Addition:

\[
\text{plus} = \lambda m. \lambda n. \lambda s. \lambda z. m \ s \ (n \ s \ z)
\]

Multiplication:

\[
\text{times} = \lambda m. \lambda n. m \ \text{plus} \ n \ c_0
\]

Zero test:

\[
\text{iszzero} = \lambda m. m \ (\lambda x. \text{fls}) \ \text{tru}
\]
Functions on Church Numerals

Successor:

\[ scc = \lambda n. \lambda s. \lambda z. s (n s z) \]
\[ scc2 = \lambda n. \lambda s. \lambda z. n s (s z); \]

Addition:

\[ plus = \lambda m. \lambda n. \lambda s. \lambda z. m s (n s z) \]

Multiplication:

\[ times = \lambda m. \lambda n. m (plus n) c_0 \]

Zero test:

\[ iszro = \lambda m. m (\lambda x. fls) tru \]

\[ c_0 = \lambda s. \lambda z. z \]
\[ c_1 = \lambda s. \lambda z. s z \]
\[ c_2 = \lambda s. \lambda z. s (s z) \]
\[ c_3 = \lambda s. \lambda z. s (s (s z)) \]

\[ times2 = \lambda m. \lambda n. \lambda s. \lambda z. m (n s) z; \]

Or, more compactly:

\[ times3 = \lambda m. \lambda n. \lambda s. m (n s); \]

\[ power1 = \lambda m. \lambda n. m (times n) c_1; \]
\[ power2 = \lambda m. \lambda n. m n; \]

What about predecessor?
Predecessor

\[
\begin{align*}
zz &= \text{pair } c_0 \ c_0 \\
ss &= \lambda p. \text{pair } (\text{snd } p) \ (\text{scc } (\text{snd } p))
\end{align*}
\]
Predecessor

\[ \text{zz} = \text{pair } c_0 \ c_0 \]
\[ \text{ss} = \lambda p. \text{pair } (\text{snd } p) \ (\text{scc } (\text{snd } p)) \]
\[ \text{prd} = \lambda m. \text{fst } (m \ \text{ss } \text{zz}) \]

Questions:
1. what’s the complexity of \text{prd}?
2. how to define equal?
3. how to define subtract?
Normal forms

Recall:

♦ A normal form is a term that cannot take an evaluation step.

♦ A stuck term is a normal form that is not a value.

Are there any stuck terms in the pure $\lambda$-calculus?

Prove it.
Normal forms

Recall:

- A **normal form** is a term that cannot take an evaluation step.
- A **stuck** term is a normal form that is not a value.

Are there any stuck terms in the pure $\lambda$-calculus?

Prove it.

Does every term evaluate to a normal form?

Prove it.
Divergence

\[
\text{omega} \quad = \quad (\lambda x. \ x \ x) \ (\lambda x. \ x \ x)
\]

Note that \text{omega} evaluates in one step to itself!

So evaluation of \text{omega} never reaches a normal form: it \text{diverges}.
Divergence

\[ \text{omega} = (\lambda x. x \ x) \ (\lambda x. x \ x) \]

Note that \text{omega} evaluates in one step to itself!

So evaluation of \text{omega} never reaches a normal form: it diverges.

Being able to write a divergent computation does not seem very useful in itself. However, there are variants of \text{omega} that are very useful...
Iterated Application

Suppose $f$ is some $\lambda$-abstraction, and consider the following term:

$$Y_f = (\lambda x. f (x x)) (\lambda x. f (x x))$$
Iterated Application

Suppose $f$ is some $\lambda$-abstraction, and consider the following term:

$$Y_f = (\lambda x. f (x x)) (\lambda x. f (x x))$$

Now the “pattern of divergence” becomes more interesting:

$$Y_f$$

$$= (\lambda x. f (x x)) (\lambda x. f (x x))$$

$$\rightarrow f ((\lambda x. f (x x)) (\lambda x. f (x x)))$$

$$\rightarrow f (f ((\lambda x. f (x x)) (\lambda x. f (x x))))$$

$$\rightarrow f (f (f ((\lambda x. f (x x)) (\lambda x. f (x x)))))$$

$$\rightarrow \ldots$$
$Y_f$ is still not very useful, since (like $\omega$), all it does is diverge.

Is there any way we could “slow it down”? 

Delaying Divergence

poisonpill = \y. \omega

Note that poisonpill is a value — it will only diverge when we actually apply it to an argument. This means that we can safely pass it as an argument to other functions, return it as a result from functions, etc.

\((\lambda p. \text{fst} (\text{pair} p \text{ fls}) \text{ tru}) \text{ poisonpill}\)

\rightarrow

\text{fst} (\text{pair} \text{ poisonpill} \text{ fls}) \text{ tru}

\rightarrow^*

\text{poisonpill tru}

\rightarrow

\omega

\rightarrow

\ldots

Cf. thunks in OCaml.
A delayed variant of \( \omega \)

Here is a variant of \( \omega \) in which the delay and divergence are a bit more tightly intertwined:

\[
\omega_v = \lambda y. (\lambda x. (\lambda y. x x y)) (\lambda x. (\lambda y. x x y)) y
\]

Note that \( \omega_v \) is a normal form. However, if we apply it to any argument \( v \), it diverges:

\[
\omega_v \ v = \\
(\lambda y. (\lambda x. (\lambda y. x x y)) (\lambda x. (\lambda y. x x y)) y) \ v \\
\rightarrow \\
(\lambda x. (\lambda y. x x y)) (\lambda x. (\lambda y. x x y)) \ v \\
\rightarrow \\
(\lambda y. (\lambda x. (\lambda y. x x y)) (\lambda x. (\lambda y. x x y)) y) \ v \\
= \\
\omega_v \ v
\]
Another delayed variant

Suppose $f$ is a function. Define

$$Z_f = \lambda y. (\lambda x. f (\lambda y. x x y)) (\lambda x. f (\lambda y. x x y)) y$$

This term combines the “added $f$” from $Y_f$ with the “delayed divergence” of $\omega_g$. 
If we now apply $Z_f$ to an argument $v$, something interesting happens:

$Z_f \ v$

$= (\lambda y. (\lambda x. f (\lambda y. x \ x \ y)) \ (\lambda x. f (\lambda y. x \ x \ y)) \ y) \ v$

$\rightarrow (\lambda x. f (\lambda y. x \ x \ y)) \ (\lambda x. f (\lambda y. x \ x \ y)) \ v$

$\rightarrow f (\lambda y. (\lambda x. f (\lambda y. x \ x \ y)) \ (\lambda x. f (\lambda y. x \ x \ y)) \ y) \ v$

$= f \ Z_f \ v$

Since $Z_f$ and $v$ are both values, the next computation step will be the reduction of $f \ Z_f$ — that is, before we “diverge,” $f$ gets to do some computation.

Now we are getting somewhere.
Recursion

Let

\[ f = \lambda fct. \]
\[ \quad \lambda n. \]
\[ \quad \text{if } n=0 \text{ then 1} \]
\[ \quad \text{else } n \times (fct (\text{pred } n)) \]

\( f \) looks just the ordinary factorial function, except that, in place of a recursive call in the last time, it calls the function \( fct \), which is passed as a parameter.

N.b.: for brevity, this example uses “real” numbers and booleans, infix syntax, etc. It can easily be translated into the pure lambda-calculus (using Church numerals, etc.).
We can use $Z$ to “tie the knot” in the definition of $f$ and obtain a real recursive factorial function:

$$ f = \lambda fct. \quad \lambda n. 
\begin{cases} 
1 & \text{if } n=0 \\
 n \times (fct (\text{pred } n)) & \text{else}
\end{cases} $$

$$ Z_f = \lambda y. (\lambda x. f (\lambda y. x \times y)) (\lambda x. f (\lambda y. x \times y)) y $$
If we define

\[ Z = \lambda f. Z_f \]

i.e.,

\[ Z = \lambda f. \lambda y. (\lambda x. f (\lambda y. x x y)) (\lambda x. f (\lambda y. x x y)) y \]

then we can obtain the behavior of \( Z_f \) for any \( f \) we like, simply by applying \( Z \) to \( f \).

\[ Z f \rightarrow Z_f \]
For example:

```
fact = Z ( \lambda fct. \\
        \lambda n. \\
          if n=0 then 1 \\
          else n * (fct (pred n)) )
```
The term $Z$ here is essentially the same as the $\text{fix}$ discussed the book.

\[
Z = \lambda f. \lambda y. (\lambda x. f (\lambda y. x x y)) (\lambda x. f (\lambda y. x x y))
\]

\[
\text{fix} = \lambda f. (\lambda x. f (\lambda y. x x y)) (\lambda x. f (\lambda y. x x y))
\]

$Z$ is hopefully slightly easier to understand, since it has the property that $Z \ f \ v \rightarrow^* f (Z \ f) \ v$, which $\text{fix}$ does not (quite) share.

$\text{fix}$ is the (call-by-value) Y-combinator

![Y Combinator](image-url)