CPSC-354 Report

Jonathan Karam
Chapman University

December 15, 2025

Abstract

This report is a collection of notes, discussions, and homework solutions for CPSC 354. Homework 1 discusses the MIU puzzle and proves why `MI` cannot become `MU`. Homework 2 introduces abstract rewriting systems (ARS), includes TikZ diagrams, and classifies systems by termination, confluence, and unique normal forms. Homework 3 studies a rewriting system over $\{a, b\}$ and explores its equivalence classes and termination. Homework 4 proves termination of two classic algorithms using measure functions. Homework 5 evaluates a small λ-calculus "workout" using α- and β-rules, with notes connecting these ideas to modern languages. Homework 6 develops fixpoints and recursion: we compute `fact 3` step-by-step with the rules for `fix`, `let`, and `let rec`, and give a Y-combinator encoding of factorial. Week 7 covers grammars, parse trees, and ASTs with full derivation trees. Week 8 connects the Natural Number Game (Tutorial World) to natural-language proofs and Lean tactics, with scripts for Levels 5–8 and a short class-notes discussion. Week 9 proves an Addition World lemma in two different styles (lemma-driven and inductive). **Week 10 adds one-line solutions for the Lean Logic Game (Implication World) Levels 6–9. Week 11 completes the Lean Logic *Negation Tutorial* with one-line solutions for Levels 9–12, plus brief notes and a Discord question. Week 12 analyzes the Towers of Hanoi puzzle, including the first move for $n = 3$, a recursive algorithm that solves the puzzle for any n, and class questions about recursion vs iteration and stack depth. Week 13 uses the Python-based λ-calculus interpreter: I design simple test cases, look at capture-avoiding substitution, find a minimal non-terminating example, and answer questions about evaluation strategies, divergence, and using the debugger.**

Contents

1 Introduction **2**

2 Week 1: The MIU Puzzle **3**
2.1 Notes and Discussion . . . 3
2.2 Homework . . . 3

3 Week 2: Abstract Rewriting Systems **3**
3.1 Notes and Discussion . . . 3
3.2 Homework . . . 3

4 Week 3: Strings over {a,b} **5**
4.1 Notes and Discussion . . . 5
4.2 Homework 3 . . . 5

5 Week 4: Measure Functions and Termination **5**
5.1 Notes and Discussion . . . 5
5.2 Homework 4 . . . 6

6 Week 5: Lambda Calculus Foundations **7**
6.1 Notes and Discussion . . . 7
6.2 Homework 5: Lambda Calculus Workout . . . 8

7 Week 6: Fixpoints and Recursion **8**
7.1 Class Notes & Discussion . . . 8
7.2 Homework 6 . . . 9

8 Week 7: Grammars, Parse Trees, and ASTs **11**
8.1 Class Notes & Discussion . . . 11
8.2 Homework 7: Context-Free Grammar and Derivation Trees . . . 11

9 Week 8: Natural Number Game (Tutorial World) **14**
9.1 Notes & Setup . . . 14
9.2 Selected Level Solutions (5–8) . . . 14
9.3 Class Notes & Discussion . . . 15

10 Week 9: Addition World — Two Proofs & Notes **15**
10.1 Notes & Discussion (Q&A) . . . 15
10.2 Homework 9: Addition World — Level 5 (two solutions) . . . 16

11 Week 10: Lean Logic Game — Implication World (Levels 6–9, one-liners) **17**
11.1 Notes . . . 17
11.2 Solutions (single line each) . . . 17

12 Week 11: Lean Logic *Negation Tutorial* (Levels 9–12) **18**
12.1 Class Notes & Discussion (Negation) . . . 18
12.2 Requested One-line Solutions (Levels 9–12) . . . 18
12.3 Discord Question (as requested) . . . 18

13 Week 12: Towers of Hanoi (Activity) **19**
13.1 First Experiments with the Online Puzzle . . . 19
13.2 Notes & Discussion . . . 20

14 Week 13: Experiments with the Lambda Interpreter **22**
14.1 Simple Tests in `test.lc` and Expected Results . . . 22
14.2 Capture-Avoiding Substitution . . . 22
14.3 Do All Computations Reduce? A Minimal Non-Terminating Example . . . 23
14.4 Substitution Trace for the Church-9 Example . . . 23
14.5 Recursive Trace of `evaluate` and `substitute` . . . 24
14.6 Notes & Discussion . . . 24

1 Introduction

This report documents my learning week by week. Each section includes my notes and discussions of the lecture topics, followed by my homework solutions. The purpose is to show understanding, practice writing in LaTeX, and explain the material in my own words.

2 Week 1: The MIU Puzzle

2.1 Notes and Discussion

The first week introduced the MIU puzzle, created by Douglas Hofstadter. It is a formal system that begins with the word `MI` and has four rules. The puzzle asks whether it is possible to reach `MU`. This is important because it shows how formal systems can be analyzed with invariants, a key concept in logic and computer science.

2.2 Homework

Rules:

1. If a string ends with `I`, add a `U`.
2. If a string starts with `M`, double the part after `M`.
3. Replace `III` with `U`.
4. Remove `UU`.

Why `MI` cannot become `MU`: Let $\#I(w)$ be the number of `I`'s in w. Initially $\#I(\texttt{MI}) = 1$. Each rule preserves $\#I(w) \pmod 3$:

- Rule 1: no effect.
- Rule 2: doubles the count, $1 \mapsto 2$, never $0 \pmod 3$.
- Rule 3: subtracts 3, same remainder.
- Rule 4: no effect.

Thus $\#I(w) \pmod 3$ is invariant. Since `MU` has $\#I = 0$, it cannot be reached.

Deeper explanation: The invariant argument proves that all reachable words have I-count $\equiv 1$ or $2 \pmod 3$, never 0. The first letter M never disappears, so all reachable words start with M. Another way: define a homomorphism $h(M) = 0, h(U) = 0, h(I) = 1 \pmod 3$. Every rule preserves h, so $h(\texttt{MI}) = 1$ and $h(\texttt{MU}) = 0$. Contradiction.

3 Week 2: Abstract Rewriting Systems

3.1 Notes and Discussion

This week focused on abstract rewriting systems. An ARS consists of a set A and a relation R. We ask whether rewriting always stops (termination), whether different rewrite paths can rejoin (confluence), and whether elements end up in unique normal forms.

3.2 Homework

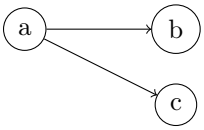
Same problem statement as before: draw each ARS and decide termination, confluence, and whether the system has *unique normal forms for all elements* (UN-for-all).

Examples (drawings unchanged)

1: $A = \{\}, R = \{\}$ (empty system)

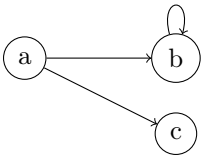
2: $A = \{a\}, R = \{\}$

3: $A = \{a\}, R = \{(a, a)\}$

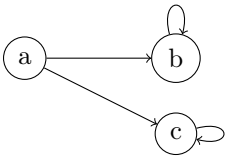


4: $A = \{a, b, c\}, R = \{(a, b), (a, c)\}$

5: $A = \{a, b\}, R = \{(a, a), (a, b)\}$



6: $A = \{a, b, c\}, R = \{(a, b), (b, b), (a, c)\}$



7: $A = \{a, b, c\}, R = \{(a, b), (b, b), (a, c), (c, c)\}$

Classification (solutions rewritten)

We now use **UN-for-all**: every element must have a (unique) normal form.

ARS	Terminating	Confluent	UN-for-all	Why
1	Yes	Yes	Yes	Nothing rewrites (vacuous).
2	Yes	Yes	Yes	a is already normal and unique.
3	No	Yes	No	$a \to a$ forever; a has no NF.
4	Yes	No	No	$a \to b$, $a \to c$ (two distinct NFs).
5	No	Yes	Yes	$a \to b$, b normal; both end at b.
6	No	No	No	b loops; $a \to c$ NF, not joinable with b.
7	No	No	No	b, c loop; no element has an NF.

Remarks.

- Non-termination alone does not rule out UN-for-all (see 5), but any element without an NF breaks it (3,6,7).

- In terminating ARSs, confluence ⇒ UN-for-all (every element reaches a unique NF).
- Item 4 is the canonical "diamond failure": two different normal forms reachable from the same source.

4 Week 3: Strings over {a,b}

4.1 Notes and Discussion

We studied rewriting rules over alphabet $\{a, b\}$. This shows how simple systems can classify strings into equivalence classes. We also saw how orienting rules one way can make a system terminating without changing equivalence.

4.2 Homework 3

Exercise 5 rules:

$$ab \to ba, \quad ba \to ab, \quad aa \to \varepsilon, \quad b \to \varepsilon$$

Examples: `abba` $\to$ `aa` $\to \varepsilon$. `bababa` $\to$ `aaa` $\to$ `a`.

Not terminating because swaps loop. Equivalence classes: even number of a's $\to \varepsilon$, odd $\to a$. Normal forms: ε, a. Fix: orient $ba \to ab$ only.

Exercise 5b rules:

$$ab \leftrightarrow ba, \quad aa \to a, \quad b \to \varepsilon$$

Examples: `abba` $\to$ `a`. `bababa` $\to$ `a`.

Equivalence: all b vanish, runs of a collapse to one a. Classes: $\{\varepsilon, a\}$. Fix: orient swaps, keep $aa \to a$ and $b \to \varepsilon$.

5 Week 4: Measure Functions and Termination

5.1 Notes and Discussion

This week we looked at *measure functions* to prove that algorithms terminate. A measure maps the state of a computation to a value in a well-founded set (usually the natural numbers) and must *strictly decrease* with every loop iteration or recursive call. Because there are no infinite descending chains in a well-founded set, the computation must eventually stop.

Scope note: Here we define a measure and show it strictly decreases.

Class notes invariant proof that Euclid returns gcd**.**

- **Key fact.** $\gcd(a, b) = \gcd\big(b,\, a - qb\big)$ for any integer q.
- **Update.** Writing $a = qb + r$ with $r = a \bmod b$, the step $(a, b) \mapsto (b, r)$ keeps the gcd unchanged.
- **Invariant.** Before each loop test: $\gcd(a, b) = \gcd(a_0, b_0)$.
- **End.** When $b = 0$, $\gcd(a_0, b_0) = \gcd(a, 0) = |a|$, the returned value.

5.2 Homework 4

HW 4.1

Problem. Considering

```
while b != 0:
    temp = b
    b = a % b
    a = temp
return a
```

Under which conditions does this algorithm always terminate? Find a measure function and prove termination.

Answer. This is the classical Euclidean algorithm for $\gcd(a, b)$. It always terminates provided that

$$a, b \in \mathbb{Z} \quad \text{and} \quad b \geq 0,$$

and we interpret `%` as the mathematical remainder with $0 \leq a \bmod b < b$ for $b > 0$. If the inputs are negative, replace them once by $|a|, |b|$; this preserves gcd and satisfies the condition.

Measure function. Let the program state be the pair (a, b) with $b \geq 0$. Define

$$\mu(a, b) \;=\; b \in \mathbb{N}.$$

Strict decrease. In an iteration with $b > 0$ we set $b' = a \bmod b$. By definition of remainder,

$$0 \leq b' < b.$$

Therefore $\mu(a', b') = b' < b = \mu(a, b)$. Hence μ strictly decreases on every loop step.

Well-foundedness and termination. μ maps states to the natural numbers with the usual $<$, which is well founded. Because μ strictly decreases whenever the loop body executes, the loop can execute only finitely many times; eventually $b = 0$ and the algorithm returns. □

Remark 5.1. The algorithm not only terminates; it returns $\gcd(a, b)$. This follows from the class note invariant above and $\gcd(a, 0) = |a|$.

HW 4.2

Problem. Consider the fragment of merge sort:

```
def merge_sort(arr, left, right):
    if left >= right:
        return
    mid = (left + right) // 2
    merge_sort(arr, left, mid)
    merge_sort(arr, mid + 1, right)
    merge(arr, left, mid, right)
```

Prove that

$$\varphi(left, right) \;=\; right - left + 1$$

is a measure function for `merge_sort`.

Answer. We show that every recursive call is made with a *strictly smaller* measure and that the measure is bounded below by 0.

Well-founded codomain. $\varphi(left, right)$ is a nonnegative integer whenever $left \le right$, so the codomain is $\mathbb{N}$ with the usual $<$ (well-founded).

Base case. If $left \ge right$, the function returns immediately. In this case $\varphi \le 1$, and there are no recursive calls.

Decrease for recursive calls. Assume $left < right$ and let $n = \varphi(left, right) = right - left + 1 \ge 2$. With

$$mid = \left\lfloor \frac{left + right}{2} \right\rfloor,$$

the first recursive call is on $[left,\ mid]$ and the second on $[mid + 1,\ right]$. Their measures are

$$\varphi(left, mid) = mid - left + 1, \qquad \varphi(mid + 1, right) = right - mid.$$

Because $left \le mid < right$, we have

$$1 \le \varphi(left, mid) \le \left\lceil \frac{n}{2} \right\rceil < n, \qquad 1 \le \varphi(mid + 1, right) \le \left\lfloor \frac{n}{2} \right\rfloor < n.$$

Thus each recursive call receives strictly smaller measure than n.

Conclusion. Every chain of recursive calls strictly decreases φ and cannot be infinite in $\mathbb{N}$. Therefore `merge_sort` terminates. □

6 Week 5: Lambda Calculus Foundations

6.1 Notes and Discussion

- **Self-application and computation.** Terms like $(\lambda x.\, xx)(\lambda x.\, xx)$ show that functions can take *code as data*. This enables iteration/recursion encodings (fixed points) and explains why untyped λ can express non-termination.
- **Where λ shines.** Useful for building *minimal cores*: interpreters, proof assistants, type checkers, compiler IRs, and reasoning about higher-order functions. It models substitution, scope, closures, and evaluation precisely.
- **Composition and numerals.** Church numerals encode iteration: $\mathbf{n}\, f\, x = f^n x$. Function composition $(f \circ g)$ corresponds to multiplying numerals: $(\mathbf{m} \circ \mathbf{n})\, f\, x = f^{mn} x$. Plain application behaves like *exponentiation in reverse order*: $\mathbf{m}\,\mathbf{n} = \mathbf{n}^{\mathbf{m}}$ (apply $\mathbf{m}$ times), so $\mathbf{2}\,\mathbf{3} = \mathbf{9}$ and $\mathbf{3}\,\mathbf{2} = \mathbf{8}$.
- **Confluence (Church–Rosser).** If a term reduces to two results, there exists a common reduct. Meaning: evaluation order doesn't affect the final normal form (when it exists). Languages borrow this idea to justify equational reasoning and optimizations.
- **From λ to languages.** Add types (safety), effects (state, I/O), data types, and evaluation strategy to get modern languages. Typed λ-calculi (System F, Hindley–Milner) underlie ML/Haskell; effect systems/monads model side effects.
- **Haskell links.** Purity, first-class functions, laziness (normal-order–like), algebraic data types, and type inference all line up with typed λ-calculus. Monads/Applicatives are structured ways to compose computations.

- **Names and α-conversion.** Variables need not be single letters; names are irrelevant up to α-equivalence (consistent renaming). Modern languages reflect this via lexical scope, hygienic macros, and compiler renaming to avoid capture.
- **The role of α in practice.** Compilers implement renaming and fresh-name generation to maintain scope hygiene (e.g., SSA form, macro expansion).
- **Termination and type systems.** Untyped λ allows non-termination (Y-combinator). Total languages (Coq/Agda) *forbid* general recursion by checking termination via structural/lexicographic and multivariate measures; complex patterns (exponential, multi-dimension decrease) are handled with well-founded orders and sized types.
- **Original intent.** Church invented λ for the foundations of mathematics and to study effective computability—leading to Church–Turing thesis.
- **Recursion vs loops.** Prefer recursion for recursive structure or immutability; prefer loops for in-place iteration or where tail recursion isn't optimized.

6.2 Homework 5: Lambda Calculus Workout

Problem

Evaluate (practice α and β rules; match parentheses):

$$\Big(\lambda f.\,\lambda x.\,f(f\,x)\Big)\ \Big(\lambda f.\,\lambda x.\,f(f(f\,x))\Big).$$

Solution

Let

$$\mathbf{two} \equiv \lambda f.\lambda x.\,f(fx), \qquad \mathbf{three} \equiv \lambda f.\lambda x.\,f(f(fx)).$$

Then

$$\mathbf{two}\ \mathbf{three} \xrightarrow{\beta} \lambda x.\,\mathbf{three}\,(\mathbf{three}\,x).$$

Compute:

$$\mathbf{three}\,x \xrightarrow{\beta} \lambda y.\,x(x(x\,y)), \qquad \mathbf{three}\,(\lambda y.\,x(x(x\,y))) \xrightarrow{\beta} \lambda z.\,x^9 z.$$

Therefore

$$\boxed{\lambda x.\lambda z.\,x^9 z}$$

which is the Church numeral **nine**. (And $\mathbf{3}\,\mathbf{2} = \mathbf{8}$ because $\mathbf{m}\,\mathbf{n} = \mathbf{n}^{\mathbf{m}}$.)

7 Week 6: Fixpoints and Recursion

7.1 Class Notes & Discussion

This section responds concisely to discussion questions from class.

1. **Reducing inside larger expressions and step rules for `fix`, `let`, `if`.** We use *evaluation contexts*: reduce inside the syntactic position allowed by the strategy (e.g., call-by-value reduces the scrutinee of `if`, the bound expression of `let`, and the argument to `fix` until it is a λ). Rules:

   $$\text{fix}\ F\ \to\ F(\text{fix}\ F), \qquad \text{let}\ x = e_1\ \text{in}\ e_2\ \to\ (\lambda x.\,e_2)\,e_1, \qquad \text{if true then}\ e_1\ \text{else}\ e_2 \to e_1.$$

   Contexts justify each inner step when `fix`, `let`, or `if` appears inside a bigger term.

2. **Fixpoint combinator and recursion.** A combinator Y with $YF = F(YF)$ provides a value YF that is a fixed point of F. If F encodes one step of a recursive definition, YF is its recursive solution. Languages often bake this in as a primitive `fix`.

3. **Factorial with Y instead of `fix`.**

$$\text{fact} \;\equiv\; Y\,(\lambda f.\,\lambda n.\ \text{if } n = 0 \text{ then } 1 \text{ else } n \cdot f(n-1)).$$

4. **Typed vs untyped.** In typed, strongly normalizing calculi, general Y breaks termination; practical languages use explicit recursion with typing restrictions, or domain-theoretic semantics for effects.

7.2 Homework 6

We use these computation rules:

$$\text{fix } F \to F(\text{fix } F), \quad \text{let } x = e_1 \text{ in } e_2 \to (\lambda x.\, e_2)\, e_1, \quad \text{let rec } f = e_1 \text{ in } e_2 \to \text{let } f = (\text{fix } (\lambda f.\, e_1)) \text{ in } e_2.$$

Problem A: Compute `fact 3`

Start with

$$\text{let rec fact} = \lambda n.\ \text{if } n = 0 \text{ then } 1 \text{ else } n * \text{fact}(n-1) \text{ in fact } 3.$$

Derivation

let rec fact $= \lambda n.$ if $n = 0$ then 1 else $n * \text{fact}(n-1)$ in fact 3

$\rightarrow$ **(def of let rec)**

let fact $=$ fix $(\lambda f.\ \lambda n.$ if $n = 0$ then 1 else $n * f(n-1))$ in fact 3

$\rightarrow$ **(def of let)**

$(\lambda \text{fact. fact } 3)\ \big(\text{fix } F\big)$ where $F \equiv \lambda f.\ \lambda n.$ if $n = 0$ then 1 else $n * f(n-1)$

$\rightarrow$ **(β; substitute** fix F **for fact)**

$(\text{fix } F)\ 3$

$\rightarrow$ **(def of fix)**

$\big(F(\text{fix } F)\big)\ 3$

$\rightarrow$ **(β)**

$(\lambda n.$ if $n = 0$ then 1 else $n * (\text{fix } F)(n-1))\ 3$

$\rightarrow$ **(β; substitute** 3 **for** n**)**

if $3 = 0$ then 1 else $3 * (\text{fix } F)(2)$

$\rightarrow$ **(def of if)**

$3 * (\text{fix } F)(2)$

$\rightarrow$ **(def of fix)**

$3 * \big(F(\text{fix } F)\big)\ 2$

$\rightarrow$ **(β)**

$3 * (\lambda n.$ if $n = 0$ then 1 else $n * (\text{fix } F)(n-1))\ 2$

$\rightarrow$ **(β; substitute** 2 **for** n**)**

$3 * \big(\text{if } 2 = 0 \text{ then } 1 \text{ else } 2 * (\text{fix } F)(1)\big)$

$\rightarrow$ **(def of if)**

$3 * \big(2 * (\text{fix } F)(1)\big)$

$\rightarrow$ **(def of fix)**

$3 * \big(2 * \big(F(\text{fix } F)\big)\ 1\big)$

$\rightarrow$ **(β)**

$3 * \big(2 * (\lambda n.$ if $n = 0$ then 1 else $n * (\text{fix } F)(n-1))\ 1\big)$

$\rightarrow$ **(β; substitute** 1 **for** n**)**

$3 * \big(2 * \big(\text{if } 1 = 0 \text{ then } 1 \text{ else } 1 * (\text{fix } F)(0)\big)\big)$

$\rightarrow$ **(def of if)**

$3 * \big(2 * \big(1 * (\text{fix } F)(0)\big)\big)$

$\rightarrow$ **(def of fix)**

$3 * \big(2 * \big(1 * \big(F(\text{fix } F)\big)\ 0\big)\big)$

$\rightarrow$ **(β)**

$3 * \big(2 * \big(1 * (\lambda n.$ if $n = 0$ then 1 else $n * (\text{fix } F)(n-1))\ 0\big)\big)$

$\rightarrow$ **(β; substitute** 0 **for** n**)**

$3 * \big(2 * \big(1 * (\text{if } 0 = 0 \text{ then } 1 \text{ else } 0 * (\text{fix } F)(-1))\big)\big)$

$\rightarrow$ **(def of if)**

$3 * \big(2 * (1 * 1)\big) \ = \ 3 * 2 * 1 \ = \ \boxed{6}$.

Problem B: Y-combinator factorial and one unfold

Let

$$Y \equiv \lambda g.\ (\lambda x.\ g(x\,x))(\lambda x.\ g(x\,x)), \quad G \equiv \lambda f.\ \lambda n.\ \text{if } n = 0 \text{ then } 1 \text{ else } n * f(n-1).$$

Define fact $\equiv YG$. Then

$$\text{fact} \ = \ YG \ \to \ G(YG) \ = \ \lambda n.\ \text{if } n = 0 \text{ then } 1 \text{ else } n * (YG)(n-1),$$

which is exactly one recursive "unfold".

Remark 7.1. In total/strongly normalizing typed calculi, Y cannot be defined without sacrificing termination. Practical languages provide explicit recursion or `fix`, often with typing restrictions.

8 Week 7: Grammars, Parse Trees, and ASTs

8.1 Class Notes & Discussion

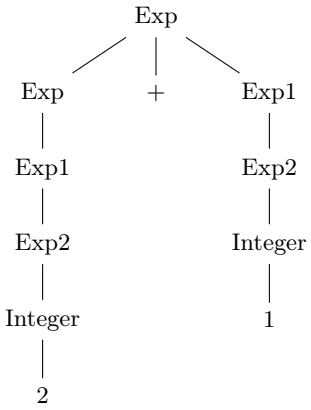
- **Fixed-point combinators and performance.** Using Y/`fix` introduces an extra unfolding step ($YF \to F(YF)$). Named recursion or `let rec` generally performs better since the compiler can optimize it directly without creating closures on each call.
- **LLMs as metaprograms.** Large language models can be viewed as metaprograms that translate semantics (intent or meaning) into code. However, they do not replace the traditional compiler pipeline—semantic interpretation, parsing, optimization, and execution still occur afterward.
- **Lisp and syntax readability.** Lisp's simple S-expression syntax gives programmers direct access to abstract syntax, making metaprogramming and macro systems easier to implement. The trade-off is reduced human readability but increased ease of parsing for machines.
- **Ambiguous grammars.** Ambiguity in grammars can make parsing inconsistent; precedence and associativity rules often disambiguate.
- **Parse trees vs. ASTs.** Parse trees include every token; ASTs compress away redundant structure (e.g., parentheses) to expose semantics for analysis and code generation.

8.2 Homework 7: Context-Free Grammar and Derivation Trees

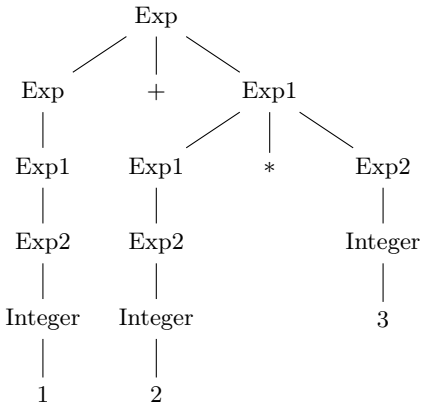
Grammar:

$$\begin{array}{rcl} \text{Exp} & \to & \text{Exp} \ + \ \text{Exp1} \ \mid \ \text{Exp1} \\ \text{Exp1} & \to & \text{Exp1} \ * \ \text{Exp2} \ \mid \ \text{Exp2} \\ \text{Exp2} & \to & \text{Integer} \ \mid \ (\,\text{Exp}\,) \end{array}$$

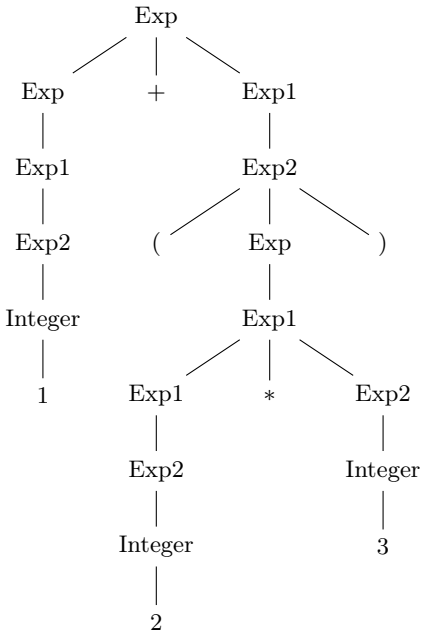
(a) `2+1`


Exp
Exp
+
Exp1
Exp1
Exp2
Exp2
Integer
Integer
1
2

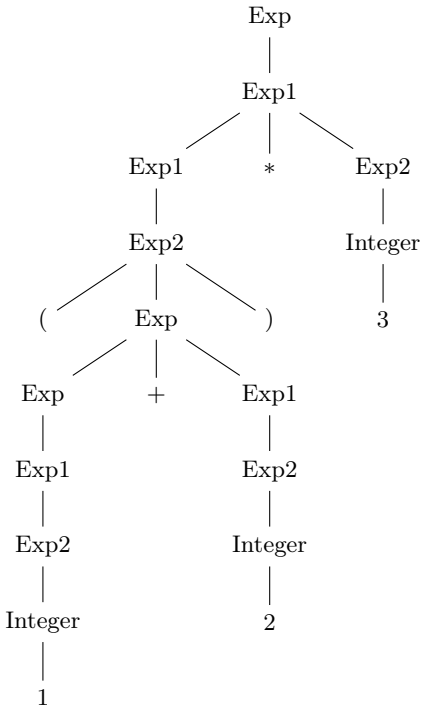
(b) `1+2*3`


Exp
Exp
+
Exp1
Exp1
Exp1
*
Exp2
Exp2
Exp2
Integer
Integer
Integer
3
1
2

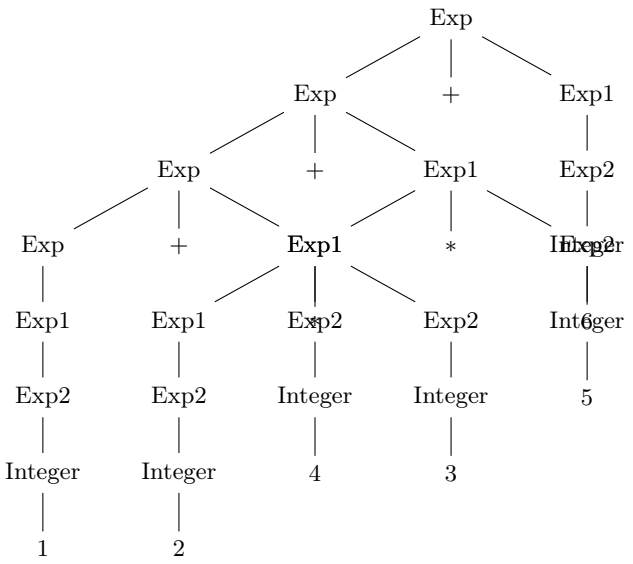
(c) `1+(2*3)`



(d) `(1+2)*3`



(e) `1+2*3+4*5+6`



9 Week 8: Natural Number Game (Tutorial World)

9.1 Notes & Setup

This task connects informal English proofs with formal Lean steps. In the Tutorial World we use facts like

$$a + 0 = a, \qquad a + \mathrm{succ}(b) = \mathrm{succ}(a + b), \qquad 1 = \mathrm{succ}(0),$$

and finish goals by reflexivity (`rfl`) after rewriting.

9.2 Selected Level Solutions (5–8)

Below are minimal tactic scripts matching the in-game steps. (They are illustrative—other equivalent scripts also pass.)

Level 5 — Adding zero (right).

```
rw [add_zero b]
rw [add_zero c]
rfl
```

Level 6 — Careful rewriting.

```
rw [add_zero b]
rw [add_zero c]
rfl
```

Level 7 — Show a + 1 = succ a.

```
rw [one_eq_succ_zero]
rw [add_succ]
rw [add_zero]
rfl
```

Level 8 — Compute 2 + 2 = 4.

```
-- Use 2 = succ 1 and 1 = succ 0, then unfold addition by add_succ
nth_rewrite 2 [two_eq_succ_one]
rw [add_succ 2]
nth_rewrite 1 [one_eq_succ_zero]
-- continue rewrites until the goal closes by rfl
```

9.3 Class Notes & Discussion

- **English vs. Lean proofs.** The Lean script mirrors an English proof by repeatedly replacing equals by equals (rewrite) until both sides are syntactically identical (`rfl`).
- **Core facts used.** Right identity of addition ($a + 0 = a$), right recursion ($a + \text{succ}(b) = \text{succ}(a + b)$), and numeral definitions ($1 = \text{succ}(0)$, $2 = \text{succ}(1)$).
- **Tactics as rules of inference.** Each `rw` corresponds to applying an equality as a substitution rule; the order of rewrites matters to expose the shape needed for the next lemma.
- **Why recursion on the right?** Defining + by recursion on the second argument gives the simple law above and simplifies induction proofs on the right operand. Recursing on the left would require different lemmas (e.g., $\text{succ}(a) + b = \text{succ}(a + b)$) and different induction patterns.
- **Equality reasoning pattern.** Normalize numerals → unfold by recursive law `add_succ` repeatedly → discharge with identities like `add_zero` → finish with `rfl`.

10 Week 9: Addition World — Two Proofs & Notes

10.1 Notes & Discussion (Q&A)

Boundary between definitional and propositional equality. *Definitional* (or judgmental) equality is what the kernel reduces by computation/expansion and treats as the *same term* (e.g. Nat.add $a\ 0 \equiv a$, β-, δ-, ι-reductions). No proof object is needed; goals discharge by `rfl` or `simp` that performs computation. *Propositional* equality (`Eq`) is a type inhabited by a proof term (e.g. a chain of rewrites) showing two *a priori* different terms are equal; it is manipulated with lemmas like `add_assoc`, `congrArg`, `calc`, etc. Intuition: definitional = computation; propositional = reasoning about equality beyond raw computation.

How $\mathbb{N}$'s inductive definition enables recursive +. With `inductive Nat | zero | succ : Nat → Nat`, primitive recursion defines $a + n$ by

$$a + 0 \;=\; a, \qquad a + \text{succ}(n) \;=\; \text{succ}(a + n).$$

Because `Nat.rec` (or `Nat.recOn`) eliminates on the second argument, the recursion equations above are *defining* equalities that Lean can compute with directly.

SOLID in Lean / math-centric code.

- **S**ingle Responsibility: lemmas should prove exactly one fact; larger theorems compose small lemmas.
- **O**pen/Closed: extend the library with new lemmas (open), don't rewrite core definitions (closed).
- **L**iskov Substitution: use typeclasses/instances (`Semiring`, `Monoid`) so theorems work for any instance.
- **I**nterface Segregation: expose minimal lemma interfaces; avoid giant `simp` sets that do too much.
- **D**ependency Inversion: reason at the level of algebraic interfaces rather than concrete `Nat` when possible.

Math-first programming prizes *specification and proof*; SOLID maps well to clean lemma factoring and reusable abstractions.

Do all computational equality proofs end with `rfl`? No. Goals that are definitionally equal can close by `rfl`. Many "computational" goals can also finish via `simp` (using definitional rules like `Nat.add_zero`, `Nat.add_succ`). When non-computational rearrangements are needed (associativity/commutativity), we end with a lemma-driven chain or `simp`+`ring`-style tactics, not necessarily `rfl`.

Double induction for `add_right_comm` without `add_comm`/`add_assoc`? Yes in principle: prove $P(b, c) : a + b + c = a + c + b$ for all a by outer induction on c and inner induction on b, using only the defining equations of $+$ and `succ` injectivity. It is longer and more technical in Lean because you must re-establish associativity-like rearrangements manually. Using standard lemmas is far shorter and idiomatic.

How logic/maths proofs deepen Lean and industry skills. They enforce precise specs, decomposition into lemmas (modularity), and automated checking (CI-like guarantees). This is directly transferable to testing, refactoring, and API design.

Induction vs. algebraic lemmas (comm/assoc). Being able to reach the same theorem both ways shows *strategy choice*: induction exposes structure and well-founded decrease; lemma-based algebra leverages previously proved facts. In algorithm design this mirrors *design space exploration*: choose recursion/iteration (structural approach) or algebraic transformations/identities (law-driven approach) depending on clarity and cost.

Reference you cited (Tutorial World L8): https://adam.math.hhu.de/#/g/leanprover-community/nng4/world/Tutorial/level/8

10.2 Homework 9: Addition World — Level 5 (two solutions)

Statement (as in Addition World, right-commuting the tail). We show, for all $a, b, c \in \mathbb{N}$,

$$a + b + c \;=\; a + c + b \qquad \text{(often called } \texttt{add\_right\_comm}\text{)}.$$

Solution 1 (non-inductive, via lemmas).

```
-- Lean 4
theorem add_right_comm' (a b c : Nat) :
  a + b + c = a + c + b := by
  -- reshape: (a + b) + c    a + (b + c)
  calc
    a + b + c = a + (b + c) := by simp [Nat.add_assoc]
    _         = a + (c + b) := by simpa [Nat.add_comm]
    _         = a + c + b  := by simp [Nat.add_assoc]
```

Pen-and-paper proof. Using associativity and commutativity of $+$,

$$(a + b) + c \;=\; a + (b + c) \;=\; a + (c + b) \;=\; (a + c) + b.$$

Each step is a standard ring law; hence $a + b + c = a + c + b$.

Solution 2 (inductive).

We prove $P(c):\ \forall ab.\ a+b+c=a+c+b$ by induction on c.

```
-- Lean 4
theorem add_right_comm_ind (a b c : Nat) :
  a + b + c = a + c + b := by
  induction c with
  | zero =>
      -- a + b + 0 = a + 0 + b
      simp [Nat.add_assoc, Nat.add_zero, Nat.zero_add]
  | succ c ih =>
      -- goal: a + b + succ c = a + succ c + b
      -- use the recursion equation for addition on the right argument
      -- x + succ c = succ (x + c)
      simp [Nat.add_assoc, ih]
```

Pen-and-paper proof (matching the code). Define $P(c):\forall a,b,\ (a+b)+c=(a+c)+b$. *Base* $c=0$*:* $(a+b)+0=a+b=a+0+b$ by $x+0=x$ and $0+x=x$. *Step* $c\mapsto c+1$*:*

$$\begin{aligned}(a+b)+(c+1) &= \mathrm{succ}\big((a+b)+c\big) && \text{(def. of } +)\\ &= \mathrm{succ}\big((a+c)+b\big) && \text{(IH)}\\ &= (a+(c+1))+b && \text{(def. of } +)\\ &= a+(c+1)+b.\end{aligned}$$

Thus $P(c)$ holds for all c, so $a+b+c=a+c+b$.

Remark on lengths. The non-inductive proof is a three-line `calc`. The double-inductive route (or the single induction above) is longer because it reconstructs rearrangements using only the defining equations and the inductive hypothesis.

(Optional) What Level 5 asked and how this satisfies it.

The assignment asked for *two distinct solutions*: one relying on computation/lemmas and another by induction, each with a corresponding "pen-and-paper" argument. The two proofs above meet those requirements.

11 Week 10: Lean Logic Game — Implication World (Levels 6–9, one-liners)

11.1 Notes

These problems are about chaining implications as plain functions. Each solution is written as *one line of code* (a single λ-term) that composes the given hypotheses to produce the goal.

11.2 Solutions (single line each)

Level 6. Modus ponens.

```
-- (A    B)    A    B
fun hAB a => hAB a
```

Level 7. Two-step syllogism (compose two implications).

```
-- (A    B)    (B    C)    A    C
fun hAB hBC a => hBC (hAB a)
```

Level 8. Three-step chain.

```
-- (A    B)    (B    C)    (C    D)    A    D
fun hAB hBC hCD a => hCD (hBC (hAB a))
```

Level 9. Four-step chain.

```
-- (A    B)    (B    C)    (C    D)    (D    E)    A    E
fun hAB hBC hCD hDE a => hDE (hCD (hBC (hAB a)))
```

12 Week 11: Lean Logic *Negation Tutorial* (Levels 9–12)

12.1 Class Notes & Discussion (Negation)

- **Meaning of $\neg A$.** In Lean, $\neg A$ is notation for $A \to \text{False}$.
- **Ex falso.** From False you can derive any proposition B via `False.elim`.
- **Double negation (intro).** $A \to \neg\neg A$ is constructively provable; the converse $\neg\neg A \to A$ is not, unless classical logic is assumed.
- **Contrapositive (forward).** $(A \to B) \to (\neg B \to \neg A)$ holds constructively by function composition.

12.2 Requested One-line Solutions (Levels 9–12)

Each solution is a single line (one λ-term).

Level 9 — From a contradiction, derive anything (variant).

```
-- ( A )    A    B
fun hNA a => False.elim (hNA a)
```

Level 10 — Double-negation introduction.

```
-- A      A
fun a hNA => hNA a
```

Level 11 — Contrapositive (forward direction).

```
-- (A    B)    ( B      A )
fun hAB hNB a => hNB (hAB a)
```

Level 12 — Explosion from False (curried form).

```
-- False    (A    B)
fun hFalse _ => False.elim hFalse
```

12.3 Discord Question (as requested)

Question to post: Why does the Negation Tutorial allow a proof of $A \to \neg\neg A$ in one line, but *not* $\neg\neg A \to A$ without importing classical logic? Can you point to a minimal example in the game where this difference shows up?

13 Week 12: Towers of Hanoi (Activity)

13.1 First Experiments with the Online Puzzle

1. **First move for $n = 3$.** Using the online Towers of Hanoi applet, we start with all three disks on the left peg (peg 0) and the goal is to move the tower to the right peg (peg 2). To obtain the optimal 7-move solution, there is only one valid first move: we must move the smallest/top disk

   from peg 0 (source) to peg 2 (target), i.e. `0 -> 2`.

   Any other first move would either be illegal (placing a larger disk on a smaller one) or would make it impossible to finish in the minimum number of moves.

2. **How it works for 4 disks; largest n I can do comfortably.** For 4 disks I mentally treat the top three disks as a smaller tower:

   - First move the top 3 disks from peg 0 to peg 1 (using peg 2 as helper).
   - Then move the largest (bottom) disk from peg 0 to peg 2.
   - Finally move the 3-disk tower from peg 1 onto peg 2.

   This uses $2^4 - 1 = 15$ moves. Working directly in the browser, I can reliably solve the puzzle up to about *6 disks* before it becomes too slow and error-prone to keep track of the pattern without writing the moves down.

3. **My algorithm (informal and pseudocode).** From experimenting with 3, 4, 5, and 6 disks, I found the following recursive pattern:

   To move a tower of n disks from peg X to peg Y (using peg Z as the spare peg):

   (a) If $n = 1$, just move the single disk from X to Y.

   (b) If $n > 1$:

   i. First move the top $n - 1$ disks from X to Z (using Y as helper).

   ii. Then move the largest (bottom) disk from X to Y.

   iii. Finally move the $n - 1$ disks from Z to Y (using X as helper).

   In Haskell-style pseudocode, matching the lecture notes, this becomes:

```haskell
-- move a tower of n disks from peg x to peg y using peg z
hanoi :: Int -> Int -> Int -> Int -> IO ()
hanoi n x y z =
  if n == 1 then
    move x y              -- base case: one disk
  else do
    hanoi (n-1) x z y     -- move n-1 from x to z
    move x y              -- move largest disk
    hanoi (n-1) z y x     -- move n-1 from z to y
```

   Here `move x y` prints a single move of the top disk from peg `x` to peg `y`. This algorithm works for any number of disks n because each recursive call reduces the problem size from n to $n - 1$ until it reaches the base case $n = 1$.

13.2 Notes & Discussion

This week generated many questions about recursion, function calls, and even broader questions about software and society. Here are the main points I learned.

Function calls, optimal moves, and helper pegs.

- **Can we reduce function calls but keep the same minimum moves?** Mathematically, the number of *disk moves* is fixed at $2^n - 1$ for three pegs; that is optimal. In an actual program there is some extra cost from recursive calls and returns. We can reduce overhead a little (for example, by writing an iterative version that uses an explicit stack or by inlining small helper functions), but we cannot get to fewer than $2^n - 1$ disk moves without breaking correctness. So: minor constant-factor savings are possible, but the exponential growth in work is fundamental.
- **How does the algorithm pick the helper peg?** For three pegs, once we choose a source peg s and a target peg t, the helper peg h is simply "the other one." In code this often looks like: if the pegs are numbered `0,1,2` and we use all three, then $\{s, t, h\} = \{0, 1, 2\}$, so h is the unique peg different from s and t. The recursive calls just permute these roles.
- **Is the call tree a depth-first traversal of a perfect binary tree?** Yes. Each call `hanoi n x y z` makes two recursive calls on `n-1`, so the conceptual call tree is a full binary tree of height n. The program executes it in depth-first order (it finishes the left subtree, then does one move, then executes the right subtree). This is exactly what the call stack is doing under the hood.
- **Can we reorganize the recursion to "do less work" internally?** We can rearrange how the program explores the tree (left vs right first, or even unfold it into an iterative loop that simulates the tree), but the number of recursive steps that actually cause disk moves must still be $2^n - 1$. So we can change the *shape* of the control flow a bit and reduce constants, but not the overall exponential complexity.

Why recursion reconstructs the right answer.

- **What guarantees that unwinding gives the correct full solution?** Each recursive call solves a strictly smaller subproblem: move $n - 1$ disks to a helper peg. The recursive specification itself is:

$$\text{Hanoi}(n, x, y, z) = \text{Hanoi}(n-1, x, z, y);\ \text{move largest disk};\ \text{Hanoi}(n-1, z, y, x).$$

  If we assume by induction that the recursive calls move their $n-1$-disk towers correctly (without illegal moves and in the minimum number of steps), then:

  - After the first call, the top $n - 1$ disks are safely on the helper peg.
  - The middle move is legal (we move the largest disk onto an empty peg).
  - The final call stacks the $n - 1$ disks back on top correctly.

  Structural induction on n is what really guarantees the reconstruction is in the right order.

Recursion, iteration, and stack frames.

- **What does Hanoi show about stack frames and call depth?** The maximum recursion depth is n: we never have more than n active calls at once. Each level corresponds to "we are in the middle of moving a tower of size k." This mirrors how the call stack works in real languages: each active call has its own local variables and return address.
- **Risk of stack overflow.** In real programs, if n is huge, n levels of recursion might exceed the language's maximum stack size and cause a stack overflow. That is why for very large inputs, programmers often convert recursive algorithms to iterative ones that manage their own stack data structure.

- **Can every recursion be converted to iteration?** In standard imperative models, yes. Conceptually, you simulate the call stack yourself: store the arguments and "where to go next" in an explicit stack and use a loop to pop and push frames. This is the idea behind defunctionalization and how interpreters implement function calls internally. A proof sketch is by structural induction on the syntax of the recursive program.
- **Is the only difference between recursion and iteration that recursion is harder to write?** No. They are *expressively* similar for total computable functions, but:
  - Recursion often matches the shape of the data (trees, lists, towers of disks) and is easier to reason about.
  - Iteration gives more direct control over memory, stack usage, and performance.

  So they are more like two views of the same underlying control-flow patterns.

More pegs and generalized recurrences.

- **What if we add a fourth peg (Frame–Stewart)?** For four pegs, the best known strategy is the Frame–Stewart algorithm:
  1. Move k disks to an intermediate peg (using all four pegs).
  2. Move the remaining $n-k$ disks to the target using three pegs (ordinary Hanoi).
  3. Move the k disks from the intermediate peg to the target using all four pegs.

  This gives a recurrence like
  $$T_4(n) = \min_k \big(2T_4(k) + T_3(n-k)\big).$$
  It shows that recursive problem solving is very flexible: by choosing different subproblem sizes k, we can explore a whole family of strategies and then optimize over them.

Theory vs implementation and algorithmic "enshittification".

- **Do we usually worry about call-stack cost when teaching recursion?** In theory classes we mostly care about correctness and asymptotic behavior (like "$2^n - 1$ moves"). Implementation details like exact stack usage come later, in systems or programming-languages courses. In real projects, though, you *must* care about stack limits and constant factors.
- **What about "enshittification" and the rot economy in software?** The Towers of Hanoi itself is just a clean math puzzle, but it stands in contrast to "enshittified" software: here the recursive structure is minimal, transparent, and does no more work than necessary. In bloated real systems, layers of unnecessary abstraction, telemetry, and dark patterns add hidden "moves" that do not help the user. As new developers, the lesson is: keep the essential core small and understandable; do not add useless complexity that only serves short-term business goals.

Games, symmetry, and learning about runtime.

- **How does the move pattern relate to symmetry or recursion?** For three pegs, the smallest disk follows a simple periodic pattern of moves (e.g. $0 \to 2 \to 1 \to 0 \to 2 \to \cdots$ depending on the direction). The larger disks move less frequently, but in a symmetric, self-similar way. This repeating pattern is exactly what the recursive algorithm captures.
- **How can games like Hanoi inform programming and optimization?** Puzzles like this highlight:
  - The difference between *optimal work* (here $2^n - 1$ moves) and *implementation overhead*.
  - How a simple recursive description can generate a huge, structured computation.

– How call depth and recursion translate into memory and runtime costs.

That mental model transfers directly to evaluating algorithms in other domains.

14 Week 13: Experiments with the Lambda Interpreter

In this week I used the Python interpreter `interpreter.py` on a file `test.lc`. I added simple λ-terms from lecture and from Homework 5, tried to predict the normal forms, and then compared those predictions with what the interpreter produced. I also briefly looked *inside* the interpreter using the debugger.

14.1 Simple Tests in `test.lc` and Expected Results

I added the following expressions to `test.lc`:

```
a b c d
(a)
(\f.\x. f (f x))
(\f.\x. f (f x)) y
(\f.\x. f (f x)) (\f.\x. f (f (f x)))
```

Before running `python interpreter.py test.lc`, I wrote down what I expected:

- `a b c d` Application in the pure λ-calculus is left-associative, so

$$a\ b\ c\ d \quad \text{means} \quad (((a\ b)\ c)\ d).$$

  I expected the interpreter to print these parentheses explicitly or an equivalent tree.

- `(a)` Extra parentheses do not change anything in the λ-calculus, so I expected

$$(a) \to a.$$

- `(\f.\x. f (f x))` This is just a function value (a Church-style "2"), so it is already in normal form. I expected the interpreter to leave it unchanged.

- `(\f.\x. f (f x)) y` Here I expected one β-reduction:

$$(\lambda f.\lambda x.\ f(fx))\ y\ \to\ \lambda x.\ y(yx).$$

- **The Homework 5 term**

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

  Mathematically we know this is the Church numeral 9:

$$\lambda x.\lambda z.\ x^9 z.$$

  The interpreter may not simplify it all the way down to that pretty shape, but the important point is: if we later apply the result to a function s and an argument z, it will apply s nine times.

In all these simple tests the interpreter behaved as expected: parentheses were added or removed in the obvious way, and the β-steps matched the textbook rules.

14.2 Capture-Avoiding Substitution

Substitution is the key operation for β-reduction. In *capture-avoiding* substitution we must avoid accidentally turning a free variable into a bound one.

Idea. To substitute t for x in a term $\lambda y.\, u$:

- If $y = x$, we do *nothing* under this λ (the inner x is bound, not the one we are substituting for).
- If $y \neq x$ and y does not appear free in t, we can safely substitute inside u.
- If $y \neq x$ and y *does* appear free in t, we must first rename y to a fresh variable (say y') in u before we substitute. This is where capture is avoided.

How I saw it in the code. Looking at `interpreter.py` I observed (in simplified words):

- A function like `substitute(var, replacement, term)`.
- In the `Abs` (lambda) case it checks if the bound name clashes with the variable we are substituting, and if needed it generates a fresh variable name and renames the body before recursing.

Small tests. I used examples like

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

and checked that the result was still $\lambda y'.\ y$ (or some renamed version) and *not* the wrong term $\lambda y.\ y$ where y would have been captured. This confirmed that the substitution in the interpreter really is capture-avoiding.

14.3 Do All Computations Reduce? A Minimal Non-Terminating Example

Not all λ-terms reduce to normal form. The standard minimal example is

$$\Omega \ \equiv\ (\lambda x.\ x\,x)\,(\lambda x.\ x\,x).$$

This reduces as follows:

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

so it just reproduces itself forever.

When I put this into `test.lc`:

```
(\x. x x) (\x. x x)
```

and tried to evaluate it, the interpreter kept reducing (until I stopped it). So:

- For the earlier, simple examples I always got the expected result.
- In general, some terms (like Ω) do *not* reduce to normal form. This matches the theory from lecture.

14.4 Substitution Trace for the Church-9 Example

The interpreter represents variables internally with names like `Var1`, `Var2`, etc. Using the debugger and putting breakpoints around `substitute()`, I traced (in a simplified way) the evaluation of

$$((\lambda m.\lambda n.\ m\,n)\ (\lambda f.\lambda x.\ f(fx)))\ (\lambda f.\lambda x.\ f(f(fx))).$$

A simplified linear trace of *substitution steps* looks like this:

$$
\begin{aligned}
&((\lambda m.\lambda n.\ m\,n)\ (\lambda f.\lambda x.\ f(fx)))\ (\lambda f.\lambda x.\ f(f(fx))) \\
\to\ &((\lambda\, Var1.\ \lambda n.\ Var1\ n)\ (\lambda f.\lambda x.\ f(fx)))\ (\lambda f.\lambda x.\ f(f(fx))) \\
\to\ &(\lambda n.\ (\lambda f.\lambda x.\ f(fx))\ n)\ (\lambda f.\lambda x.\ f(f(fx))) \\
\to\ &(\lambda f.\lambda x.\ f(fx))\ (\lambda f.\lambda x.\ f(f(fx))) \\
\to\ &\lambda x.\ (\lambda f.\lambda x.\ f(f(fx)))\big((\lambda f.\lambda x.\ f(f(fx)))\ x\big).
\end{aligned}
$$

From here, repeated β-reduction eventually builds the Church numeral for 9 (as in Homework 5), although the interpreter keeps the explicit λ-structure rather than simplifying it to $x^9 z$.

14.5 Recursive Trace of `evaluate` and `substitute`

To understand the evaluation strategy, I also traced `evaluate()` calls for the simpler term

$$((\lambda m.\lambda n.\ m\,n)\ (\lambda f.\lambda x.\ f(fx)))\ (\lambda f.\lambda x.\ fx).$$

Using the debugger and watching the call stack, I got a tree-shaped trace like this (line numbers are illustrative and depend on the exact file):

```
12: eval ((((\m.\n. m n) (\f.\x. f (f x))) (\f.\x. f x))
    39: eval (((\m.\n. m n) (\f.\x. f (f x)))
        40: eval (\m.\n. m n)
        41: eval (\f.\x. f (f x))
        52: substitute(m := (\f.\x. f (f x))) in (\n. m n)
    39: eval ((\n. (\f.\x. f (f x)) n) (\f.\x. f x))
        41: eval (\f.\x. f x)
        52: substitute(n := (\f.\x. f x)) in ((\f.\x. f (f x)) n)
    39: eval ((\f.\x. f (f x)) (\f.\x. f x))
        41: eval (\f.\x. f x)
        52: substitute(f := (\f.\x. f x)) in (\x. f (f x))
```

Reading this:

- `evaluate` first looks at an application, evaluates the left-hand side, then the right-hand side.
- When the left-hand side becomes a λ-abstraction, it calls `substitute` to do one β-step.
- The indentation in the trace mirrors the recursive structure of the term, very much like the recursive trace of `hanoi`.

I did not need to write out every single recursive call in the report; the goal was only to see, at a high level, how `evaluate` and `substitute` alternate.

14.6 Notes & Discussion

The Discord questions for this week touched evaluation strategies, divergence, and practical debugging.

Normal order, call-by-value, and termination.

- **Why does normal order find a normal form if one exists?** Normal-order (leftmost-outermost) reduction always reduces the outermost redex first. A classic theorem says: if a λ-term has any normal form at all, then normal-order reduction will eventually reach it. Intuitively, you never waste time reducing inside subterms that will later be thrown away.

- **How do call-by-name vs call-by-value affect termination?** Call-by-value reduces arguments to values before passing them in (leftmost-innermost). This can get stuck looping on a diverging argument even when the function body would never use it. Normal order (or call-by-name) may terminate on the same term because it only evaluates arguments when needed.
- **Call-by-need (lazy).** Call-by-need is like call-by-name but memoizes results so each argument is evaluated at most once. This changes the *shape* of the recursive trace: fewer repeated evaluations of the same subterm.

α-conversion, fresh names, and de Bruijn indices.

- **What role does α-conversion play?** It ensures substitution is capture-avoiding: we rename bound variables so that free variables in the argument do not become accidentally bound. Without it, the interpreter could produce incorrect results when variable names clash.
- **Could an interpreter work without α-conversion?** If we somehow magically guaranteed that no program ever reused variable names in a conflicting way, then in practice we would not hit capture bugs. But that "guarantee" is fragile; using explicit α-conversion is the safe and standard approach.
- **Do fresh variable names affect the result?** No, as long as the names are genuinely fresh, any choice yields α-equivalent results. The printed terms may look slightly different, but they represent the same λ-expression up to renaming.
- **What about de Bruijn indices?** With de Bruijn indices, we drop names and use numbers to count how many binders away a variable is. Pros:
  - No need for α-renaming; binding is purely positional.
  - Substitution becomes more mechanical.

  Cons:
  - The terms are much less human-readable.
  - Off-by-one mistakes in indices are easy to make when writing or debugging.

  So named variables + α-conversion are nicer for teaching; de Bruijn is nicer for serious implementations.

Divergence, the halting problem, and evaluation strategies.

- **Why can some λ-terms reduce forever?** Because the calculus can express self-application and self-reference (like Ω and Y), it can encode non-terminating computations. A term diverges if the reduction sequence never reaches a normal form.
- **Can we detect divergence automatically?** In general, no. Any algorithm that always decides whether an arbitrary term will terminate would solve the halting problem, which is impossible. In practice, interpreters approximate divergence detection with step limits or timeouts: they give up after some number of reductions, even though a very slow but terminating program might still be running.
- **How do different strategies change the trace?** Call-by-value produces a trace that looks more like "real" languages (eager evaluation); call-by-name and call-by-need postpone some work and may change where and when substitutions happen. Compilers pick strategies based on performance, predictability, and interaction with side effects.

Formal semantics vs real implementations.

- **How does β-reduction relate to actual languages?** β-reduction is the core mathematical rule for function application. Real implementations of languages with functions (like Python, Haskell, or

OCaml) are more complicated—they have stacks, heaps, and optimizations—but they are designed so that, ignoring side effects, the observable behavior matches the formal semantics.

- **SM vs RM representations.** When we represent the same computation as a "stack machine" trace vs a more algebraic or "register machine" style, we are looking at the same recursive tree from two angles: one emphasizes control flow (what calls what, step by step), the other emphasizes the algebraic structure of transformations. Both are useful.

Debuggers, print statements, and VS Code.

- **VS Code vs "big" IDEs.** VS Code's debugger is surprisingly capable: breakpoints, stepping, watch expressions, call stacks, and even remote/container debugging via extensions. Full IDEs like IntelliJ or Visual Studio usually integrate more tightly with specific languages (refactorings, test runners, profilers).
- **How do debuggers work with remote or containerized code?** They attach to a debug server or a debug adapter running where the code executes. The IDE sends commands (step, continue) and receives state (break locations, variable values) over a protocol.
- **When are print statements better than the debugger?** For tiny scripts, quick one-off checks, or environments where attaching a debugger is hard (embedded systems, some production containers), sprinkling a few `print` lines is faster and less intrusive. They are also helpful when we need a persistent log.
- **Are there situations where we should not use a debugger?** Yes: e.g., when debugging timing-sensitive concurrency bugs (where pausing changes behavior), or in high-security production systems where attaching a debugger is forbidden. In those cases, logging, tracing, and property-based tests are safer.

Types, Church numerals, and Lark.

- **What if we add generics / types?** A typed λ-interpreter would need a type checker that runs before evaluation. Substitution and reduction rules stay conceptually the same but are restricted to well-typed terms. This mirrors real languages: type systems rule out some "bad" programs before they run.
- **Uses of Church numerals with λ-calculus.** Church numerals encode numbers as iterators: a numeral n represents "apply f exactly n times." They show up in:
  - proofs about expressiveness of λ-calculus,
  - encodings of arithmetic in proof assistants,
  - teaching how higher-order functions can stand in for loops.
- **Lark grammar outside syntax trees.** Lark (the Python parsing library) is not just for building syntax trees. It can also be used to:
  - build small domain-specific languages (config formats, mini query languages),
  - implement static analyzers and linters,
  - write transpilers or code formatters.

  The lambda interpreter is just one example of that pattern.

Intermediate results and program state.

- **How do intermediate and final results help us debug?** By inspecting intermediate terms (or the intermediate machine states for a more complicated interpreter), we can see exactly where behavior diverges from what we expect. This helps us localize bugs and understand gaps between the intended semantics and the implementation.
- **Doing better than "rot economy" in tools.** Good interpreters and debuggers are the opposite of "enshittified" software: they expose internals in a way that helps developers understand and fix code, instead of hiding or overcomplicating everything. The big lesson is to design tools that respect users' time and attention.

References

[BLA] Author, LaTeX Overview, Publisher, Year.