

Conversions and de Bruijn Indexing

α , β , η — and Life Without Variable Names

James Pustejovsky

Brandeis University
Department of Computer Science

CS 135: Computational Semantics

Fall 2026 (Session 1A — supplement to Session 1)

Why This Supplement

Session 1 introduced α , β , and η quickly, because the goal was to reach a semantic derivation before the hour ended.

They deserve more care, for two reasons:

- **They are easy to confuse.** All three are written with a squiggle and relate two terms. They do quite different work, and knowing which is which will matter on Quiz 1.
- **α is a genuine engineering problem.** Renaming bound variables sounds trivial. It is the single most common source of bugs in implementations of binding, and there is a standard fix worth knowing: **de Bruijn indexing**.

Today: the three conversions, properly distinguished and practiced; then variable names removed entirely.

Today

- 1 The three conversions, side by side
- 2 α : renaming, and the side condition
- 3 β : substitution, defined carefully
- 4 η : extensionality, and when it matters
- 5 Practice
- 6 Reduction strategies, and a term that never stops
- 7 **de Bruijn indexing**: variables as distances
- 8 β -reduction without names: shifting and substitution
- 9 What implementations actually do

The Three, Side by Side

Rule	Changes	Reading
$\alpha \quad \lambda x.M \equiv_{\alpha} \lambda y.M[x := y]$	the <i>name</i> of a bound variable	renaming
$\beta \quad (\lambda x.M) N \rightsquigarrow M[x := N]$	performs an <i>application</i>	computation
$\eta \quad \lambda x.(M x) \equiv_{\eta} M$	removes a <i>redundant wrapper</i>	extensionality

The one-line distinction:

- α says two terms *were already the same*; nothing happens.
- β *does* something – it is the computation step.
- η says a function that only forwards its argument *is* that function.

Note the arrows: β is usually written as a directed *reduction* because it makes progress. α and η are usually written as equivalences.

α : Renaming

α -conversion

$\lambda x.M \equiv_{\alpha} \lambda y.M[x := y]$ provided y does not occur free in M

Valid.

$$\lambda x.x \equiv_{\alpha} \lambda y.y \quad \lambda x.\lambda y.x y \equiv_{\alpha} \lambda a.\lambda b.a b \quad \lambda x.f x \equiv_{\alpha} \lambda z.f z$$

Invalid.

$$\lambda x.\lambda y.x \not\equiv_{\alpha} \lambda y.\lambda y.y \quad \lambda x.x y \not\equiv_{\alpha} \lambda y.y y$$

In the first, renaming x to y lets the inner binder **capture** it. In the second, y was free and has now been captured.

The side condition is the whole rule. Renaming is only safe when the new name is not already in use *free* inside the body.

β : Substitution, Carefully

β -reduction

$$(\lambda x.M) N \rightsquigarrow M[x := N]$$

The subterm $(\lambda x.M) N$ is called a **redex** (reducible expression).

Substitution is defined by recursion on the term:

$$x[x := N] = N$$

$$y[x := N] = y \quad (y \neq x)$$

$$(P Q)[x := N] = (P[x := N]) (Q[x := N])$$

$$(\lambda x.P)[x := N] = \lambda x.P \quad (x \text{ is rebound – stop})$$

$$(\lambda y.P)[x := N] = \lambda y.P[x := N] \quad \text{if } y \text{ not free in } N$$

$$(\lambda y.P)[x := N] = \lambda z.(P[y := z])[x := N] \quad \text{otherwise, } z \text{ fresh}$$

The last two lines are where α hides inside β : substitution has to rename on the fly to avoid capture. This is why the two rules cannot be separated in practice.

Why $(\lambda x.P)[x := N] = \lambda x.P$

This clause surprises people. Two things it is *not* saying.

It does not give N

$[x := N]$ replaces *free occurrences of the variable x* . Only the clause $x[x := N] = N$ ever produces N , and it fires only when the term *is* the variable x . Here the term is an abstraction, so a different clause applies.

It does not give $\lambda x.(P[x := N])$

x has no free occurrences in $\lambda x.P$. Every x inside P is bound by *that very* λx . Substitution touches only free occurrences, so there is nothing to replace.

This is variable shadowing, and you already rely on it:

```
let x = 1 in (\x -> x) 5    -- evaluates to 5, not 1
```

The inner binder wins. The outer x is invisible inside the lambda.

The Wrong Rule Gives the Wrong Answer

Reduce $(\lambda x. \lambda x. x) a b$ two ways.

Correct – the inner λx rebinds, so substitution stops:

$$\begin{aligned}(\lambda x. \lambda x. x) a &\rightsquigarrow (\lambda x. x)[x := a] = \lambda x. x \\(\lambda x. x) b &\rightsquigarrow b\end{aligned}$$

Wrong – recursing into the body anyway:

$$\begin{aligned}(\lambda x. \lambda x. x) a &\rightsquigarrow \lambda x. a \\(\lambda x. a) b &\rightsquigarrow a\end{aligned}$$

The two rules disagree on the final answer: b versus a . Only the first is right, because the x in the body is bound by the *inner* binder and therefore refers to the second argument, not the first.

The Three Abstraction Clauses, Reorganized

All three ask the same question – **what happened to x ?**

Situation	Action	Clause
x rebound by this binder	stop	$(\lambda x.P)[x := N] = \lambda x.P$
different binder, safe	recurse	$(\lambda y.P)[x := N] = \lambda y.P[x := N]$
different binder, would capture	rename, then recurse	$(\lambda y.P)[x := N] = \lambda z.(P[y := z])[x := N]$

The first row is the only case where substitution does **no work at all**.

A notational aside: some presentations define substitution directly in terms of $FV(M)$ rather than by cases on the term's shape. Under that definition this clause is not stipulated – it *follows*, since $x \notin FV(\lambda x.P)$ by construction.

Capture, Once More, Slowly

Reduce $(\lambda x. \lambda y. x y) y$.

Wrong. Substitute blindly:

$$(\lambda y. x y)[x := y] = \lambda y. y y \quad \times$$

The argument y was a *free* variable – some particular thing. Now it is bound by λy and means something else entirely.

Right. The final substitution clause applies, because y is free in the argument. Rename first:

$$(\lambda y. x y) \equiv_{\alpha} (\lambda z. x z) \quad \text{then} \quad (\lambda z. x z)[x := y] = \lambda z. y z \quad \checkmark$$

The result $\lambda z. y z$ is a function that applies the *external* y to its argument. The wrong answer $\lambda y. y y$ is self-application. **These are not remotely the same function.**

η : Extensionality

η -conversion

$\lambda x.(M x) \equiv_{\eta} M$ provided x does not occur free in M

Read it right to left: wrapping M in a lambda that immediately hands its argument to M changes nothing.

Examples.

$$\lambda x. f x \equiv_{\eta} f \qquad \lambda x. \text{sleep}(x) \equiv_{\eta} \text{sleep}$$

Side condition needed:

$$\lambda x. x x \not\equiv_{\eta} x$$

because x is free in the body $M = x$. The rule does not apply.

What η asserts. Two functions that agree on every argument are the same function. That is a substantive claim – it says functions are determined by their *behaviour*, not their *definition*.

Does η Matter for Semantics?

Usually not, and occasionally a lot.

In our fragment, these are η -equivalent:

```
sleeps = \x -> sleep x  
sleeps = sleep
```

Both are fine, and you will see both styles in the Problem Set 1 scaffold.

Where it does matter:

- **Normal forms.** Whether $\lambda x.f\ x$ counts as “already reduced” depends on whether η is in your calculus.
- **Comparing denotations.** Asking whether two lexical entries mean the same thing is asking whether they are $\beta\eta$ -equivalent.
- **Type-directed reasoning.** Some type systems use η -expansion to put terms into a canonical form before comparison.

An extensional theory of meaning is committed to η . An intensional one – Session 6 – may not be, since two functions can agree extensionally and differ in ways that matter for belief.

Which Conversion Is This?

Name the conversion relating each pair. Two minutes.

- ① $\lambda a. a b$ and $\lambda c. c b$
- ② $(\lambda x. x + 1) 5$ and $5 + 1$
- ③ $\lambda y. g y$ and g
- ④ $\lambda x. \lambda y. x$ and $\lambda p. \lambda q. p$
- ⑤ $(\lambda f. f a)(\lambda z. z)$ and $(\lambda z. z) a$

Which Conversion Is This?

Name the conversion relating each pair. Two minutes.

- ① $\lambda a. a\ b$ and $\lambda c. c\ b$
- ② $(\lambda x. x + 1)\ 5$ and $5 + 1$
- ③ $\lambda y. g\ y$ and g
- ④ $\lambda x. \lambda y. x$ and $\lambda p. \lambda q. p$
- ⑤ $(\lambda f. f\ a)(\lambda z. z)$ and $(\lambda z. z)\ a$

Answers.

- ① α – bound variable renamed; the free b untouched.
- ② β – an application was performed.
- ③ η – redundant wrapper removed.
- ④ α – both binders renamed consistently.
- ⑤ β – the outer redex was reduced.

The test: did anything get *computed* (β)? Was only a name changed (α)? Did a layer disappear (η)?

Practice: Reduce to Normal Form

- ① $(\lambda x. \lambda y. y x) a b$
- ② $(\lambda f. \lambda x. f (f x)) (\lambda y. y) c$
- ③ $(\lambda x. x) ((\lambda y. y) (\lambda z. z))$
- ④ $(\lambda x. \lambda y. x y) (\lambda w. w) d$

Practice: Reduce to Normal Form

- ① $(\lambda x. \lambda y. y x) a b$
- ② $(\lambda f. \lambda x. f (f x)) (\lambda y. y) c$
- ③ $(\lambda x. x) ((\lambda y. y) (\lambda z. z))$
- ④ $(\lambda x. \lambda y. x y) (\lambda w. w) d$

Answers.

- ① $(\lambda y. y a) b \rightsquigarrow b a$
- ② $(\lambda x. (\lambda y. y)((\lambda y. y) x)) c \rightsquigarrow (\lambda y. y)((\lambda y. y) c) \rightsquigarrow c$
- ③ $(\lambda x. x)(\lambda z. z) \rightsquigarrow \lambda z. z$
- ④ $(\lambda y. (\lambda w. w) y) d \rightsquigarrow (\lambda w. w) d \rightsquigarrow d$

Item 4 is worth a second look: the intermediate term $\lambda y. (\lambda w. w) y$ is η -equivalent to $\lambda w. w$, so you could have shortcut. Whether that is *allowed* depends on whether η is in your calculus.

A Term That Never Stops

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

Reduce it:

$$(\lambda x. x x) (\lambda x. x x) \rightsquigarrow (\lambda x. x x) (\lambda x. x x) \rightsquigarrow \dots$$

It reduces to itself. **There is no normal form.**

The **untyped** lambda calculus is Turing complete, so non-termination is unavoidable. The **simply typed** lambda calculus – the one we use for semantics – is *strongly normalizing*: every term reduces to a normal form, in finitely many steps, whatever order you choose.

Try to type $\lambda x. x x$. You need $x : \langle \sigma, \tau \rangle$ and $x : \sigma$ simultaneously, so $\sigma = \langle \sigma, \tau \rangle$ – an infinite type. Types rule out exactly the terms that misbehave, which is one more reason semantics uses them.

Reduction Strategies

When a term has several redexes, which do you reduce first?

Strategy	Reduces	Analogue
Normal order	leftmost, outermost first	lazy evaluation
Applicative order	leftmost, innermost first	eager evaluation

Example. $(\lambda x. a) \Omega$

- **Normal order:** reduce the outer redex, discarding Ω unevaluated $\rightsquigarrow a$. **Terminates.**
- **Applicative order:** evaluate the argument Ω first $\rightsquigarrow$ **never terminates.**

Church–Rosser guarantees that if two strategies both terminate, they agree. It does not guarantee both terminate. Normal order finds a normal form whenever one exists.

Haskell is lazy, so it is closer to normal order – which is why `undefined` in an unused position does not crash the Problem Set 1 test harness.

The Trouble With Names

Every implementation of the lambda calculus has to deal with α . That means:

- **Equality is not syntactic.** Deciding whether $\lambda x.\lambda y.x$ and $\lambda p.\lambda q.p$ are the same term requires an α -equivalence check, not a structural comparison.
- **Substitution must generate fresh names.** You need a supply of unused variables, and “unused” means unused *anywhere relevant* – a global condition inside a local operation.
- **Capture bugs are silent.** A missing freshness check does not crash. It produces a well-formed term with the wrong meaning.

What if we simply did not have names?

de Bruijn's Idea

Replace each variable occurrence by a **number**: how many binders you must cross, going outward, to reach the one that binds it.

Named	de Bruijn	Why
$\lambda x. x$	$\lambda. 0$	bound by the nearest binder
$\lambda x. \lambda y. x$	$\lambda. \lambda. 1$	cross one binder to reach λx
$\lambda x. \lambda y. y$	$\lambda. \lambda. 0$	bound by the nearest
$\lambda x. \lambda y. x y$	$\lambda. \lambda. 1\ 0$	
$\lambda x. x (\lambda y. x y)$	$\lambda. 0 (\lambda. 1\ 0)$	same x , two depths

Note the last row. The *same* variable gets *different* indices depending on depth. The index is a distance, not a name.

After N. G. de Bruijn, who introduced the notation in 1972 for Automath.

The Payoff

$\lambda x.\lambda y. x$ and $\lambda p.\lambda q. p$

are α -equivalent but syntactically different.

In de Bruijn notation, both are:

$\lambda.\lambda. 1$

α -equivalence becomes syntactic identity. Two terms are α -equivalent exactly when their de Bruijn representations are *the same string*. The α rule disappears – not because it was solved, but because it can no longer be stated.

This is the same design lesson we will meet again in Session 6A: choose a representation in which things that *mean* the same thing *are* the same thing. Canonical form eliminates a whole class of problems rather than managing it.

Free Variables Need a Context

$\lambda x. x y$ has a free y . What index does it get?

Free variables are numbered by **continuing to count outward** past the outermost binder, relative to a fixed naming context Γ .

With $\Gamma = [y, z]$ (so y is nearest, then z):

Named	de Bruijn	Reading
y	0	first in the context
z	1	second in the context
$\lambda x. x y$	$\lambda. 0\ 1$	inside one binder, y shifts up by one
$\lambda x. \lambda w. y$	$\lambda. \lambda. 2$	inside two binders

This is the price: the representation of a free variable depends on its depth. Closed terms – which is what our lexical entries are – avoid the issue entirely.

Practice: Translate

Convert to de Bruijn notation. All are closed.

① $\lambda x. \lambda y. \lambda z. x z (y z)$

② $\lambda f. \lambda x. f (f x)$

③ $\lambda x. (\lambda y. y) x$

④ $\lambda a. \lambda b. b (\lambda c. a)$

Practice: Translate

Convert to de Bruijn notation. All are closed.

① $\lambda x. \lambda y. \lambda z. x z (y z)$

② $\lambda f. \lambda x. f (f x)$

③ $\lambda x. (\lambda y. y) x$

④ $\lambda a. \lambda b. b (\lambda c. a)$

Answers.

① $\lambda. \lambda. \lambda. 2\ 0\ (1\ 0)$

② $\lambda. \lambda. 1\ (1\ 0)$

③ $\lambda. (\lambda. 0)\ 0$

④ $\lambda. \lambda. 0\ (\lambda. 2)$

Item 4: inside three binders, a is two levels out from λc , so index 2. Count binders crossed, not variables.

β -Reduction Without Names

Substitution now has to **renumber**, because removing a binder changes every distance beneath it.

Shifting $\uparrow_c^d$ – add d to every free index, with cutoff c

$$\uparrow_c^d(k) = \begin{cases} k & \text{if } k < c \text{ (bound below the cutoff)} \\ k + d & \text{otherwise} \end{cases}$$

$$\uparrow_c^d(\lambda. t) = \lambda. \uparrow_{c+1}^d(t) \quad (\text{one more binder, raise the cutoff})$$

$$\uparrow_c^d(t_1 t_2) = \uparrow_c^d(t_1) \uparrow_c^d(t_2)$$

Intuition. The cutoff c separates indices that point *inside* the term being moved (leave alone) from those that point *outside* it (adjust).

This is the bookkeeping that variable names were hiding. It has not gone away – it has become arithmetic, which is at least mechanical and testable.

Substitution and the β Rule

Substitution $[j \mapsto s] t$ – replace index j by term s

$$\begin{aligned}[j \mapsto s] k &= s \text{ if } k = j, \text{ else } k \\[j \mapsto s] (\lambda. t) &= \lambda. [j + 1 \mapsto \uparrow_0^1(s)] t \\[j \mapsto s] (t_1 t_2) &= ([j \mapsto s] t_1) ([j \mapsto s] t_2)\end{aligned}$$

Going under a binder: the target index goes up by one, *and* the substituted term must be shifted, since it is now one level deeper.

β -reduction

$$(\lambda. t_1) t_2 \rightsquigarrow \uparrow_0^{-1} \left([0 \mapsto \uparrow_0^1(t_2)] t_1 \right)$$

Shift the argument up (it is entering a binder), substitute for index 0, then shift the whole result down (the binder is gone).

A Worked Reduction

Reduce $(\lambda. 0\ 0)\ (\lambda. 0)$ – that is, $(\lambda x. x\ x)(\lambda y. y)$.

Here $t_1 = 0\ 0$ and $t_2 = \lambda. 0$.

Step 1 – shift the argument up.

$$\uparrow_0^1 (\lambda. 0) = \lambda. \uparrow_1^1 (0) = \lambda. 0$$

(index 0 is below the cutoff 1, so unchanged)

Step 2 – substitute for index 0.

$$[0 \mapsto \lambda. 0] (0\ 0) = (\lambda. 0)\ (\lambda. 0)$$

Step 3 – shift the result down.

$$\uparrow_0^{-1} ((\lambda. 0)(\lambda. 0)) = (\lambda. 0)\ (\lambda. 0)$$

Result: $(\lambda. 0)(\lambda. 0)$, which reduces again to $\lambda. 0$ – the identity. Matches $(\lambda x. x\ x)(\lambda y. y) \rightsquigarrow \lambda y. y$. ✓

What Have We Bought?

	Named	de Bruijn
α -equivalence	a relation to check	string equality
Fresh name supply	required	not needed
Capture avoidance	a side condition to remember	built into shifting
Substitution	rename-then-substitute	arithmetic
Readability	good	poor
Error messages	meaningful names	numbers

The trade is clear. Correctness and canonicity, at the cost of human legibility.

So the standard engineering answer is: **parse into de Bruijn, compute in de Bruijn, print back with names.** The user never sees an index; the implementation never sees a name.

Variations You May Meet

Scheme	Idea	Used because
de Bruijn <i>indices</i>	count binders from the <i>inside</i> out	canonical, standard
de Bruijn <i>levels</i>	count from the <i>outside</i> in	stable under weakening
Locally nameless	indices for bound, names for free	best of both; common in proof assistants
HOAS	reuse the host language's binders	substitution comes free

Locally nameless is worth knowing about: bound variables get indices (so α is trivial), free variables keep names (so error messages stay readable). Coq and many Agda developments use it.

HOAS – higher-order abstract syntax – represents object-language binders using the *implementation* language's lambda. You get capture-avoiding substitution for free, at the cost of no longer being able to inspect the body structurally.

Why You Will Not Do This in Problem Set 1

Reassuring news: the scaffold requires none of this machinery.

```
likes :: TV  
likes = \obj subj -> (subj, obj) 'elem' likePairs
```

Haskell's own binding does the work. When you write `\x -> ...`, GHC handles renaming, capture avoidance, and substitution. You are borrowing a correct implementation.

That is precisely the argument for writing semantics in a functional language: the lambda calculus you are studying *is* the language you are writing in, so its metatheory is the compiler's problem rather than yours.

GHC does not use de Bruijn indices internally – it uses names paired with unique integers – but the problem it is solving is the one on the last several slides.

Forward: Binding Will Come Back

Two threads from today run to the end of the course.

- 1 **Canonical representations matter.** de Bruijn eliminates α by making equivalent things identical. Session 6A shows the same move in λ -DCS, where implicit variables eliminate spurious ambiguity for a semantic parser. Same lesson, different century.
- 2 **Variable binding is a real mechanism, not a triviality.** It needs shifting, freshness, and capture avoidance – an entire apparatus. When Part IV proposes that the wedge product implements “binding”, the question to ask is: *which* binding?

The wedge binds a role to a filler. λ binds a *variable*, with capture avoidance and the ability to be used twice. Those are different operations sharing a word, and whether the geometric one can do the work of the logical one is an open question – and a candidate final project.

Summary

Concept	Key point
α	renaming a bound variable; nothing is computed
β	the computation step; substitution with capture avoidance
η	extensionality; a pure forwarder <i>is</i> the function
Substitution	recursive; the hard clause renames on the fly
Ω	untyped terms may not normalize; typed ones always do
Normal order	outermost first; finds a normal form if one exists
de Bruijn index	distance to the binder, counted in binders crossed
Payoff	α -equivalence becomes string equality
Shifting	renumbering when a term crosses a binder
β in de Bruijn	shift up, substitute at 0, shift down
Practice	parse to indices, compute, print with names

Reading and Exercises

Reading:

- EU Chapter 2, for the conversions in the semantic setting
- Pierce, *Types and Programming Languages*, Ch. 6 – the definitive short treatment of de Bruijn indices and shifting
- Barendregt, *The Lambda Calculus* – if you want the full metatheory

Exercises (not collected, but Quiz 1 will resemble them):

- 1 Translate the Session 1 practice terms into de Bruijn notation and reduce them there. Check that the answers agree.
- 2 Show $\uparrow_0^{-1} (\uparrow_0^1 (t)) = t$ for any t . Why does the reverse order fail?
- 3 Give two terms that are β -equivalent but not η -equivalent, and two that are η -equivalent but not β -equivalent.

Questions?

Main track resumes: Session 2 — Semantic Types