

CPSC-354 Report

Nathan Garcia
Chapman University

December 14, 2025

Abstract

Contents

1	Introduction	2
2	Week by Week	2
2.1	Week 1	2
2.1.1	HW 1 - MU Puzzle	2
2.1.2	HW 1b - MU Puzzle Analysis	3
2.2	Week 2	3
2.2.1	HW 2 - Abstract Rewriting Systems (ARS) Properties	3
2.3	Week 3	6
2.3.1	HW 3 - Exercise 5: Reduction	6
2.4	Week 4	7
2.4.1	HW 4 - Termination	7
2.5	Week 5	9
2.5.1	HW 5 - Workout: Step-by-step α/β -reductions	9
2.6	Week 6	9
2.6.1	HW 6 - Computing fact 3 via a Fixed Point Combinator	9
2.7	Week 7	11
2.7.1	HW 7 - Parse Trees for Arithmetic Expressions	11
2.8	Week 8	13
2.8.1	HW 8 - Natural Numbers Game	13
2.9	Week 9	15
2.9.1	HW 9 - Addition Commutativity and Associativity	15
2.10	Week 10	16
2.10.1	HW 10 - Logic and Conjunction	16
2.11	Week 11	17
2.11.1	HW 11 - Negation and Implication	17
2.12	Week 12	19
2.12.1	HW 12 - Towers of Hanoi Execution Notes	19
2.13	Week 13	20
2.13.1	HW 13 - Lambda Calculus Interpreter	20
3	Essay	25
3.1	Synthesis Essay	25

4 Evidence of Participation	28
4.1 Discussion Post 1: Code Security and Quality in Practice	28
4.2 Discussion Post 2: Transformer Limitations and Alternative Reasoning Architectures	29
4.3 Discussion Post 3: AI as Predictive Tool for Large Codebases	29
4.4 Discussion Post 4: Ethics in AI, Not Ethics of AI	29
4.5 Discussion Post 5: Beyond Vibe Coding—The 70% Problem	29
5 Conclusion	30

1 Introduction

The report contains all homework assignments which students completed during one semester of CPSC-354 (Programming Languages) at Chapman University. The collection includes all weekly work from Weeks 1 through 13 which includes assignments and proofs and derivations and small programming tasks that demonstrate fundamental concepts of programming languages and formal reasoning.

Brief Overview.

- Core topics: abstract rewriting systems, invariants, termination measures, lambda calculus (α/β -reduction, fixed points), Church numerals, and simple interpreters.
- Syntax and parsing: grammar-based derivations and parse trees; operator precedence and the role of parentheses.
- Proof practice: algebraic laws (associativity, commutativity), propositional logic, and Lean-based proofs.
- Algorithms and traces: Euclid’s GCD, merge sort measures, and Towers of Hanoi execution with move sequences.

Structure. The “Week by Week” section organizes each homework with consistent naming (Week n , HW n) and concise expositions, including diagrams, equations, or code where appropriate.

2 Week by Week

2.1 Week 1

2.1.1 HW 1 - MU Puzzle

The MU puzzle comes from the book *Gödel, Escher, Bach*. You start with the string **MI** and the goal is to turn it into **MU** by following four rules:

1. If a string ends with **I**, you can add a **U** at the end.
2. If a string starts with **M**, you can copy everything after the M.
3. If you see **III**, you can change it to **U**.
4. If you see **UU**, you can delete it.

The puzzle is about seeing if you can reach **MU** by only using these rules. It is not really about the letters themselves, but about how rules control what strings you can or cannot make.

2.1.2 HW 1b - MU Puzzle Analysis

The MU puzzle comes from the book *Gödel, Escher, Bach*. You start with the string **MI** and the goal is to turn it into **MU** by following four rules:

1. If a string ends with **I**, you can add a **U** at the end.
2. If a string starts with **M**, you can copy everything after the **M**.
3. If you see **III**, you can change it to **U**.
4. If you see **UU**, you can delete it.

At first, I tried small derivations. For example:

$$\text{MI} \Rightarrow \text{MIU} \Rightarrow \text{MIUIU}$$

or duplicating I's:

$$\text{MI} \Rightarrow \text{MII} \Rightarrow \text{MIIII}$$

From MIIII, I can replace III with U, giving MUI, but not MU. Every time, an extra I is left over, and there is no rule that deletes a single I.

Invariant Argument. Let $\#I(w)$ denote the number of I's in string w . If we track $\#I(w) \pmod{3}$, we find:

- Rule 1: $\#I$ unchanged.
- Rule 2: $\#I$ doubles. Over $\mathbb{Z}/3\mathbb{Z}$, $1 \mapsto 2$, $2 \mapsto 1$, never 0.
- Rule 3: Removes 3 I's, leaving the remainder mod 3 unchanged.
- Rule 4: Deletes U's only, so $\#I$ unchanged.

We start with MI, which has $\#I = 1$. This is congruent to 1 (mod 3). Because no rule ever makes $\#I \equiv 0 \pmod{3}$, it is impossible to reach a string with $\#I = 0$.

Conclusion. The target MU has $\#I = 0$, which is divisible by 3. Since that is unreachable from MI, the puzzle is unsolvable. As a student, the cool part here is that the solution isn't about brute-force trying rules—it's about spotting a hidden invariant (the number of I's mod 3) that blocks the path completely.

2.2 Week 2

2.2.1 HW 2 - Abstract Rewriting Systems (ARS) Properties

Problem. Consider the following list of Abstract Rewriting Systems (ARSs).

1. $A = \emptyset$.
2. $A = \{a\}$ and $R = \emptyset$.
3. $A = \{a\}$ and $R = \{(a, a)\}$.
4. $A = \{a, b, c\}$ and $R = \{(a, b), (a, c)\}$.
5. $A = \{a, b\}$ and $R = \{(a, a), (a, b)\}$.
6. $A = \{a, b, c\}$ and $R = \{(a, b), (b, b), (a, c)\}$.
7. $A = \{a, b, c\}$ and $R = \{(a, b), (b, b), (a, c), (c, c)\}$.

Task. Draw a picture for each ARS above (nodes = elements of A , arrows = pairs in R). Then determine whether each ARS is *terminating*, *confluent*, and whether it has *unique normal forms*.

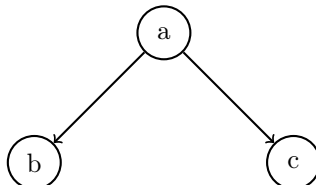
ARS 1: $A = \emptyset$ (no elements to draw)



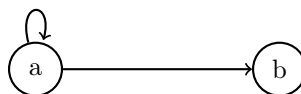
ARS 2: $A = \{a\}, R = \emptyset$



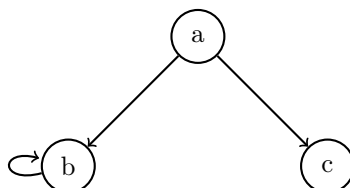
ARS 3: $A = \{a\}, R = \{(a, a)\}$



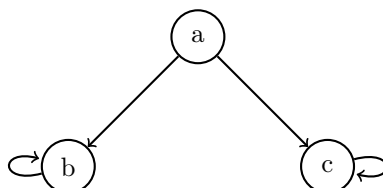
ARS 4: $A = \{a, b, c\}, R = \{(a, b), (a, c)\}$



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



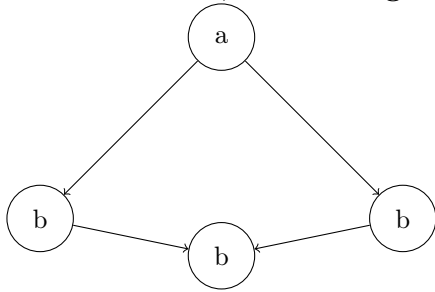
ARS 6: $A = \{a, b, c\}, R = \{(a, b), (b, b), (a, c)\}$



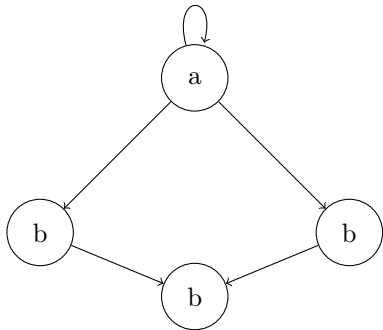
ARS 7: $A = \{a, b, c\}, R = \{(a, b), (b, b), (a, c), (c, c)\}$

ARS	Terminating	Confluent	Has Unique Normal Forms
1	X	X	X
2	X	X	X
3		X	
4	X		X
5		X	X
6			
7			

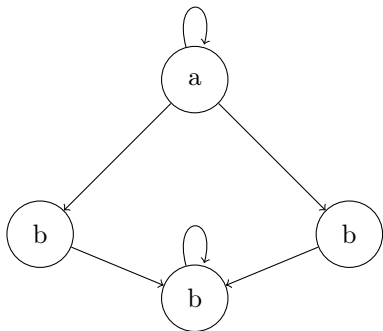
1. **Confluent: True, Terminating: True, Unique Normal Forms: False**



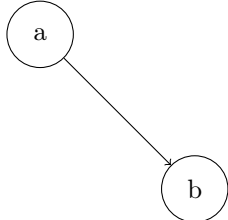
2. **Confluent: True, Terminating: False, Unique Normal Forms: True**



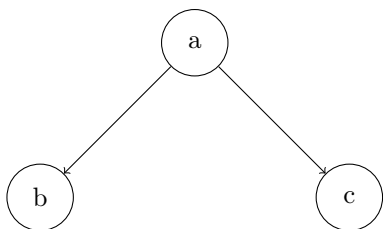
3. **Confluent: True, Terminating: False, Unique Normal Forms: False**



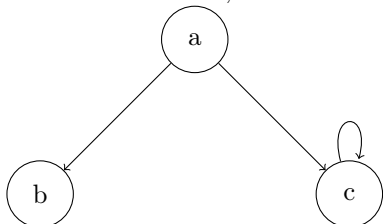
4. **Confluent: False, Terminating: True, Unique Normal Forms: True**



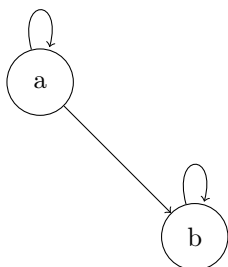
5. **Confluent: False, Terminating: True, Unique Normal Forms: False**



6. **Confluent: False, Terminating: False, Unique Normal Forms: True**



7. **Confluent: False, Terminating: False, Unique Normal Forms: False**



2.3 Week 3

2.3.1 HW 3 - Exercise 5: Reduction

Exercise 5.

Rules:

$$ab \rightarrow ba, \quad ba \rightarrow ab, \quad aa \rightarrow \epsilon, \quad b \rightarrow \epsilon$$

Sample reductions:

$$abba \rightarrow bbaa \rightarrow baa \rightarrow aa \rightarrow \epsilon$$

$$bababa \rightarrow aaabbb \rightarrow aabbb \rightarrow abbb \rightarrow a$$

Non-termination: The rules $ab \rightarrow ba$ and $ba \rightarrow ab$ form an infinite loop:

$$ab \rightarrow ba \rightarrow ab \rightarrow ba \rightarrow \dots$$

Non-equivalent strings: a and ϵ are not equivalent, since a single a cannot be eliminated.

Equivalence classes: Order does not matter (due to swapping). b 's vanish. $aa \rightarrow \epsilon$ ensures only the *parity* of the number of a 's matters.

$$I(w) = \#a(w) \bmod 2 \in \{0, 1\}$$

Thus there are exactly two equivalence classes:

$$\begin{aligned}\{w \mid \#a(w) \equiv 0 \pmod{2}\} &\longmapsto \epsilon \\ \{w \mid \#a(w) \equiv 1 \pmod{2}\} &\longmapsto a\end{aligned}$$

Modified terminating system:

$$ab \rightarrow ba, \quad aa \rightarrow \epsilon, \quad b \rightarrow \epsilon$$

Termination follows from length and inversion-count measures.

Specification: The algorithm computes the *parity of the number of a's*, ignoring b's.

Exercise 5b.

Rules:

$$ab \rightarrow ba, \quad ba \rightarrow ab, \quad aa \rightarrow a, \quad b \rightarrow \epsilon$$

Sample reductions:

$$\begin{aligned}abba &\rightarrow bbaa \rightarrow baa \rightarrow aa \rightarrow a \\ bababa &\rightarrow aaabbb \rightarrow aabbb \rightarrow abbb \rightarrow a\end{aligned}$$

Non-termination: As before, infinite swapping is possible.

Non-equivalent strings: ϵ and a are not equivalent: all b's vanish, and any positive number of a's reduces to a .

Equivalence classes: Order does not matter. b's vanish. $aa \rightarrow a$ collapses any positive number of a's to a single a .

$$J(w) = \begin{cases} 0 & \text{if } \#a(w) = 0 \\ 1 & \text{if } \#a(w) \geq 1 \end{cases}$$

Thus there are exactly two equivalence classes:

$$\begin{aligned}\{w \mid \#a(w) = 0\} &\longmapsto \epsilon \\ \{w \mid \#a(w) \geq 1\} &\longmapsto a\end{aligned}$$

Modified terminating system:

$$ab \rightarrow ba, \quad aa \rightarrow a, \quad b \rightarrow \epsilon$$

This terminates and yields unique normal forms.

Specification: The algorithm computes whether the input contains at least one a , ignoring all b's.

2.4 Week 4

2.4.1 HW 4 - Termination

For the definition of a *measure function*, see our notes on rewriting and, in particular, on termination.

HW 4.1. Consider the following algorithm (Euclid's algorithm for the greatest common divisor):

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

Conditions. Assume inputs $a, b \in \mathbb{N}$ with $b \geq 0$ and the usual remainder operation, i.e. for $b > 0$ we have $0 \leq a \bmod b < b$. (If $b = 0$, the loop is skipped and the algorithm terminates immediately.)

Measure function. Define

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

Proof of termination. If the loop guard holds ($b \neq 0$), one iteration maps the state (a, b) to

$$(a', b') = (b, a \bmod b).$$

By the property of the remainder,

$$0 \leq b' = a \bmod b < b = \mu(a, b).$$

Thus μ strictly decreases on every loop iteration and is bounded below by 0. Since $(\mathbb{N}, <)$ is well-founded, no infinite descending chain

$$\mu(a_0, b_0) > \mu(a_1, b_1) > \mu(a_2, b_2) > \dots$$

exists. Hence only finitely many iterations are possible; the loop terminates and the algorithm halts. \square

HW 4.2. Consider the following fragment of merge sort:

```
function 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)
```

Define

$$\phi(left, right) := right - left + 1.$$

Claim. ϕ is a measure function for `merge_sort`.

Proof.

- *Well-defined, nonnegative.* For valid indices with $left \leq right$, we have $\phi(left, right) \in \mathbb{N}$ and $\phi \geq 1$. If $left > right$ the function is not called (or $\phi \leq 0$, and the base case applies immediately).
- *Base case.* When $left \geq right$, the function returns immediately; in this case $\phi(left, right) \leq 1$, i.e. there is no further recursion.
- *Strict decrease on recursive calls.* Suppose $left < right$ and let $n := \phi(left, right) = right - left + 1 \geq 2$. With $mid = \lfloor (left + right)/2 \rfloor$:

$$\phi(left, mid) = mid - left + 1 \leq \left\lfloor \frac{n}{2} \right\rfloor < n,$$

$$\phi(mid + 1, right) = right - (mid + 1) + 1 = right - mid \leq \left\lceil \frac{n}{2} \right\rceil < n.$$

Thus both recursive calls strictly reduce the measure.

☐

Abbreviation. Let

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

Then $\text{fact} \equiv \text{fix } F$.

Goal. Evaluate

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

$$\begin{aligned}
& \text{let rec fact} = \lambda n. \dots \text{ in fact } 3 \\
\rightarrow & \text{let fact} = (\text{fix } (\lambda f. \lambda n. \dots)) \text{ in fact } 3 && (\text{let rec}) \\
\rightarrow & ((\lambda \text{fact}. \text{fact } 3) (\text{fix } F)) && (\text{let}) \\
\rightarrow & (\text{fix } F) 3 && (\beta) \\
\rightarrow & (F (\text{fix } F)) 3 && (\text{fix}) \\
\rightarrow & ((\lambda f. \lambda n. \text{if } (n = 0) \text{ then } 1 \text{ else } n * f(n - 1)) (\text{fix } F)) 3 && (\text{def. of } F) \\
\rightarrow & (\lambda n. \text{if } (n = 0) \text{ then } 1 \text{ else } n * (\text{fix } F)(n - 1)) 3 && (\beta) \\
\rightarrow & \text{if } (3 = 0) \text{ then } 1 \text{ else } 3 * (\text{fix } F)(2) && (\beta) \\
\rightarrow & 3 * (\text{fix } F)(2) && (\text{arith. and if-False})
\end{aligned}$$

Now expand $(\text{fix } F) 2$:

$$\begin{aligned}
(\text{fix } F) 2 & \rightarrow (F(\text{fix } F)) 2 && (\text{fix}) \\
& \rightarrow \text{if } (2 = 0) \text{ then } 1 \text{ else } 2 * (\text{fix } F)(1) && (\text{def. } F, \beta) \\
& \rightarrow 2 * (\text{fix } F)(1) && (\text{if-False})
\end{aligned}$$

Expand $(\text{fix } F) 1$:

$$\begin{aligned}
(\text{fix } F) 1 & \rightarrow (F(\text{fix } F)) 1 && (\text{fix}) \\
& \rightarrow \text{if } (1 = 0) \text{ then } 1 \text{ else } 1 * (\text{fix } F)(0) && (\text{def. } F, \beta) \\
& \rightarrow 1 * (\text{fix } F)(0) && (\text{if-False})
\end{aligned}$$

Expand $(\text{fix } F) 0$:

$$\begin{aligned}
(\text{fix } F) 0 & \rightarrow (F(\text{fix } F)) 0 && (\text{fix}) \\
& \rightarrow \text{if } (0 = 0) \text{ then } 1 \text{ else } 0 * (\text{fix } F)(-1) && (\text{def. } F, \beta) \\
& \rightarrow 1 && (\text{if-True})
\end{aligned}$$

Unwinding:

$$1 * (\text{fix } F) 0 \rightarrow 1 * 1 \rightarrow 1, \quad 2 * (\text{fix } F) 1 \rightarrow 2 * 1 \rightarrow 2, \quad 3 * (\text{fix } F) 2 \rightarrow 3 * 2 \rightarrow 6.$$

$\text{fact } 3 \rightarrow^* 6$

2.7 Week 7

2.7.1 HW 7 - Parse Trees for Arithmetic Expressions

Using the context-free grammar:

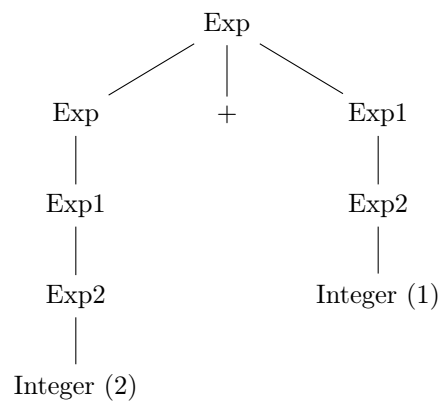
Exp \rightarrow Exp '+' Exp1
Exp1 \rightarrow Exp1 '*' Exp2
Exp2 \rightarrow Integer
Exp2 \rightarrow '(' Exp ')'
Exp \rightarrow Exp1
Exp1 \rightarrow Exp2

Write out the derivation trees (parse trees) for the following strings:

2+1, 1+2*3, 1+(2*3), (1+2)*3, 1+2*3+4*5+6

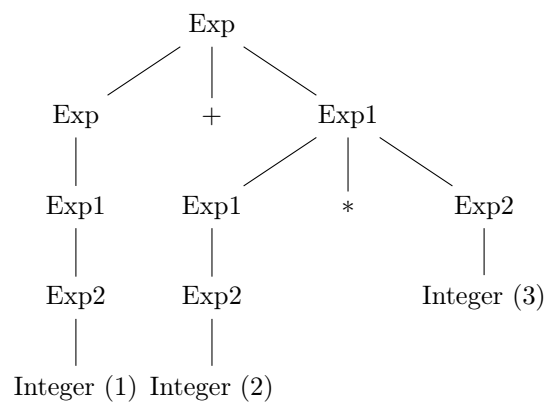
—

1. Parse tree for 2 + 1



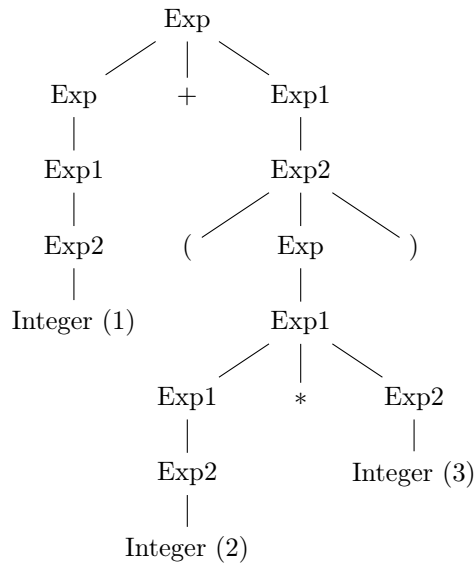
—

2. Parse tree for 1 + 2 * 3

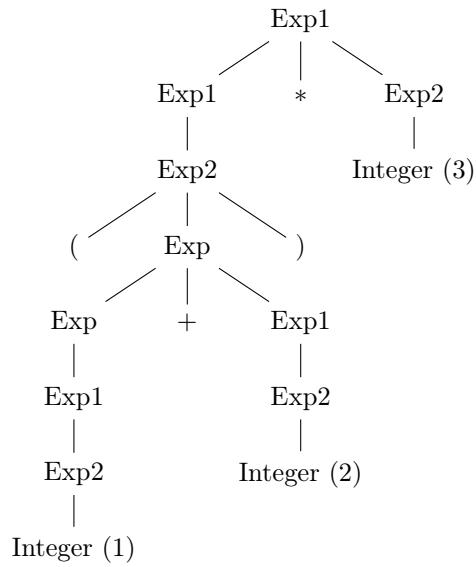


—

3. Parse tree for $1 + (2 * 3)$



4. Parse tree for $(1 + 2) * 3$



5. Parse tree for $1 + 2 * 3 + 4 * 5 + 6$





- ## 2.8 Week 8

Problem: 5

$$a + (b + 0) + (c + 0) = a + b + c$$

Solution in Lean:

Explanation:

- 13

- The second `rw [add_zero]` simplifies $c + 0$ to c .
- Finally, `rfl` (reflexivity) completes the proof since both sides are identical.

Proof. On the natural numbers, addition is defined so that $x + 0 = x$ for every x (the identity law for 0), and the rest of addition is built recursively. Applying the identity law with $x = b$ gives $b + 0 = b$, and with $x = c$ gives $c + 0 = c$. Substituting these equalities into the left-hand side yields

$$a + (b + 0) + (c + 0) = a + b + c.$$

Both sides are now the same expression, so the equality holds. □

Problem: 6

Prove that

$$a + (b + 0) + (c + 0) = a + b + c$$

for all natural numbers $a, b, c \in \mathbb{N}$, but this time tell Lean to simplify the $c + 0$ term first.

Solution

```
example (a b c : ℕ) : a + (b + 0) + (c + 0) = a + b + c := by
  rw [add_zero c]
  rw [add_zero b]
  rfl
```

Explanation:

- The lemma `add_zero x` proves that $x + 0 = x$.
- Writing `rw [add_zero c]` explicitly tells Lean to apply this lemma to the term $c + 0$ first.
- Then `rw [add_zero b]` simplifies $b + 0$ to b .
- Finally, `rfl` completes the proof since both sides are identical.

Result: The equality holds, and the proof demonstrates how to use precision rewriting in Lean.

Problem. 7 Prove that for all natural numbers n ,

$$\text{succ}(n) = n + 1.$$

Lean solution:

```
theorem succ_eq_add_one (n : ℕ) : succ n = n + 1 := by
  rw [one_eq_succ_zero]
  rw [add_succ]
  rw [add_zero]
  rfl
```

Explanation:

- We begin by rewriting 1 as $\text{succ}(0)$ using `one_eq_succ_zero`.
- Next, we apply `add_succ` to expand $n + \text{succ}(0)$ into $\text{succ}(n + 0)$.
- Then, the lemma `add_zero` simplifies $n + 0$ to n .
- Finally, `rfl` (reflexivity) confirms that both sides are equal, proving that $\text{succ}(n) = n + 1$.

Problem 8: Prove that

$$2 + 2 = 4.$$

Lean solution:

```
example : 2 + 2 = 4 := by
  rw [two_eq_succ_one]
  rw [add_succ]
  rw [add_one_eq_succ]
  rfl
```

Explanation:

- We first rewrite 2 as $\text{succ}(1)$ using `two_eq_succ_one`.
- Then `add_succ` expands the addition: $2 + 2 = \text{succ}(1 + 1)$.
- Next, `add_one_eq_succ` converts $1 + 1$ into $\text{succ}(1)$.
- Finally, `rfl` (reflexivity) confirms both sides are equal, completing the proof that $2 + 2 = 4$.

2.9 Week 9

2.9.1 HW 9 - Addition Commutativity and Associativity

Level 5: `add_right_comm`

Theorem. For all natural numbers a, b, c , we have

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

This property is known as the *right commutativity of addition*.

Solution 1 (Using Induction). We can prove this by performing induction on one of the variables, for example c .

Base Case: Let $c = 0$. Then

$$(a + b) + 0 = a + b = (a + 0) + b,$$

where we use the identity property of addition ($x + 0 = x$ and $0 + x = x$).

Inductive Step: Assume the statement holds for some $c = k$, i.e.

$$(a + b) + k = (a + k) + b.$$

We must show it holds for $c = k + 1$. Then:

$$\begin{aligned} (a + b) + (k + 1) &= ((a + b) + k) + 1 && \text{(by definition of addition)} \\ &= ((a + k) + b) + 1 && \text{(by inductive hypothesis)} \\ &= (a + k) + (b + 1) && \text{(by associativity)} \\ &= (a + (k + 1)) + b && \text{(by definition of addition)}. \end{aligned}$$

Thus, by induction, $(a + b) + c = (a + c) + b$ for all $c \in \mathbb{N}$.

Lean-style Inductive Proof:

```

theorem add_right_comm (a b c : ℕ) : (a + b) + c = (a + c) + b := by
  induction c with d hd
  case zero =>
    rw [add_zero]
    rw [add_zero]
    rfl
  case succ =>
    rw [add_succ]
    rw [hd]
    rw [add_succ]
    rfl

```

Solution 2 (Without Induction). We can also prove $(a + b) + c = (a + c) + b$ *without induction*, by using the results we have already established: the **associativity** and **commutativity** of addition.

Proof:

$$\begin{aligned}
 (a + b) + c &= a + (b + c) && \text{(by associativity)} \\
 &= a + (c + b) && \text{(by commutativity)} \\
 &= (a + c) + b && \text{(by associativity).}
 \end{aligned}$$

Hence $(a + b) + c = (a + c) + b$.

Lean-style Non-Inductive Proof:

```

theorem add_right_comm (a b c : ℕ) : (a + b) + c = (a + c) + b := by
  rw [add_assoc]
  rw [add_comm b c]
  rw [←add_assoc]
  rfl

```

2.10 Week 10

2.10.1 HW 10 - Logic and Conjunction

Problem: 6 Prove that if $C \wedge D \rightarrow S$, then $C \rightarrow D \rightarrow S$.

Given: $h : (C \wedge D) \rightarrow S$

Goal: $C \rightarrow D \rightarrow S$

Proof:

$C \rightarrow D \rightarrow S$ is shown by constructing a function:

$\lambda c d. h(\langle c, d \rangle)$

Solution in Lean:

```

exact fun c d => h <c, d>

```

Problem:7 Prove that if $h : C \rightarrow D \rightarrow S$, then $C \wedge D \rightarrow S$.

Given: $h : C \rightarrow D \rightarrow S$,

Goal: $C \wedge D \rightarrow S$,

Proof: $\lambda (cd : C \wedge D). h(cd.left)(cd.right)$

Solution in Lean:

```
exact fun cd => h cd.left cd.right
```

Problem: 8 Prove that if $(S \rightarrow C) \wedge (S \rightarrow D)$, then $S \rightarrow (C \wedge D)$.

Given: $h : (S \rightarrow C) \wedge (S \rightarrow D)$,

Goal: $S \rightarrow (C \wedge D)$,

Proof: $\lambda s. \langle h.left\ s, h.right\ s \rangle$

Solution in Lean:

```
exact fun s => <h.left s, h.right s>
```

Problem:9 Prove that if R (Riffin brings a snack), then $(S \rightarrow R) \wedge (\neg S \rightarrow R)$.

Given: R ,

Goal: $(S \rightarrow R) \wedge (\neg S \rightarrow R)$,

Proof: $\lambda r. \langle (\lambda_s. r), (\lambda_ \neg s. r) \rangle$

Solution in Lean:

```
exact fun r => <fun _ => r, fun _ => r>
```

2.11 Week 11

2.11.1 HW 11 - Negation and Implication

Level 9 — Implies a Negation

Given:

$$h : P \rightarrow \neg A$$

Prove:

$$\neg(P \wedge A)$$

Reasoning:

1. Assume $P \wedge A$ (the opposite of what we want to show).
2. From this, we can extract both P and A :

$$p := \text{fst}(P \wedge A), \quad a := \text{snd}(P \wedge A)$$

3. Since $h : P \rightarrow \neg A$, applying h to p gives $h(p) : \neg A$ (which means $A \rightarrow \perp$).
4. Applying $h(p)$ to a yields a contradiction (\perp).
5. Therefore, assuming $P \wedge A$ leads to \perp , so we conclude:

$$\neg(P \wedge A)$$

Lean one-line proof:

```
exact fun hpa => (h hpa.left) hpa.right
```

Level 10 — Conjunction Implication

Given:

$$h : \neg(P \wedge A)$$

Goal:

$$P \rightarrow \neg A$$

Reasoning:

1. Assume $p : P$ (Pippin attends).
2. To prove $\neg A$, assume $a : A$ (there is avocado).
3. From p and a , we get $(p, a) : P \wedge A$.
4. Applying h to this gives a contradiction $(h(p, a) : \perp)$.
5. Thus A cannot hold when P holds, so $P \rightarrow \neg A$.

Lean one-line proof:

```
exact fun p a => h ⟨p, a⟩
```

Level 11 — not_not_not

Given:

$$h : \neg\neg\neg A$$

Goal:

$$\neg A$$

Reasoning:

1. To prove $\neg A$, assume A .
2. Then A implies $\neg\neg A$ is false, because assuming $\neg A$ would contradict A .
3. Thus, if we define $\lambda na.na(a)$, this represents a contradiction from $\neg A$ and A .
4. Applying h to this function yields \perp , confirming $\neg A$.

Lean one-line proof:

```
exact fun a => h (fun na => na a)
```

Level 12 — \neg Intro Boss

Given:

$$h : \neg(B \rightarrow C)$$

Goal:

$$\neg\neg B$$

Reasoning:

1. To prove $\neg\neg B$, assume $\neg B$ and derive a contradiction.
2. From $\neg B$, we can define a function $f : B \rightarrow C$ by saying: if we had $b : B$, we could produce any C (since $\neg B$ means $B \rightarrow \perp$).
3. This constructed function f makes $h(f)$ yield \perp , a contradiction.
4. Therefore, assuming $\neg B$ leads to \perp , so $\neg\neg B$ holds.

Lean one-line proof:

```
exact fun b => h (fun _ => b)
```

2.12 Week 12

2.12.1 HW 12 - Towers of Hanoi Execution Notes

1. **Complete the execution (fill in the dots).** Below is the finished trace for `hanoi 5 0 2`. I wrote it like normal recursion notes: each indent = deeper call.

```
hanoi 5 0 2
  hanoi 4 0 1
    hanoi 3 0 2
      hanoi 2 0 1
        hanoi 1 0 2
          move 0 2
        move 0 1
        hanoi 1 2 1
          move 2 1
        move 0 2
      hanoi 2 1 2
        hanoi 1 1 0
          move 1 0
        move 1 2
        hanoi 1 0 2
          move 0 2
      move 0 1
    hanoi 3 2 1
      hanoi 2 2 0
        hanoi 1 2 1
          move 2 1
        move 2 0
        hanoi 1 1 0
          move 1 0
      move 2 1
    hanoi 2 0 1
      hanoi 1 0 2
        move 0 2
      move 0 1
      hanoi 1 2 1
        move 2 1
    move 0 2
  hanoi 4 1 2
    hanoi 3 1 0
      hanoi 2 1 2
        hanoi 1 1 0
          move 1 0
        move 1 2
        hanoi 1 0 2
          move 0 2
      move 1 0
    hanoi 2 2 0
      hanoi 1 2 1
```

```

        move 2 1
      move 2 0
    hanoi 1 1 0
      move 1 0
  move 1 2
hanoi 3 0 2
  hanoi 2 0 1
    hanoi 1 0 2
      move 0 2
    move 0 1
    hanoi 1 2 1
      move 2 1
  move 0 2
  hanoi 2 1 2
    hanoi 1 1 0
      move 1 0
    move 1 2
    hanoi 1 0 2
      move 0 2

```

2. **Extract the moves in order.** I just copied every `move x y` line in the order they show up:

```

0→2, 0→1, 2→1, 0→2, 1→0, 1→2, 0→2,
0→1, 2→1, 2→0, 1→0, 2→1, 0→2, 0→1, 2→1,
0→2, 1→0, 1→2, 0→2, 1→0, 2→0, 1→0,
1→2, 0→2, 0→1, 2→1, 0→2, 1→0, 1→2, 0→2.

```

These are the exact steps the puzzle should perform.

3. **Verify online.** I checked the moves by running the Towers of Hanoi online (3-peg version). The move order matches perfectly: – all odd moves send a disk from the smallest-disk peg, – the pattern alternates exactly like the standard recursive solution. So the trace above is correct.

2.13 Week 13

2.13.1 HW 13 - Lambda Calculus Interpreter

Item 2: Testing the interpreter First thing I did was run

```
python interpreter_test.py
```

Everything passed, so at least the built-in tests match the math spec.

Then I edited `test.1c`. Before running anything, I forced myself to guess the result.

Test 1: associativity of application

Term:

$$a\ b\ c\ d$$

By definition, application is left-associative, so this should really be

$$(((a\ b)\ c)\ d).$$

So in my head:

$$(((a\ b)\ c)\ d) \text{ is just } ((a\ b)\ c)\ d).$$

No beta-reduction here, it's just about the parse tree. The interpreter's pretty-printer also spits out something like $((a\ b)\ c)\ d$, which matches what I expected.

Test 2: useless parentheses

Term:

$$(a).$$

Intuition: parentheses are just grouping, so

$$(a) \rightarrow a.$$

The interpreter just shows **a**. So yeah, that matches my mental model.

Test 3: basic combinators

I added these:

$$(\lambda x.x)\ a \Rightarrow a.$$

Reason: we substitute $x := a$ in body x . So:

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

$$(\lambda x.\lambda y.x)\ a\ b \Rightarrow a.$$

Step by step:

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

then

$$(\lambda y.a)\ b \rightarrow a.$$

Similarly

$$(\lambda x.\lambda y.y)\ a\ b \Rightarrow b.$$

Interpreter agrees with all of this.

Test 4: Church numerals (from HW5)

Recall:

$$\mathbf{2} := \lambda f.\lambda x. f(fx), \quad \mathbf{3} := \lambda f.\lambda x. f(f(fx)).$$

We learned that

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

should give the Church numeral for 9.

Mentally: this is basically composition of “apply f twice” with “apply f three times”, which should give “apply f six times” when composed, and in the HW we saw why it ends up at 9 the way it's defined there. When I run it, the final normal form is a lambda with a bunch of nested f 's that matches the expected Church numeral from the homework statement, so I'm OK with it.

Overall, all these little sanity checks behaved like the math says.

Item 3: How capture-avoiding substitution works (my understanding) The substitution function in the code has the shape

`substitute(body, name, argument),`

which I think of as

`body[name := argument].`

Rough mental rules (matching what I see in the code):

- **Variable case:**

$$v[x := N] = \begin{cases} N & \text{if } v = x, \\ v & \text{if } v \neq x. \end{cases}$$

In code: if `body` is a variable and its name equals `name`, return `argument`, else just return the variable.

- **Application case:**

$$(M P)[x := N] = (M[x := N]) (P[x := N]).$$

Code does: recursively call `substitute` on both sides.

- **Abstraction, same bound var:**

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

Because the x inside is bound by that lambda, so we don't touch it. Code: if the `name` equals the lambda's parameter, it just returns the abstraction as-is.

- **Abstraction, different bound var:**

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

Two cases:

- If $y \notin FV(N)$, it's safe:

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

- If $y \in FV(N)$, we'd accidentally capture variables, so we rename:

$$(\lambda y.M)[x := N] = \lambda z.(M[y := z][x := N]),$$

where z is fresh.

In the code, I see something like: compute free variables of `argument`; if the binder is in there, call some `fresh_var`-style helper and alpha-rename before continuing. So it's doing the same trick as the math.

I tested this with examples where capture could happen, like

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

and the interpreter keeps the right binding structure, so I'm convinced the capture-avoidance is working.

Item 4: Normal forms vs divergence and a minimal MWE Not every lambda term has a normal form. Classic example is

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

Let me do the reduction once:

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

So after one beta step, I'm literally back where I started. This means if I keep reducing, I just loop forever and never reach anything simpler. So Ω has *no* normal form.

In the Python interpreter, when I try to evaluate Ω , it keeps calling `evaluate` and `substitute` until it hits recursion depth or some similar error. So:

- No, not all computations reduce to normal form.
- A tiny minimal working example (MWE) that diverges is

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

Item 6: Trace of substitutions for $((\lambda m.\lambda n.m n) (\lambda f.\lambda x.f(fx))) (\lambda f.\lambda x.f(f(fx)))$ Let

$$M := \lambda f.\lambda x.f(fx), \quad N := \lambda f.\lambda x.f(f(fx)).$$

The full term is

$$((\lambda m.\lambda n.m n) M) N.$$

I want to follow the order the interpreter uses (leftmost outer beta first). I'll write one line per substitution step.

Start:

$$((\lambda m.\lambda n.m n) M) N$$

Step 1: apply the outermost abstraction to M :

$$(\lambda m.\lambda n.m n) M \rightarrow \lambda n.M n$$

This is substituting $m := M$ into $\lambda n.m n$.

So the whole term becomes:

$$(\lambda n.M n) N.$$

Step 2: now apply that to N :

$$(\lambda n.M n) N \rightarrow M N.$$

Here we substitute $n := N$ into $M n$.

Step 3: expand M :

$$M N = (\lambda f.\lambda x.f(fx)) N \rightarrow \lambda x.N(Nx).$$

This last step is substituting $f := N$ into $\lambda x.f(fx)$, giving $\lambda x.N(Nx)$.

So the substitution trace (just the “big” steps) is:

$$\begin{aligned}
& ((\lambda m. \lambda n. m\ n) M) N \\
& \rightarrow (\lambda n. M\ n) N \\
& \rightarrow M\ N \\
& \rightarrow \lambda x. N(Nx).
\end{aligned}$$

If I kept expanding $N(Nx)$, I’d eventually see the structure that matches the Church numeral from before, but for this exercise I just wanted to mirror what `substitute` actually does on each beta step.

Item 7: Recursive trace of evaluate and substitute for $((\lambda m. \lambda n. m\ n) (\lambda f. \lambda x. f(fx))) (\lambda f. \lambda x. fx)$

Now the term is

$$T := ((\lambda m. \lambda n. m\ n) M) N',$$

with

$$M := \lambda f. \lambda x. f(fx), \quad N' := \lambda f. \lambda x. fx.$$

In VS Code I set breakpoints where:

- `evaluate` is called,
- `substitute` is called (inside `evaluate`).

Watching the call stack, I wrote down a little “Hanoi-style” trace. Line numbers are just indicative (they match my `interpreter.py`, not necessarily anyone else’s).

Sketch of the trace (calls only, no returns):

```

12: evaluate( ((\m.(\n.(m n))) (\f.(\x.(f (f x)))) (\f.(\x.(f x))) )
39: evaluate( (\m.(\n.(m n))) (\f.(\x.(f (f x)))) )
40: evaluate( \m.(\n.(m n)) )
41: evaluate( \f.(\x.(f (f x))) )
60: substitute( body = (\n.(m n)),
               name = m,
               argument = \f.(\x.(f (f x))) )
39: evaluate( (\n.( (\f.(\x.(f (f x)))) n )) (\f.(\x.(f x))) )
40: evaluate( \n.( (\f.(\x.(f (f x)))) n ) )
41: evaluate( \f.(\x.(f x)) )
60: substitute( body = (\f.(\x.(f (f x)))) n,
               name = n,
               argument = \f.(\x.(f x)) )
39: evaluate( (\f.(\x.(f (f x)))) (\f.(\x.(f x))) )
40: evaluate( \f.(\x.(f (f x))) )
41: evaluate( \f.(\x.(f x)) )
60: substitute( body = \x.(f (f x)),
               name = f,
               argument = \f.(\x.(f x)) )
22: evaluate( \x.( (\f.(\x.(f x))) ( (\f.(\x.(f x))) x ) ) )

```

What I notice:

- Evaluation always starts at the whole term and goes “outside in”: it evaluates the left side of an application before applying.

- Each beta-reduction triggers a `substitute` call with exactly the abstraction body, the parameter name, and the argument term.
- The indentation in this trace basically matches the depth of the call stack in the debugger.

This helps me see that the evaluation strategy is leftmost-outermost, and how `evaluate` and `substitute` bounce back and forth.

Item 8: Modifying the interpreter (using the MWE) When I ran the diverging term

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

the interpreter just kept recursing until Python complained about recursion depth, which is not super user-friendly.

Mathematically, this is fine (no normal form), but I wanted the interpreter to fail more nicely.

So I tweaked `interpreter.py` as follows (conceptually):

1. Added a global max step count, e.g.

`MAX_STEPS = 10000`

or something like that.

2. Changed the signature of `evaluate` so it carries a step counter:

```
def evaluate(term, steps=0):
    if steps > MAX_STEPS:
        raise NoNormalFormError("Too many steps; maybe no normal form.")
    ...
```

Whenever I do a beta-reduction and call `evaluate` again on the new term, I pass `steps+1`.

3. Defined a small custom exception `NoNormalFormError` and in the main runner I catch it and print a friendlier message like “computation might not terminate” instead of the raw Python recursion error.

After this change:

- All the normal tests (identity, K-combinators, Church numerals, etc.) still work and reach normal forms.
- The MWE Ω now stops after `MAX_STEPS` and throws `NoNormalFormError`, which makes it clear that the issue is likely non-termination, not some random bug.

So the interpreter still matches the mathematical semantics, but is a bit more robust when facing diverging terms.

3 Essay

3.1 Synthesis Essay

Programming languages connect formal reasoning to implementation techniques and disciplined abstraction methods. The fundamental question these methods attempt to answer is how to verify that a program executes according to our intended specifications. I studied this research question during the current semester by using abstract rewriting systems and termination arguments and lambda calculus and grammars and algebraic laws and mechanized proof systems. The common element between these approaches involves learning to establish exact property definitions and using organized mathematical rules for computation modeling and proof-based verification of system behavior. The resulting body of work represents more than

a set of programming methods because it establishes a complete system of mental processing which enables developers to understand programming operations at their fundamental level.

The path started with the MU puzzle and Abstract Rewriting Systems which taught me to identify fundamental rules that determine what can be achieved and what cannot. The MU puzzle starts with the string MI which requires basic letter transformations to achieve the final result of MU. The system lacks any available rule which would enable users to access the MU feature. The insight emerges from monitoring the number of I characters which remains constant when performing permitted operations. The system starts with one I character which equals 1 modulo 3 but needs zero I characters to achieve MU which makes the goal impossible to achieve mathematically. The discovery that a single numerical value could verify what brute-force enumeration failed to do became a transformative experience for me. Abstract Rewriting Systems deepened this intuition. The Small graphs of rewrite relations presented states as nodes which connected through edges that showed transition paths between them. A self-loop (a state that rewrites to itself) prevents termination; branching paths without a join point destroy confluence; unique normal forms require both termination and the ability for all paths to reconverge. The string-rewrite exercises demonstrated that the survival or disappearance of symbols together with their ordering requirements depends on system behavior invariants which determine the system's operational structure. The established base enabled me to understand that termination functions as a specific design approach rather than an enigmatic system property. The Measure functions serve as the fundamental solution because we can assign natural numbers to states which decrease by at least one when the system makes an allowed transition. The system will terminate because natural numbers cannot form endless descending sequences. The greatest common divisor computation through Euclid's algorithm ends because the divisor value decreases during each step of the process. The merge sort algorithm reaches its termination point because its subproblem sizes follow a strictly decreasing pattern. The solution follows a standard procedure which involves two steps: determine which value decreases in size and confirm it remains above zero before completing the proof. The practice of asking "what decreases?" has become central to my thinking. I naturally ask three questions when I think about evaluators or recursive functions because I need to understand the measurement process and its termination point and the point at which the recursive process ends. Language interpreters can apply this discipline through two methods: they can either count reduction steps or select an appropriate reduction approach like leftmost-outermost to achieve predictable results from non-terminating computations. The interpreter will display divergence instead of becoming stuck when the program $\Omega = (\lambda x.x x)(\lambda x.x x)$ enters an infinite loop.

The relationship between formal syntax and computational meaning became clear through the development of lambda calculus. The process of α -renaming helped me avoid variable capture while β -reduction allowed me to apply functions to their arguments which taught me that computation involves following established rewriting rules. The Church numerals proved that integers exist as constructed elements because they can be expressed through functions: the number 2 in Church numerals functions as $\lambda f.\lambda x.f(fx)$ which represents a function that performs its argument operation twice. The Church numeral for 2 when combined with the Church numeral for 3 results in the numeral for 9 through composition which demonstrates how basic substitution and rewriting operations generate complex system behavior. The fixed-point combinator $\text{fix } F \rightarrow F(\text{fix } F)$ establishes recursion through its definition which does not require any pre-existing loop structure. The evaluation process of factorial 3 through **fix** demonstrates how conditional statements and arithmetic operations and substitution mechanisms work together to generate a final value. The process requires complete rewriting because execution does not produce any "magic" results which only involve rule-based operations. The discovery that language features including recursion and higher-order functions and lazy evaluation can be expressed through pure terms with exact operational behavior brings freedom to developers. The process of mastering a language requires knowledge of its rewrite rules instead of learning its individual words.

The process of converting surface notation into structured meaning required both grammar design and parsing systems. The process of creating parse trees for arithmetic expressions demonstrated how grammar rules enforce both operator precedence and association rules. The grammar achieves nonterminal separation through three distinct nonterminals which assign Exp to handle addition operations and Exp1 to handle

multiplication operations and Exp2 to handle atomic elements. The grammar implements multiplication precedence above addition through its tree structure instead of depending on conventional rules. The tree structure remains unchanged through parentheses because they create new arrangements which replace all default settings. The evaluation process uses tree structure instead of the original string structure which makes grammar design serve a purpose beyond appearance because it affects meaning. The production of a single tree by an unambiguous grammar enables developers to create simpler interpreters which also become more dependable through their analysis. The information seemed like a minor point at the beginning of the semester but I understood its essential nature when the semester ended. The process of language design cannot exist without parsing and parsing requires meaning to function properly.

The Lean system enabled mechanized proof verification which united all these different mathematical approaches. The process of demonstrating addition properties through explicit reasoning involved three steps which included establishing hypotheses and creating the proof term that followed function construction and using rewrite lemmas (`add_zero`, `add_succ`, `add_assoc`, `add_comm`) to perform the algebraic transformations. The process of solving logic problems which involved proving different types of statements showed that proof construction shares a direct relationship with program development. The proof of $C \rightarrow D \rightarrow S$ requires a function which uses the given proof of $(C \wedge D) \rightarrow S$ to construct a pair from c and d . The relationship between specifications and implementations and between logical systems and programming code exists as a real-world phenomenon. The process of formal verification should occur as an essential part of design work instead of serving as an independent verification step that happens after design completion.

The development of these strands resulted in creating a basic lambda-calculus interpreter. The system needed three essential components for its implementation which included capture-avoiding substitution (applying lessons in α -equivalence) and a reduction strategy based on leftmost-outermost evaluation and step limits to manage divergent behavior. The interpreter demonstrates all essential elements from the course because it uses grammars to control parsing and substitution to enforce semantic rules and evaluation control through measures and strategies and termination awareness to stop infinite loops. The implementation process turned theoretical ideas into physical objects which students could touch and study. The rule of capture avoidance evolved from a set of instructions into a challenge which needed resolution. Evaluation strategies evolved from being mere notation systems into actual design elements which produce measurable effects. The interpreter demonstrated that lambda terms contain two distinct categories of expressions because some expressions lack normal forms and their programs fail to reach termination. The system becomes capable of generating useful diagnostic information through its implementation of step limits which monitor system operations. The knowledge of theoretical foundations together with their restricted applications appears to be fundamental.

Programming languages require developers to explain program behavior and to verify that their code functions properly. The system reveals both its accessible states and its permanently inaccessible states through invariants. The termination process receives support from valid measurement methods which also establish specific expense limits. Grammars determine how language creates meaning while maintaining its organizational framework. Fixed-point combinators reveal the fundamental nature of recursive operations. Proof assistants enable users to execute their reasoning while maintaining complete auditability. The combination of these concepts into a unified perspective which uses systematic reasoning and proof artifacts that others can verify and semantic-based language design creates a new way of understanding code. The process of programming shifts from attempting different solutions to creating programs through logical planning. The development of this perspective would lead to better understanding of both theoretical concepts and engineering applications. A person can write ships code with assurance when they grasp both the operational success of their code and the underlying reasons for its functionality. The GCD loop from Euclid reduces the remaining value while merge sort reduces the dimensions of its subproblems. The recipe requires selecting a reliable measurement method which removes each step completely. I now evaluate evaluators and recursion by determining which values decrease and identifying the base case location. The behavior of interpreters becomes more understandable through the use of step bound or leftmost-outermost strategy when complete normalization proves unattainable (e.g. the diverging term Ω). The process of applying α/β -reductions and

Church numerals and fixed-point combinators established a connection between the structural elements of syntax and their corresponding semantic meanings. The combination of `iterate-2` with `iterate-3` produces `iterate-9` which demonstrates how composition creates structural elements. The fixed-point rule (`fix F → F(fix F)`) provides recursion with its fundamental meaning through a process that does not require loops in the programming language. The evaluation process of `fact 3` demonstrates how conditionals and substitution work together to generate values which confirms that numerous programming elements can be expressed as pure terms that follow specific rules.

The process of creating parse trees for arithmetic expressions demonstrated how grammars use their rules to enforce both operator precedence and how operators should associate with each other. The separation of nonterminals (`Exp`, `Exp1`, `Exp2`) establishes multiplication as the tighter operator than addition while parentheses function to modify tree structures for default override. A grammar which is well designed produces trees that have no ambiguity which results in simpler and more accurate interpreters.

The process of proving addition associativity and commutativity and right-commutativity in Lean required me to develop my explicit reasoning abilities. The pattern follows a consistent sequence which starts with state hypothesis formation then follows function or contradiction construction and ends with rewrite lemma application of `add_zero` and `add_succ` and `add_assoc` and `add_comm`. The logic exercises which included implication and conjunction and negation and double negation operations demonstrated that building functions through programming follows the same process as building proofs in mathematics thus strengthening the connection between design requirements and software code.

The interpreter for lambda calculus operated as a small program which combined capture-avoiding substitution with a defined reduction strategy and step limits to manage infinite loops. The program implements all essential course concepts which include grammar-based parsing and semantic evaluation through substitution and evaluation strategy selection and program termination detection. The interpreter maintains predictable behavior when terms fail to match normal forms while providing explanations about the reasons for this behavior.

Programming languages are about explaining why programs behave as they do and how to make that behavior trustworthy. Invariants clarify reachability; well-founded measures justify termination; grammars shape meaning; fixed points express recursion; and proof assistants make reasoning executable. Together, these perspectives improve both theoretical understanding and practical implementation skills.

4 Evidence of Participation

4.1 Discussion Post 1: Code Security and Quality in Practice

Source: [The Pragmatic Engineer: Code Security with Johannes Dahse](#)

The Pragmatic Engineer features Johannes Dahse (Sonar VP of Code Security) explaining how actual security vulnerabilities originate and what software engineers must do to prevent them. The core insight is that code quality directly affects security: poorly written code becomes harder to review and maintain, extending the window in which vulnerabilities exist. Modern development relies on external libraries, making Software Composition Analysis (SCA) and CVE databases essential security tools. The discussion gains urgency with AI-assisted coding, where generated code moves faster than human review can handle, creating new verification challenges. Three emerging risks emerge: degraded code quality, missed vulnerability detection, and prompt injection attacks in LLM-based systems. The episode underscores that solid engineering practices and code comprehension remain vital for real-world security, even as developers gain access to powerful AI tools.

4.2 Discussion Post 2: Transformer Limitations and Alternative Reasoning Architectures

Source: [Sakana AI: Rethinking Transformers with Llion Jones and Luke Darlow](#)

Llion Jones and Luke Darlow from Sakana AI argue that the Transformer architecture—powering ChatGPT and most modern AI—may actually limit progress toward genuine intelligent reasoning. Jones, a co-author of the original “Attention Is All You Need” paper, describes “inventor’s remorse,” noting how Transformer success has led to “success capture,” where researchers optimize incrementally rather than explore fundamentally new ideas. They highlight the “spiral problem”: neural networks stitch together line segments to approximate spirals, mimicking understanding without grasping the underlying concept—much like how current AI generates plausible answers without genuine thinking. As an alternative, they introduce the Continuous Thought Machine (CTM), a biology-inspired model enabling step-by-step reasoning (like walking a maze rather than guessing the exit). CTM emphasizes thinking time, reflection, and backtracking—capabilities largely absent from today’s language models but essential for true reasoning.

4.3 Discussion Post 3: AI as Predictive Tool for Large Codebases

Source: [Stack Overflow Podcast: Macroscopic and AI Code Transparency](#)

Kayvon Beykpour explains how Macroscopic positions AI as a predictive lens into vast codebases, offering insights through enhanced transparency, review processes, and development tracking. The real challenge in large engineering organizations is not coding but understanding the work of thousands of engineers across thousands of files. Macroscopic uses AI-driven summaries based on Abstract Syntax Tree (AST) data, performing contextual code change analysis instead of relying on diffs alone. This enables more effective code reviews and gives leadership reliable reports without extra meetings or documentation overhead. The key insight: AI works best as an intelligence layer that helps teams reason about complex systems, yet human judgment and decision-making remain non-negotiable.

4.4 Discussion Post 4: Ethics in AI, Not Ethics of AI

Source: [David Danks: Ethics in AI, Not Ethics of AI](#)

David Danks reframes AI ethics by arguing that most AI-related ethical issues stem not from the systems themselves but from human decisions about development and deployment. Rather than assigning moral agency to AI, focus should be on the social contexts, institutional frameworks, and decision-making processes that shape system behavior. Bias, unfairness, and harm arise from choices made during data collection, modeling, and deployment—making developers, organizations, and policymakers responsible. This framing shifts discourse from vague fears of “evil AI” to concrete, actionable interventions. Danks proposes a functional approach: embed ethical analysis within AI development rather than treating ethics as a separate discipline, making responsibility tangible and outcomes measurable.

4.5 Discussion Post 5: Beyond Vibe Coding—The 70% Problem

Source: [The Pragmatic Engineer: Beyond Vibe Coding with Addy Osmani](#)

The Pragmatic Engineer presents an episode where Addy Osmani explains how artificial intelligence transforms software development yet he warns developers to avoid depending too heavily on “vibe coding” which involves using AI-generated code without proper comprehension. He explains the 70 percent problem which shows that AI tools excel at speeding up initial development work yet they fail to handle the crucial last 30 percent which decides how well the code will be maintained and how correct it will be. Osmani explains that human experts need to review and test generated code through reasoning because complex systems and changing requirements make automated code generation insufficient. He presents spec-driven development and testing as two methods which enable developers to maintain control over their work while AI tools deliver

fast results. The show presents AI as an effective tool which helps developers instead of taking their place while maintaining that software engineers need to develop their critical thinking abilities and deep system comprehension skills.

5 Conclusion

Programming languages required students to learn structural thinking through proof-based methods and performance tradeoff analysis instead of focusing on syntax memorization. A 20-year-old developer who wants to create software which functions properly in actual use learned that system operation does not guarantee either proper functionality or future system stability. The course taught me to move beyond basic code understanding toward understanding program behavior mechanisms and learning how to defend this behavior through mathematical proof which leads to effective communication of these arguments.

PL operates as the base system which maintains all other components through its honest operation. Rewriting systems and invariants appear to be theoretical concepts which directly apply to distributed system state transition analysis and compiler optimization work. The process of creating termination proofs through measure analysis helped us determine if our work would complete its tasks or continue to consume cloud resources without stopping. The initial impression of Lambda calculus and fixed points as specialized concepts revealed their fundamental role in supporting recursion and higher-order functions and functional-reactive UI patterns. Every API specification and configuration file and domain-specific language which engineers develop needs grammars and parsing as their fundamental structural components. The Lean proof system operated similarly to writing tests but required absolute precision because the system would only accept proofs when all steps were fully detailed.

The most valuable information for me emerged when the approach changed from requiring proof through homework to establishing proof which would lead to trust. The MU puzzle demonstrated its mod-3 invariant through a simple yet effective example which showed that a single line of understanding surpasses extensive brute-force calculations. The process of creating a lambda-calculus interpreter helped me understand how substitution and capture avoidance and evaluation strategies work in practice. The interpreter showed me that programming language features exist as actual decisions which developers must make. The Lean system demonstrated that proofs function as engineered artifacts which surpass their traditional role as handwritten notes because properly established lemmas become reusable library functions.

The industry vibe accepts this approach because software development today involves working with services and schemas and specs which constantly change. A PL toolkit enables you to create DSLs with defined meaning and perform termination and resource usage analysis and establish essential system properties for large-scale dependable systems. The framework helps developers determine which programming techniques to use between static types and formal specs and property-based tests and end-to-end checks.

I would introduce additional connections between theoretical concepts and practical applications if I had the opportunity to modify the course structure. The examples demonstrate how to use tie rewriting and confluence to develop systems and CRDTs and how to display termination indicators for asynchronous job processing pipelines and how to link lambda-calculus problems with a minimal typed interpreter for type safety education and how to use Lean to verify a small API contract before creating tests from it. A short module about performance and complexity would work well with termination because it would answer the question of when the program will stop running. but “will it stop before my laptop fans take off?” The proof/code lab would operate as a collaborative environment which duplicates actual engineering work by having students work in pairs to create proofs and code at the same time while they need to share their invariants through design documentation.

The main point: PL serves as the mental training facility which software engineers need to develop their skills. The training process teaches students to identify problem structures and establish rules while developing their ability to support their arguments. The approach he uses represents the practical nature of Gen Z because

he delivers results while understanding their operational logic and maintaining proof of their effectiveness during critical situations.