Lambda Calculus

An Accessible Introduction

Agenda

Foundations

Boolean Algebra

Natural Numbers

Omega: Self-Application

Y Combinator and Recursion

Typed Lambda Calculus

Historical Origins

- Alonzo Church, 1930s: a formal system where *functions* are the fundamental objects
- Goal: provide a foundation for logic and mathematics
- The logical system proved inconsistent — but the *computational core* survived and flourished
- Alan Turing (1937): lambda calculus and Turing machines are computationally equivalent

Why Study Lambda Calculus?

Theory

- Universal model of computation
- Foundation of type theory
- Basis of formal proofs

Practice

- Lisp, ML, Haskell
- Proof assistants: Coq, Agda, Lean
- Anonymous functions in *every* modern language

Key insight

Three constructs are enough to compute *anything* computable.

The Grammar

Lambda Terms — Λ

$$M, N \;::=\; x \;\mid\; \lambda x.\, M \;\mid\; M\, N$$

- x **variable** — a placeholder for a value
- $\lambda x.\, M$ **abstraction** — a function with parameter x, body M
- $M\, N$ **application** — apply M to argument N

That is *all*. No numbers, booleans, or loops — yet.

Precedence Conventions

- **Application is left-associative:**

$$M\,N\,P \;\equiv\; (M\,N)\,P$$

- **Abstraction extends as far right as possible:**

$$\lambda\,x.\,M\,N \;\equiv\; \lambda\,x.\,(M\,N)$$

- **Multiple parameters abbreviate currying:**

$$\lambda\,x\,y.\,M \;\equiv\; \lambda\,x.\,(\lambda\,y.\,M)$$

Some Lambda Terms

Examples

- $\lambda\, x.\, x$ — the **identity** function $\mathbf{I}$
- $\lambda\, x\, y.\, x$ — **constant** (K combinator): returns first arg
- $\lambda\, f\, x.\, f\, x$ — explicit function application
- $(\lambda\, x.\, x\, x)\, (\lambda\, x.\, x\, x)$ — self-application (preview of $\mathbf{\Omega}$)

Free and Bound Variables

In $\lambda\, x.\, M$, the variable x is bound in M — like the dummy variable in $\int f(x)\, dx$.

Free variables $\mathsf{FV}(M)$

$$\begin{aligned} \mathsf{FV}(x) &= \{x\} \\ \mathsf{FV}(\lambda\, x.\, M) &= \mathsf{FV}(M) \setminus \{x\} \\ \mathsf{FV}(M\, N) &= \mathsf{FV}(M) \cup \mathsf{FV}(N) \end{aligned}$$

Example

In $\lambda\, x.\, (x\, y)$: x is bound; y is **free**.

In $(\lambda\, x.\, x)\, x$: the first x is bound; the last is **free**.

Substitution

$M[N/x]$ — replace all *free* occurrences of x in M by N.

Capture-avoiding substitution (key cases)

$$
\begin{aligned}
x[N/x] &= N \\
y[N/x] &= y \quad (y \neq x) \\
(\lambda\, y.\, M)[N/x] &= \lambda\, y.\, M[N/x] \quad \text{if } y \neq x,\ y \notin \mathsf{FV}(N) \\
(\lambda\, x.\, M)[N/x] &= \lambda\, x.\, M
\end{aligned}
$$

If $y \in \mathsf{FV}(N)$, first rename y to a fresh variable. This avoids *variable capture*.

Alpha-Equivalence

The name of a bound variable is irrelevant:

$$\lambda x.\, x \quad =_\alpha \quad \lambda y.\, y \quad =_\alpha \quad \lambda z.\, z$$

These are all the *same* function, written differently.

Alpha-equivalence

$M =_\alpha N$ if N can be obtained from M by consistently renaming bound variables. We treat α-equivalent terms as identical.

Beta-Reduction: The Rule

The only computational rule:

β-reduction

$$(\lambda\, x.\ M)\, N \quad \to_\beta \quad M[N/x]$$

A *redex* is any subterm of the form $(\lambda\, x.\, M)\, N$.

$\twoheadrightarrow_\beta$ denotes zero or more β-reduction steps.

Example

$$(\lambda\, x.\, x\, x)\, (\lambda\, y.\, y) \quad \to_\beta \quad (\lambda\, y.\, y)\, (\lambda\, y.\, y) \quad \to_\beta \quad \lambda\, y.\, y$$

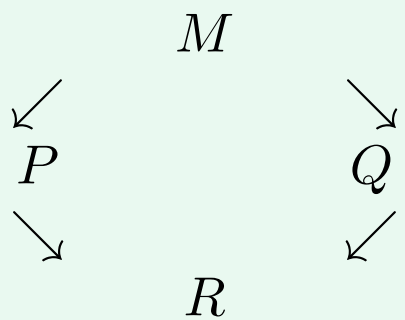
Normal Forms

Definitions

- A term is in **normal form** if it contains no redex
- A term **diverges** if it reduces forever (no normal form)

Church–Rosser Theorem (Confluence)

If $M \twoheadrightarrow_\beta P$ and $M \twoheadrightarrow_\beta Q$, there exists R such that $P \twoheadrightarrow_\beta R$ and $Q \twoheadrightarrow_\beta R$.



Consequence: normal forms are unique (up to α-equivalence).

Evaluation Strategies

Normal-order (lazy)

Always reduce the **leftmost outermost** redex first.
Normalising: finds a normal form whenever one exists.

Applicative-order (strict / call-by-value)

Always reduce the **leftmost innermost** redex first (evaluate arguments before applying).
May diverge even when a normal form exists.

Key example

$(\lambda\, x.\, \lambda\, y.\, y)\, \mathbf{\Omega}$
Normal-order: $\rightarrow_\beta \;\; \lambda\, y.\, y$ (terminates)
Applicative-order: reduces $\mathbf{\Omega}$ first (diverges)

Eta-Reduction

Two functions are *extensionally equal* if they agree on all inputs.

η-reduction

If $x \notin \mathsf{FV}(M)$, then:

$$\lambda\, x.\, (M\; x) \;\to_\eta\; M$$

- $\lambda\, x.\, (M\; x)$ and M behave identically on every input
- $\beta\eta$-equivalence is the standard notion of function equality
- Often omitted in introductions; β-reduction alone is sufficient

Part 2

Boolean Algebra

Church Booleans

Key insight: data can be encoded as *functions*.
A boolean is a *selector*: it chooses one of two options.

Church booleans

$$\text{true} \quad :\equiv \quad \lambda\, t\, f.\ t \qquad \text{(selects first argument)}$$

$$\text{false} \quad :\equiv \quad \lambda\, t\, f.\ f \qquad \text{(selects second argument)}$$

Both are closed terms (no free variables) — they are *values*.

The Conditional

If–then–else

$$\mathsf{if} \;\; :\equiv \;\; \lambda\, b\, t\, f.\; b\, t\, f$$

Evaluation:

- if true $M\ N \;\twoheadrightarrow_\beta\; (\lambda\, t\, f.\, t)\ M\ N \;\twoheadrightarrow_\beta\; M$
- if false $M\ N \;\twoheadrightarrow_\beta\; (\lambda\, t\, f.\, f)\ M\ N \;\twoheadrightarrow_\beta\; N$

The conditional is just function application — b itself does the selecting.

Boolean Operations

Definitions

$$\begin{aligned} \text{not} &:\equiv \lambda\, b.\ b\ \text{false}\ \text{true} \\ \text{and} &:\equiv \lambda\, b_1\, b_2.\ b_1\ b_2\ \text{false} \\ \text{or} &:\equiv \lambda\, b_1\, b_2.\ b_1\ \text{true}\ b_2 \end{aligned}$$

Intuition for and:
If $b_1 =$ true, return b_2 (conjunction is whatever b_2 is).
If $b_1 =$ false, return false immediately.

Example: not true $\twoheadrightarrow_\beta$ false

Reduction trace

$$\begin{aligned}
\text{not true} &= (\lambda\, b.\ b\ \text{false true})\ \text{true} \\
&\to_\beta\ \text{true false true} \\
&= (\lambda\, t\ f.\, t)\ \text{false true} \\
&\twoheadrightarrow_\beta\ \text{false} \qquad \checkmark
\end{aligned}$$

Similarly: and true false $\twoheadrightarrow_\beta$ false, or false true $\twoheadrightarrow_\beta$ true.
Full step-by-step derivations in the companion notes.

Part 3

Natural Numbers

Church Numerals: The Idea

Key insight: represent the number n as n-fold iteration of a function.

$$\overline{n}\ f\ x = \underbrace{f(f(\cdots(f}_{n \text{ times}}\ x)\cdots))$$

- $\overline{n}$ is a higher-order function
- It takes f and x, and applies f exactly n times to x
- Structure mirrors the Peano axioms

Church Numerals: Definitions

The first few numerals

$$\overline{0} = \lambda\, f\, x.\ x \qquad \text{(apply } f \text{ zero times)}$$

$$\overline{1} = \lambda\, f\, x.\ f\, x \qquad \text{(apply } f \text{ once)}$$

$$\overline{2} = \lambda\, f\, x.\ f\, (f\, x) \qquad \text{(apply } f \text{ twice)}$$

$$\overline{3} = \lambda\, f\, x.\ f\, (f\, (f\, x)) \qquad \text{(apply } f \text{ three times)}$$

Successor

Definition

$$\mathsf{succ} \quad :\equiv \quad \lambda\, n\, f\, x.\ f\,(n\, f\, x)$$

Verification: succ $\overline{2} = \overline{3}$

$$\begin{aligned} \mathsf{succ}\ \overline{2} &= (\lambda\, n\, f\, x.\ f\,(n\, f\, x))\ \overline{2} \\ &\twoheadrightarrow_\beta \ \lambda\, f\, x.\ f\,(\overline{2}\, f\, x) \\ &\twoheadrightarrow_\beta \ \lambda\, f\, x.\ f\,(f\,(f\, x)) = \overline{3} \quad \checkmark \end{aligned}$$

succ $\overline{n}$ applies f once more than $\overline{n}$ would.

Addition

Definition

$$\text{plus} \quad :\equiv \quad \lambda\, m\, n\, f\, x.\ m\, f\, (n\, f\, x)$$

Intuition: apply f first n times (via $n\, f\, x$), then m more times (via $m\, f$). Total: $m + n$ applications.

plus $\overline{2}\ \overline{2} \twoheadrightarrow_\beta \overline{4}$

$$\begin{aligned}
\text{plus}\ \overline{2}\ \overline{2} &\twoheadrightarrow_\beta \lambda\, f\, x.\ \overline{2}\, f\, (\overline{2}\, f\, x) \\
&\twoheadrightarrow_\beta \lambda\, f\, x.\ \overline{2}\, f\, (f\, (f\, x)) \\
&\twoheadrightarrow_\beta \lambda\, f\, x.\ f\, (f\, (f\, (f\, x))) = \overline{4} \quad \checkmark
\end{aligned}$$

Multiplication and Exponentiation

Multiplication

$$\text{times} \quad :\equiv \quad \lambda\, m\, n\, f.\ m\,(n\, f)$$

Apply f n times, m times over: total $m \times n$ applications.

Exponentiation

$$\text{exp} \quad :\equiv \quad \lambda\, m\, n.\ n\, m$$

The Church numeral n represents "apply n times".
Applying it to m means applying m exactly n times: m^n.

Exponentiation is just reverse application. Remarkably compact!

Testing for Zero

IsZero

$$\text{isZero} \quad :\equiv \quad \lambda\, n.\ n\,(\lambda\, x.\,\text{false})\,\text{true}$$

How it works:

- If $n = \overline{0}$: apply $(\lambda\, x.\,\text{false})$ *zero* times to true $\Rightarrow$ returns true
- If $n \geq \overline{1}$: apply $(\lambda\, x.\,\text{false})$ *at least once* $\Rightarrow$ always returns false, discarding the initial true

Predecessor: The Challenge

- succ is easy: add one more application of f
- pred is hard: cannot remove an application already built in
- Naive approaches fail — there is no direct "one less" operation

Solution (Church and Kleene, 1935)

Use **pairs** to carry an extra counter alongside the main value, then extract what we need at the end.

Encoding Pairs

Pairs as selector functions

$$\begin{aligned} \mathsf{pair} &:\equiv \lambda\, a\, b\, f.\ f\, a\, b \\ \mathsf{fst} &:\equiv \lambda\, p.\ p\, \mathsf{true} \\ \mathsf{snd} &:\equiv \lambda\, p.\ p\, \mathsf{false} \end{aligned}$$

A pair $\mathsf{pair}\ a\ b$ stores a and b and waits for a selector:

$$\mathsf{fst}\ (\mathsf{pair}\ a\ b)\ \twoheadrightarrow_\beta\ a, \qquad \mathsf{snd}\ (\mathsf{pair}\ a\ b)\ \twoheadrightarrow_\beta\ b$$

The Step Function

Idea: iterate a pair $(\text{prev}, \text{curr})$, starting from $(\overline{0}, \overline{0})$.

Step function

$$step \quad :\equiv \quad \lambda\, p.\ \text{pair}\ (\text{snd}\ p)\ (\text{succ}\ (\text{snd}\ p))$$

Iteration trace

$$(\overline{0}, \overline{0}) \xrightarrow{step} (\overline{0}, \overline{1}) \xrightarrow{step} (\overline{1}, \overline{2}) \xrightarrow{step} (\overline{2}, \overline{3}) \to \cdots$$

After n steps, the *first* component is $\overline{\max(0, n-1)}$.

Predecessor and Subtraction

Predecessor

$$\text{pred} \quad :\equiv \quad \lambda\, n.\ \text{fst}\ (n\ \mathit{step}\ (\text{pair}\ \overline{0}\ \overline{0}))$$

Apply step exactly n times, starting from $(\overline{0}, \overline{0})$, then take the first component.

Subtraction (monus)

$$\text{sub} \quad :\equiv \quad \lambda\, m\, n.\ n\ \text{pred}\ m$$

Apply pred exactly n times to m.
If $m < n$: result is $\overline{0}$ (natural numbers cannot go negative).

Part 4

Omega: Self-Application

The Self-Applying Term ω

Nothing in the grammar prevents a variable from being applied to itself.

Definition

$$\omega \quad :\equiv \quad \lambda x.\ x\, x$$

- ω takes an argument and applies it to itself
- Self-application is legal syntax in the *untyped* calculus
- As we'll see, it leads directly to non-termination *and* recursion

The Omega Term

Definition

$$\boldsymbol{\Omega} \quad :\equiv \quad \boldsymbol{\omega}\,\boldsymbol{\omega} = (\lambda\, x.\, x\, x)\,(\lambda\, x.\, x\, x)$$

Reduction

$$\boldsymbol{\Omega} = (\lambda\, x.\, x\, x)\,(\lambda\, x.\, x\, x) \;\rightarrow_{\beta}\; (\lambda\, x.\, x\, x)\,(\lambda\, x.\, x\, x) = \boldsymbol{\Omega}$$

Ω diverges

Each β-step produces $\boldsymbol{\Omega}$ again — an *infinite loop*.
$\boldsymbol{\Omega}$ is the archetypal divergent term.

Evaluation Order Matters

The term $(\lambda x.\, \lambda y.\, y)\, \mathbf{\Omega}$

- **Normal-order** (reduce outermost first):

$$(\lambda x.\, \lambda y.\, y)\, \mathbf{\Omega} \quad \to_\beta \quad \lambda y.\, y \quad \text{terminates!}$$

  ($\mathbf{\Omega}$ is the argument but is never evaluated)

- **Applicative-order** (evaluate argument first):

$$\mathbf{\Omega} \;\to_\beta\; \mathbf{\Omega} \;\to_\beta\; \mathbf{\Omega} \;\to_\beta\; \cdots \quad \text{diverges}$$

Three Lessons from Ω

1. **Non-termination is native.** The lambda calculus can express infinite loops — it is Turing-complete in the full sense.

2. **Evaluation order is observable.** Different strategies can give different termination behaviour. Language design must commit to a strategy.

3. **Self-application enables fixed points.** ω is the key ingredient in building combinators that can call themselves — the basis of the Y combinator.

Part 5

Y Combinator and Recursion

The Fixed-Point Theorem

Theorem (Church, 1941)

For every term $F \in \Lambda$ there exists a term M such that

$$F\,M \quad \twoheadrightarrow_\beta \quad M.$$

M is called a **fixed point** of F.

- In analysis: $f(x^*) = x^*$ (e.g. $f(x) = \cos x$)
- In the lambda calculus: *every function* has a fixed point
- The proof is constructive — it exhibits a fixed-point combinator

The Y Combinator

Definition

$$\mathbf{Y} \quad :\equiv \quad \lambda f.\ (\lambda x.\ f\,(x\,x))\,(\lambda x.\ f\,(x\,x))$$

- Introduced by Church (1941)
- $\mathbf{Y}$ is a closed term — no free variables
- The inner $\lambda x.\ f\,(x\,x)$ uses self-application (compare $\boldsymbol{\omega} = \lambda x.\ x\,x$)
- $\mathbf{Y}$ has no normal form under any evaluation strategy

Verifying the Fixed-Point Property

Claim: $\mathbf{Y}\,F \;\twoheadrightarrow_\beta\; F\,(\mathbf{Y}\,F)$.

Proof by reduction (let $H \;:\equiv\; \lambda\,x.\,F\,(x\,x)$)

$$
\begin{aligned}
\mathbf{Y}\,F &= (\lambda\,f.\;(\lambda\,x.\,f\,(x\,x))\,(\lambda\,x.\,f\,(x\,x)))\;F \\
&\to_\beta\;(\lambda\,x.\,F\,(x\,x))\,(\lambda\,x.\,F\,(x\,x)) \\
&= H\,H \\
&\to_\beta\;F\,((\lambda\,x.\,F\,(x\,x))\,(\lambda\,x.\,F\,(x\,x))) \\
&= F\,(H\,H) \\
&= F\,(\mathbf{Y}\,F) \qquad \checkmark
\end{aligned}
$$

Recursion via Fixed Points

Goal: define a function g satisfying $g\,n = \ldots g \ldots$
Lambda terms cannot refer to themselves by name.

The fixed-point trick

1. Define a non-recursive helper F such that $g = F\,g$
2. Then $g \;:\equiv\; \mathbf{Y}\,F$
3. By the fixed-point property: $\mathbf{Y}\,F \;\twoheadrightarrow_\beta\; F\,(\mathbf{Y}\,F) = F\,g$

F receives the function to call recursively as its *first argument*.

Factorial via Y

Define the one-step unroller F:

$$F \;\equiv\; \lambda\, f\, n.\ \text{if } (\text{isZero } n)\ \overline{1}\ (\text{times } n\ (f\ (\text{pred } n)))$$

Here f is the placeholder for the recursive call.

Recursive factorial

$$\mathit{fact} \;\equiv\; \mathbf{Y}\, F$$

By the fixed-point property:

$$\mathit{fact} \;=\; \mathbf{Y}\, F \;\twoheadrightarrow_\beta\; F\,(\mathbf{Y}\, F) \;=\; F\,\mathit{fact}$$

So fact satisfies its defining equation.

The Z Combinator (Call-by-Value)

Problem with Y under strict evaluation

In *applicative-order* semantics, $\mathbf{Y}\,F$ diverges immediately before any useful computation occurs.

Z combinator (call-by-value fixed point)

$$\mathbf{Z} \;:\equiv\; \lambda\, f.\; (\lambda\, x.\, f\,(\lambda\, v.\, x\, x\, v))\,(\lambda\, x.\, f\,(\lambda\, v.\, x\, x\, v))$$

The extra $\lambda\, v.\cdots$ delays self-application until the function is actually called.

Used in languages with strict evaluation (OCaml, Scheme without laziness).

Church–Turing Thesis

Key facts

- $\mathbf{\Omega}$ proves the lambda calculus can loop forever
- $\mathbf{Y}$ proves every function has a fixed point $\Rightarrow$ general recursion is expressible
- Church (1936) used this to show the *Entscheidungsproblem* is unsolvable

Church–Turing Thesis

Lambda calculus and Turing machines are equivalent models of computation. Both capture exactly the class of computable functions.

Part 6

Typed Lambda Calculus

Why Types?

The untyped calculus allows:

- Applying a "number" to another "number"
- Self-application ($\boldsymbol{\omega} = \lambda\, x.\, x\, x$) and divergence
- No static guarantee that a program is meaningful

Types impose discipline

- Ill-formed programs are rejected at type-checking time
- Well-typed programs enjoy strong safety guarantees
- Cost: some valid programs become inexpressible

Simple Types

Grammar of simple types

$$A, B \ ::= \ \alpha \ \mid \ A \to B$$

- $\alpha, \beta, \ldots$ are **base types** (atoms)
- $A \to B$ is the type of **functions** from A to B
- $\to$ is right-associative: $A \to B \to C = A \to (B \to C)$

Examples

- $\alpha \to \alpha$ — type of the identity function
- $\alpha \to \beta \to \alpha$ — type of the K combinator
- $(\alpha \to \alpha) \to \alpha \to \alpha$ — type of a Church numeral

Typing Contexts and Judgments

Typing context Γ

A finite set of **assumptions**: $x_1 : A_1,\ x_2 : A_2,\ \ldots$
Each variable appears at most once.

Typing judgment

$$\Gamma \vdash M : A$$

Read: "under assumptions Γ, term M has type A".

A judgment is *derivable* if it can be proved using the three typing rules.

Typing Rules

Simply Typed Lambda Calculus

$$\frac{(x : A) \in \Gamma}{\Gamma \vdash x : A}\ (\textsc{Var}) \qquad \frac{\Gamma,\, x{:}A \vdash M : B}{\Gamma \vdash \lambda\, x.\, M : A \to B}\ (\textsc{Abs})$$

$$\frac{\Gamma \vdash M : A \to B \quad \Gamma \vdash N : A}{\Gamma \vdash M\, N : B}\ (\textsc{App})$$

One rule per term constructor: variable, abstraction, application.

Examples: Identity and K

Identity $\lambda\, x.\, x : A \to A$

$$\frac{(x{:}A) \in \{x{:}A\}}{x{:}A \vdash x : A}\ (\textsc{Var}) \implies \frac{x{:}A \vdash x : A}{\varnothing \vdash \lambda\, x.\, x : A \to A}\ (\textsc{Abs})$$

K combinator $\lambda\, x\, y.\, x : A \to B \to A$

$$\varnothing \vdash \lambda\, x\, y.\, x : A \to B \to A$$

Takes two arguments; returns the first, discarding the second.

Type Preservation

Theorem (Subject Reduction)

If $\Gamma \vdash M : A$ and $M \ \rightarrow_\beta \ N$, then $\Gamma \vdash N : A$.

- Computation preserves types
- A well-typed program never "goes wrong" during reduction
- Proof by induction on the derivation of $M \ \rightarrow_\beta \ N$ (the key lemma: substitution preserves types)

Progress and Strong Normalisation

Theorem (Progress)

If $\varnothing \vdash M : A$ and M is not a normal form, then there exists N with $M \ \to_\beta \ N$.

Well-typed closed terms are never stuck.

Theorem (Strong Normalisation, Tait 1967)

Every well-typed term terminates: every reduction sequence eventually reaches a normal form.

Well-typed programs always halt. Proof method: **reducibility candidates** (Tait's method).

Y is Not Typable in STLC

The type equation has no solution

For $\boldsymbol{\omega} = \lambda x.\, x\, x$ to have type $A \to B$ we need:

- $x : A \quad$ and $\quad x\, x : B$
- So x must be of type $A \to B$, meaning $A = A \to B$
- No finite simple type satisfies $A = A \to B$

$\boldsymbol{\omega}$ is not typable $\Rightarrow$ $\mathbf{Y}$ (which uses $\boldsymbol{\omega}$) is not typable.

This is *by design*: ruling out self-application is exactly what guarantees termination.

Restoring Recursion

Several extensions add general recursion back in a typed setting:

- **fix constant.** $\mathsf{fix}_A : (A \to A) \to A$ with $\mathsf{fix}_A\, F \;\to_\beta\; F\,(\mathsf{fix}_A\, F)$. Breaks strong normalisation.
- **Recursive types (μ-types).** Allow type equations $\mu X.\, F[X]$. Makes self-application typable. Used in ML, Haskell.
- **System F (Girard 1972 / Reynolds 1974).** Universal type quantification $\forall X.\, A$. Strongly normalising; expresses all primitive recursive functions.
- **Dependent types.** Types depend on terms. Underpins Coq, Agda, Lean.

Summary

1. β-**reduction** is the sole computation rule. Church–Rosser: normal forms are unique.
2. **Church encodings** represent data as functions. Booleans, pairs, natural numbers, and arithmetic (including subtraction via the pairing trick) are all expressible.
3. $\boldsymbol{\Omega}$ shows that non-termination is native. Evaluation order determines whether a term terminates.
4. **Y combinator** gives every function a fixed point, enabling general recursion. Z is the call-by-value variant.
5. **Simple types** guarantee termination by ruling out self-application. Extensions (fix, μ, System F, dependent types) trade back safety for expressiveness.

Further Reading

- A. Church (1932, 1936, 1941) — original papers and monograph
- H.P. Barendregt — *The Lambda Calculus: Its Syntax and Semantics* (1984). The standard comprehensive reference.
- H.P. Barendregt — "Lambda calculi with types" in *Handbook of Logic in Computer Science*, vol. 2 (1992).
- B.C. Pierce — *Types and Programming Languages* (MIT Press, 2002). Excellent, accessible graduate textbook.
- W.W. Tait (1967) — original strong normalisation proof.
- J.-Y. Girard (1972) — System F (PhD thesis).
- S.C. Kleene (1935) — predecessor and pairing construction.