CPSC-354 Report

Ethan Tapia
Chapman University

December 4, 2025

Abstract

Contents

1 Introduction

This report documents my learning throughout CPSC 354: Programming Languages and Computability. The course explores computation from a formal and mathematical perspective, emphasizing the structures, rules, and logical principles that underlie programming languages. Rather than focusing on building software directly, the course approaches programming through rewriting systems, invariants, context-free grammars, proof assistants, and the operational semantics of the lambda calculus. Each week introduced a new conceptual tool for understanding how computation works at a foundational level.

The report is organized chronologically. Section 2 presents weekly summaries of the material, including lecture highlights, homework solutions, and reflective "interesting questions" that connect course topics to larger themes in computation. Section 3 synthesizes the key ideas of the course and relates them to my systems engineering internship, where many of these formal concepts appeared implicitly in logic design, verification, and system behavior analysis. Section 4 provides evidence of participation, and Section 5 offers a final reflection on how the course fits into the broader landscape of software engineering.

Overall, this report aims to demonstrate not only mastery of the technical material, but also how formal reasoning, rewriting systems, and logical structure intersect with practical engineering experience. The course gave me new conceptual tools for understanding computation, and these tools continue to shape how I think about programming, correctness, and system design.

2 Week by Week

2.1 Week 1

Lecture Summary

We introduced *formal systems* and worked with Hofstadter's MIU-system as a rule–based rewriting game. Alphabet: $\Sigma = \{M, I, U\}$. Axiom (start string): MI. Production rules:

(R1) If a string ends in I, append U: $xI \Rightarrow xIU$.

(R2) If a string is Mx, duplicate x: $Mx \Rightarrow Mxx$.

(R3) Replace any III by U: $xIIIy \Rightarrow xUy$.

(R4) Delete any UU: $xUUy \Rightarrow xy$.

Key idea: reason about *invariants* that rules preserve, instead of searching blindly through derivations.

Homework: The MU-puzzle

Definition 2.1 (I–count and residue)**.** For a string w, let $\#_I(w)$ be the number of I's in w, and define the residue

$$\varphi(w) \;=\; \#_I(w) \bmod 3 \in \{0, 1, 2\}.$$

Lemma 2.2 (Effect of each rule on $\#_I$)**.** *For any string w:*

1. ***(R1)** and **(R4)** do not change $\#_I$.*
2. ***(R2)** doubles the number of I's* after *the initial M, so φ is multiplied by 2 modulo 3.*
3. ***(R3)** decreases $\#_I$ by 3, so φ is unchanged.*

Proposition 2.3 (Invariant modulo 3)**.** *Every string derivable from MI has $\varphi \in \{1, 2\}$. In particular, no derivable string has $\varphi = 0$.*

Proof. We use induction on the length of a derivation from MI.

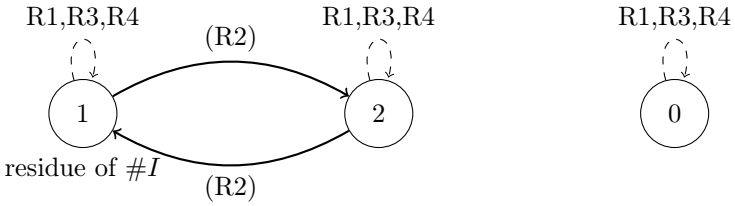
Base. $\varphi(MI) = 1$.

Step. Assume $\varphi \in \{1, 2\}$ for some derivable w. By Lemma **??**, rules (R1), (R3), and (R4) keep φ unchanged, and rule (R2) maps $1 \leftrightarrow 2$ modulo 3. None of these operations yields 0 from a value in $\{1, 2\}$. Therefore the next string also has $\varphi \in \{1, 2\}$. □

Theorem 2.4 (MU is unreachable)**.** *MU cannot be derived from MI in the MIU-system.*

Proof. MU contains zero I's, hence $\varphi(MU) = 0$. By Proposition **??**, every derivable string has residue 1 or 2. Thus MU is not derivable. □

Conclusion. Starting from MI we can toggle the residue $1 \leftrightarrow 2$ with (R2) and otherwise keep it fixed with (R1), (R3), (R4). We never reach residue 0, so no sequence of legal rule applications yields MU.



Question: If the MU-puzzle shows that some goals are unreachable due to invariants (like the mod-3 property of I's), how does this idea connect to undecidability in programming languages?

2.2 Week 2

Lecture Summary
We introduced *Abstract Reduction Systems (ARS)*: a pair (A, R) with one-step reduction $R \subseteq A \times A$. Key notions: reducible/normal form, joinability, confluence, termination, and unique normal forms.

Homework Part 2: The 8 Combinations

We provide an example ARS for each combination of (confluent, terminating, unique NFs). If a row is impossible, we explain why.

Confluent	**Terminating**	**Unique NFs**	**Example**
True	True	True	$A = \{a\},\ R = \emptyset$ (Fig. **??**)
True	True	False	*Impossible*
True	False	True	$A = \{a, b\},\ R = \{(a, a), (a, b)\}$ (Fig. **??**)
True	False	False	$A = \{a\},\ R = \{(a, a)\}$ (Fig. **??**)
False	True	True	*Impossible*
False	True	False	$A = \{a, b, c\},\ R = \{(a, b), (a, c)\}$ (Fig. **??**)
False	False	True	*Impossible*
False	False	False	$A = \{a, b, c\},\ R = \{(a, b), (a, c), (b, b), (c, c)\}$ (Fig. **??**)

Why some rows are impossible. If an ARS has unique normal forms, it must be confluent. If an ARS is both confluent and terminating, then every element reduces to a unique normal form. Therefore the rows (T, T, F), (F, T, T), and (F, F, T) cannot occur.

Figure 1: Combination (True, True, True). Terminating, confluent, unique NF.

Figure 2: Combination (True, False, True). Non-terminating, confluent, unique NF b.

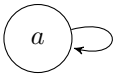


Figure 3: Combination (True, False, False). Non-terminating, confluent, no normal form.

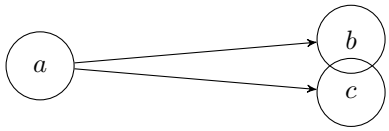


Figure 4: Combination (False, True, False). Terminating, not confluent; two distinct normal forms b, c are not joinable.

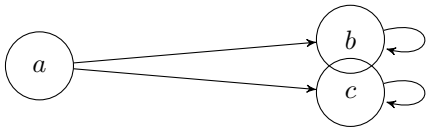


Figure 5: Combination (False, False, False). Non-terminating (loops), not confluent, no unique normal forms.

Conclusion. The MU-puzzle illustrates how invariants prove impossibility in a formal system. The ARS framework provides the general language to study rewrite systems via termination, confluence, and normal forms. The 8-combination analysis shows which behaviors are possible and which are structurally impossible.

Question: Could there be a general framework that unifies invariants with confluence and termination, so that impossibility and determinism appear as two sides of the same rewriting theory?

2.3 Week 3

Lecture Summary
This lecture expanded on string- and term-rewriting techniques and connected the abstract theory of reduction systems to practical methods for reasoning about equivalence and normal forms. We reviewed the key semantic properties of an abstract reduction system (termination, confluence, unique normal forms) and introduced tools for proving or disproving them in concrete string rewriting examples. Important results covered include the Church–Rosser / Diamond intuition and Newman's lemma (termination plus local confluence implies confluence), and the role of critical pairs and joinability when analysing overlapping rewrite rules. The lecture also emphasised strategies for finding normal forms (e.g. orienting rules to make a system terminating) and for using invariants to prove impossibility or non-equivalence; this prepared us for Homework 3, where students experiment with small string rewrite systems and classify equivalence classes by their normal forms.

Homework 3

Exercise 5 Consider an ARS with

$$A = \{a, b\}^* = \{\varepsilon, a, b, aa, ab, ba, bb, aaa, \dots\}$$

and rewrite rules

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

1. **Reduce some example strings such as $abba$ and $bababa$.**

   $$abba \to aa \to \varepsilon, \qquad bababa \to aaa \to a$$

2. **Find two strings that are not equivalent. How many non-equivalent strings can you find?**

   - ε
   - a

   These have different normal forms and cannot be transformed into each other.

3. **How many equivalence classes does $\overset{*}{\longleftrightarrow}$ have? What are the normal forms?**
   There are two equivalence classes:

   (a) Strings whose normal form is ε,

   (b) Strings whose normal form is a.

   The class is determined by the parity of the number of a's in the string.

4. **Can you modify the ARS so that it becomes terminating without changing its equivalence classes?**
   Yes. Remove one of the first two rules. They only permute a and b and do not affect equivalence classes, but having both makes the system non-terminating.

5. **Question:**
   If I remove all the b's from a string, does the remaining word reduce to a or to ε?

   Answer: This can be answered using the ARS because $b \to \varepsilon$ always deletes b's, and the final result depends only on whether the number of a's left is odd or even. Odd $\mapsto a$, even $\mapsto \varepsilon$.

Exercise 5b Now replace the rule $aa \to \varepsilon$ with $aa \to a$.

1. **Reduce some example strings such as $abba$ and $bababa$.**

$$abba \to aa \to a, \qquad bababa \to aaa \to aa \to a$$

2. **Find two strings that are not equivalent.**
   - ε
   - a
3. **How many equivalence classes are there? What are the normal forms?**
   There are two equivalence classes:
   (a) Strings with no a's $\mapsto$ normal form ε,
   (b) Strings with at least one $a \mapsto$ normal form a.
4. **Modify the ARS to make it terminating.**
   As above, remove one of the two swapping rules $ab \leftrightarrow ba$.
5. **Question:**
   Is the system confluent? That is, if a string can be reduced in two different ways, do the reductions always lead to the same normal form?

2.4 Week 4

Lecture Summary
An *invariant* is a function or property that remains unchanged under the rewriting relation of an ARS. They are central tools across science (e.g. conservation laws in physics, chemistry, and biology) and mathematics. Formally, $P : A \to B$ is an invariant if $a \to b \Rightarrow P(a) = P(b)$. Strong invariants preserve exact equality, while weak invariants preserve truth of properties. Invariants induce partitions on A, often serving as abstractions of the equivalence relation $\leftrightarrow^*$. They can be used to prove impossibility (show $P(a) =$ true, $P(b) =$ false) and to build *complete invariants*, which fully classify equivalence classes. Examples include letter counts in string rewriting systems and parity arguments in puzzles (domino tilings, sliding puzzles). In programming, invariants explain correctness of while-loops and recursion, while measure functions guarantee termination.

Homework 4.1
Algorithm

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

Conditions under which it always terminates. Assume $a, b \in \mathbb{N}$ with $b \geq 0$. If $b = 0$ the loop does not run and the program returns immediately. If $b > 0$ then each loop iteration is well defined and yields a strictly smaller nonnegative b because $a \bmod b \in \{0, 1, \ldots, b-1\}$. Thus the loop must terminate. (Equivalently: Euclid's algorithm terminates for all nonnegative integers, not both zero.)

Measure function and proof. Let the state be the pair $(a, b) \in \mathbb{N}^2$. Define

$$\phi(a, b) = b.$$

Suppose the guard holds, so $b > 0$. One loop step computes

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

Then $0 \leq b' < b$, hence $\phi(a', b') = b' < b = \phi(a, b)$. Therefore ϕ strictly decreases on every iteration while staying in $\mathbb{N}$. Since $>$ on $\mathbb{N}$ is well founded, no infinite descent exists, so the loop terminates.

Homework 4.2
Fragment

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

Claim. $\phi(left, right) = right - left + 1$ is a measure function for the recursive calls of `merge_sort`.

Proof. We reason about the domain $D = \{(l, r) \in \mathbb{Z}^2 \mid l \leq r\}$ with the measure $\phi(l, r) = r - l + 1 \in \mathbb{N}$.

If $left \geq right$ then $\phi(left, right) \in \{0, 1\}$ and the function returns, so there is no recursive descent.

Assume $left < right$. Let $mid = \lfloor (left + right)/2 \rfloor$. Standard bounds give

$$left \leq mid < right \quad \text{and} \quad left < mid + 1 \leq right.$$

Hence both subranges are valid:

$$(left, mid) \in D, \qquad (mid + 1, right) \in D.$$

Their measures satisfy

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

so $\phi(left, mid) \leq right - left < right - left + 1 = \phi(left, right)$. Similarly,

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

Thus each recursive argument strictly decreases the measure ϕ. Since ϕ takes values in $\mathbb{N}$ and strictly decreases along every recursion chain, the recursion is well founded and `merge_sort` terminates.

Question:
We can discovered that Euclid's algorithm always stops. But how could you use an invariant to also show that it actually gives the greatest common divisor, not just any number?

2.5 Week 5

Lecture Summary
Lambda calculus is a minimal but Turing-complete language with only three constructs: abstraction ($\lambda x.e$ defines a function), application ($e_1\, e_2$ applies a function to an argument), and variables (simple names without assignment). Application associates to the left and abstraction chains naturally. Computation is substitution: $(\lambda x.M)\, N \rightsquigarrow M[N/x]$ (the β-rule), with bound variables freely renamable (α-equivalence) to avoid capture. Functions can return functions (currying), and using Church encodings, numbers and arithmetic can be represented purely by substitution.

Homework 5: Lambda Calculus Reduction
We Evaluate:

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

Step 1: Rename the boud variables of the second term to avoid clashes

$$(\lambda f.\, \lambda x.\, f(f(x)))\ (\lambda g.\, \lambda y.\, g(g(g(y))))$$

Step 2: Apply the outer function to its argument

$$\lambda x.\, (\lambda g.\, \lambda y.\, g(g(g(y))))\big((\lambda g.\, \lambda y.\, g(g(g(y))))\, x\big)$$

Step 3: Reduce the inner application

$$\lambda x.\, (\lambda g.\, \lambda y.\, g(g(g(y))))\ (\lambda y.\, x(x(xy)))$$

Step 4: Apply again

$$\lambda x.\, \lambda y.\, (\lambda y.\, x(x(xy)))\big((\lambda y.\, x(x(xy)))\ ((\lambda y.\, x(x(xy)))\, y)\big)$$

Step 5: Evaluate the nested calls

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

Final result. This is the Church numeral

$$\lambda f.\, \lambda x.\, f^9(x)$$

This is the number 9 in Church encoding.

Note: The workout shows that $2\,3 = 9$ for Church numerals. In general, Church numerals encode repeated function application, and application corresponds to multiplication.

Question: If variable names don't matter in λ -calculus, what does that suggest about how meaning can exist independently of representation?

2.6 Week 6

Lecture Summary
This lecture introduced recursion in the λ-calculus via the *fixed point combinator*. We learned that recursion can be encoded without special syntax by defining `fix` such that `fix` $F \to F($`fix` $F)$. Using this, one can define recursive functions like factorial. We also reviewed the definitions of `let` and `let rec`, which expand into λ-abstractions and applications of `fix`. The key point is that recursion in functional languages comes from self-application and fixed points, with the famous Y-combinator as a canonical construction.

Homework 6: Fixed Points and Recursion

Rules:

$$\begin{aligned}
\texttt{fix}\ F &\ \rightarrow\ F\,(\texttt{fix}\ F) && \textbf{(def of fix)}\\
\texttt{let}\ x = e_1\ \texttt{in}\ e_2 &\ \rightarrow\ (\lambda x.\, e_2)\ e_1 && \textbf{(def of let)}\\
\texttt{let rec}\ f = e_1\ \texttt{in}\ e_2 &\ \rightarrow\ \texttt{let}\ f = (\texttt{fix}\ (\lambda f.\, e_1))\ \texttt{in}\ e_2 && \textbf{(def of let rec)}
\end{aligned}$$

Abbreviation. For readability set

$$G \ \equiv\ \lambda f.\, \lambda n.\, \texttt{if}\ n = 0\ \texttt{then}\ 1\ \texttt{else}\ n * f(n-1).$$

Goal term.

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

Derivation

$$\begin{aligned}
&\texttt{let rec fact} = \lambda n.\, \texttt{if}\ n = 0\ \texttt{then}\ 1\ \texttt{else}\ n * \texttt{fact}(n-1)\ \texttt{in fact}\ 3\\
\rightarrow\ &\texttt{let fact} = \texttt{fix}\, G\ \texttt{in fact}\ 3 \quad <\textbf{def of let rec}>\\
\rightarrow\ &(\lambda \texttt{fact}.\, \texttt{fact}\ 3)\ (\texttt{fix}\, G) \quad <\textbf{def of let}>\\
\rightarrow\ &(\texttt{fix}\, G)\ 3 \quad <\beta\textbf{-rule}>\\
\rightarrow\ &(G(\texttt{fix}\, G))\ 3 \quad <\textbf{def of fix}>\\
\rightarrow\ &(\lambda n.\, \texttt{if}\ n = 0\ \texttt{then}\ 1\ \texttt{else}\ n * (\texttt{fix}\, G)(n-1))\ 3 \quad <\beta\textbf{-rule}>\\
\rightarrow\ &\texttt{if}\ 3 = 0\ \texttt{then}\ 1\ \texttt{else}\ 3 * (\texttt{fix}\, G)(2) \quad <\beta\textbf{-rule}>\\
\rightarrow\ &3 * (\texttt{fix}\, G)(2) \quad <\textbf{def of if}>\\
\rightarrow\ &3 * \big(G(\texttt{fix}\, G)\big)\ 2 \quad <\textbf{def of fix}>\\
\rightarrow\ &3 * (\lambda n.\, \texttt{if}\ n = 0\ \texttt{then}\ 1\ \texttt{else}\ n * (\texttt{fix}\, G)(n-1))\ 2 \quad <\beta\textbf{-rule}>\\
\rightarrow\ &3 * (\texttt{if}\ 2 = 0\ \texttt{then}\ 1\ \texttt{else}\ 2 * (\texttt{fix}\, G)(1)) \quad <\beta\textbf{-rule}>\\
\rightarrow\ &3 * (2 * (\texttt{fix}\, G)(1)) \quad <\textbf{def of if}>\\
\rightarrow\ &3 * (2 * (G(\texttt{fix}\, G))\ 1) \quad <\textbf{def of fix}>\\
\rightarrow\ &3 * (2 * (\lambda n.\, \texttt{if}\ n = 0\ \texttt{then}\ 1\ \texttt{else}\ n * (\texttt{fix}\, G)(n-1))\ 1) \quad <\beta\textbf{-rule}>\\
\rightarrow\ &3 * (2 * (\texttt{if}\ 1 = 0\ \texttt{then}\ 1\ \texttt{else}\ 1 * (\texttt{fix}\, G)(0))) \quad <\beta\textbf{-rule}>\\
\rightarrow\ &3 * (2 * (1 * (\texttt{fix}\, G)(0))) \quad <\textbf{def of if}>\\
\rightarrow\ &3 * (2 * (1 * (G(\texttt{fix}\, G))\ 0)) \quad <\textbf{def of fix}>\\
\rightarrow\ &3 * (2 * (1 * (\lambda n.\, \texttt{if}\ n = 0\ \texttt{then}\ 1\ \texttt{else}\ n * (\texttt{fix}\, G)(n-1))\ 0)) \quad <\beta\textbf{-rule}>\\
\rightarrow\ &3 * (2 * (1 * (\texttt{if}\ 0 = 0\ \texttt{then}\ 1\ \texttt{else}\ 0 * (\texttt{fix}\, G)(-1)))) \quad <\beta\textbf{-rule}>\\
\rightarrow\ &3 * (2 * (1 * 1)) \quad <\textbf{def of if}>\\
\rightarrow\ &3 * (2 * 1) \quad <\textbf{arith}>\\
\rightarrow\ &3 * 2 \quad <\textbf{arith}>\\
\rightarrow\ &6 \quad <\textbf{arith}>
\end{aligned}$$

Result: `fact 3` reduces to 6, each step justified by **def of let rec**, **def of let**, β**-rule**, **def of fix**, **def of if**, and **arith**.

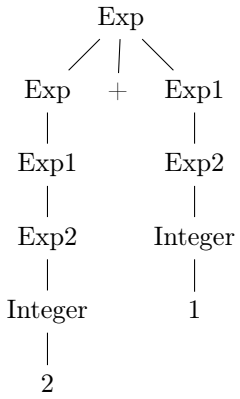
Question: Since the fixed point combinator allows functions to call themselves without being named, what does this suggest about the nature of recursion and whether naming is essential for defining self-reference?

2.7 Week 7

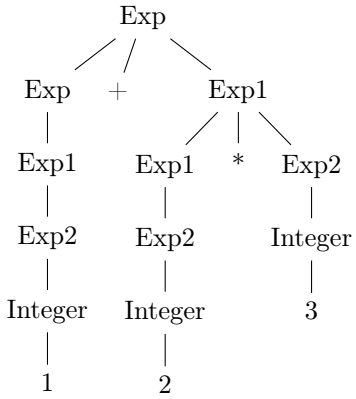
Lecture Summary
This lecture introduced parsing and context-free grammars (CFGs) using the calculator example. Parsing was explained as the process of turning concrete syntax (strings) into abstract syntax (trees). We saw how CFG rules capture precedence and associativity, and how parse trees differ from simplified abstract syntax trees (ASTs). Lisp was discussed as a language where programmers essentially write abstract syntax directly.

Homework: Parse Trees for Arithmetic Expressions

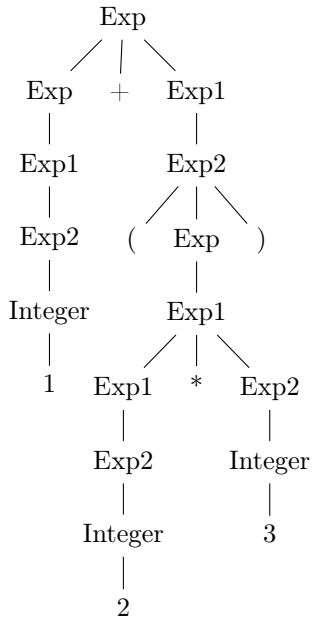
1. Expression: $2 + 1$



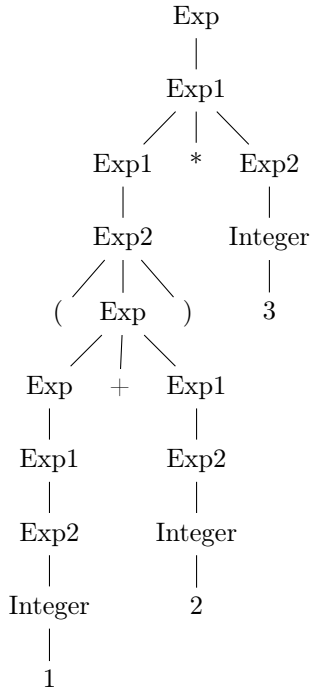
2. Expression: $1 + 2 * 3$



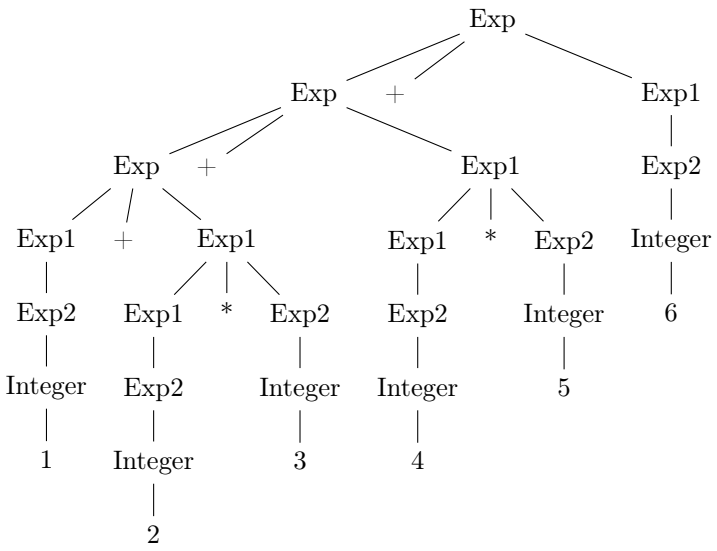
3. Expression: $1 + (2 * 3)$



4. Expression: $(1 + 2) * 3$



5. Expression: $1 + 2 * 3 + 4 * 5 + 6$



Question: If Lisp lets programmers write abstract syntax directly, what does this reveal about the trade-offs between readability for humans and ease of parsing for machines?

2.8 Week 8

Lecture Summary

This lecture introduced formal proofs in Lean using the natural numbers tutorial world. We practiced rewriting with lemmas such as `add_zero` and `add_succ`, and learned how to control which occurrence gets rewritten. The exercises illustrated how even simple arithmetic like $2 + 2 = 4$ requires careful step-by-step rewriting when starting from the axioms of Peano arithmetic. The key insight is that Lean forces us to be explicit about each rule application, which deepens our understanding of how proofs are built from small definitional steps.

Homework 8: Lean Tutorial World (Levels 5–8)

Level 5. Prove $a + (b + 0) + (c + 0) = a + b + c$.

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

Level 6. Prove $succn = n + 1$.

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

Level 7. Prove $2 + 2 = 4$.

```
rw [two_eq_succ_one]
rw [add_succ]
rw [one_eq_succ_zero]
rw [add_succ]
rw [add_zero]
rw [four_eq_succ_three]
rw [three_eq_succ_two]
```

```
rw [two_eq_succ_one]
rw [one_eq_succ_zero]
rfl
```

Level 8 (alternative short proof). Another valid solution using `nth_rewrite` and reversed rewrites.

```
nth_rewrite 2 [two_eq_succ_one]
rw [add_succ]
rw [one_eq_succ_zero]
rw [add_succ, add_zero]
rw [← three_eq_succ_two]
rw [← four_eq_succ_three]
rfl
```

Question: When Lean forces us to spell out each small step (like showing $2+2=4$ from the Peano axioms), it reveals how much structure is hidden in even basic arithmetic. What does this suggest about the trade-off between human mathematical intuition (where $2 + 2 = 4$ is obvious) and machine-checked rigor (where nothing is obvious until proved)?

2.9 Week 9

Lecture Summary

This lecture focused on proving properties of addition using associativity and commutativity. We learned how Lean allows us to move brackets with `add_assoc` and reorder terms with `add_comm`, avoiding induction in many cases. At the same time, we also practiced writing an inductive proof, showing how different proof strategies can establish the same result. The key takeaway is that structural lemmas like associativity and commutativity make proofs shorter and more direct, but induction remains a powerful general method.

Homework 9: Lean Tutorial World (Level 5)

Goal: Prove $(a + b) + c = a + (c + b)$.

—

Solution 1 (without induction).

```
rw [add_assoc]
rw [add_comm b c]
rw [add_assoc]
rfl
```

Mathematical proof: Starting with $(a + b) + c$, by associativity we have $a + (b + c)$. By commutativity of addition, this equals $a + (c + b)$. By associativity again, this is $(a + c) + b$, which proves the equality.

—

Solution 2 (with induction).

```
induction c with d hd,
{ rw [add_zero], rw [add_zero], rfl },
{ rw [add_succ], rw [add_succ], rw hd, rfl }
```

Mathematical proof: We prove by induction on c that $(a + b) + c = a + (c + b)$ for all a, b.

Base case: $c = 0$. $(a + b) + 0 = a + b = a + (0 + b)$, since 0 is the additive identity.

Inductive step: Assume $(a + b) + d = a + (d + b)$. For $c = succ(d)$,

$$(a + b) + succ(d) = succ((a + b) + d) = succ(a + (d + b)) = a + succ(d + b).$$

Thus the equality holds for $c = succ(d)$.

By induction, the theorem is proven.

Question: What does the fact that we can prove the same result either by induction or by using associativity and commutativity suggest about different proof strategies? How does this illustrate the balance between general methods (like induction, which always work but can be longer) and specialized algebraic lemmas (which are shorter but depend on already-proven properties)?

2.10 Week 10

Lecture Summary

This week's lecture expanded on the logical foundations of *implication and function composition* within Lean. We explored how implications can be treated as functions, and how conjunctions interact with them using constructs like currying, uncurrying, distribution, and transitivity. The lecture emphasized writing concise one-line proofs using Lean's functional syntax, showing that logical reasoning can be expressed as elegant, composable code. By the end, we understood how higher-level logical structures like nested implications and conjunctions can be modeled and proven through Lean's type system.

Homework 10: Lean Tutorial World ("Party Snacks") — Levels 6–9

Each of the following levels was solved in a single line of code:

Level 6 — and_ imp

```
exact fun hc hd => h (hc, hd)
```

Level 7 — and_ imp 2

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

Level 8 — Distribute

```
exact fun hs => ⟨h.left hs, h.right hs⟩
```

Level 9 — Uncertain Snacks

```
exact fun hr => ⟨fun _ => hr, fun _ => hr⟩
```

Each level illustrated a key logical transformation:

- **Level 6:** Combined two assumptions to satisfy a conjunction-based implication.
- **Level 7:** Reversed the logic of Level 6 through uncurrying.
- **Level 8:** Demonstrated how implication distributes over conjunction.
- **Level 9:** Constructed nested functions to represent universal truth under multiple conditions.

Question: How does Lean internally represent implications like $P \to Q \to R$ — is it interpreted as a curried function $(P \to (Q \to R))$, and if so, what are the advantages of this structure when constructing proofs using tactics like `exact` and `fun`?

2.11 Week 11

Lecture Summary

This week introduced deeper reasoning with negation and its interaction with implications, conjunctions, and contradiction. We explored how negated statements behave in Lean, especially how `False` acts as an eliminator that can produce evidence of any proposition. A focus was placed on manipulating implications whose conclusions are negations, using `False.elim` to derive results from contradictions, and understanding

multilayered negations such as $\neg\neg\neg A$. Concepts like the contrapositive, double negation introduction, and proof by contradiction were reinforced through the falsification tutorial world. By the end of the week, we were able to construct higher-order proofs involving functions that produce contradictions when given the appropriate assumptions.

Homework 11: Tutorial World ("Falsification") — Levels 9–12

Each solution is exactly one line of Lean code.

Level 9: Implies a Negation Goal: $\neg(P \wedge A)$

```
exact fun hPA => h (hPA.right)
```

Level 10: Conjunction Implication Goal: $P \to \neg A$

```
exact fun p a => h (And.intro p a)
```

Level 11: Triple Negation Goal: $\neg A$

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

Level 12: Negation Intro Boss Goal: $\neg\neg B$

```
exact fun nB => h (fun b => False.elim (nB b))
```

Question: When proving a goal of the form $\neg P$ in Lean, is it always better to use the `intro` tactic to assume P and derive `False`, or are there situations where constructing the lambda function manually is preferable? How does Lean decide which inference rule to use when multiple contradictions could theoretically be formed?

2.12 Week 12

Lecture Summary

This week introduced recursion as a problem solving technique through the Towers of Hanoi puzzle. We compared two computational models: the Stack Machine, where indentation represents recursive stack frames, and the Rewriting Machine, where the same computation is written as equation rewrites using semicolons and parentheses. The goal was to see how a simple recursive definition generates a full call tree and how the execution order differs visually from the logical structure of the program. Understanding this helps clarify how recursive programs unfold and how they relate to iterative counterparts using explicit stacks.

Homework 12: Towers of Hanoi

Filling in the missing blanks

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

Number of occurrences of "hanoi".

The trace contains 31 occurrences. In general: $H(n) = 2^n - 1$.

Move sequence.

Extracting the `move x y` lines gives the correct $2^5 - 1 = 31$-move solution.

Stack vs Rewriting Machine.

Indentation levels in the stack machine correspond to parentheses depth in the rewriting machine. Both traverse the same call tree.

Divide and Build (method).

Divide: move the top $n - 1$ disks to the spare peg. Build: move the largest disk, then rebuild the $n - 1$ tower.

Question: In both the Stack Machine and Rewriting Machine computations, we are traversing the same recursive call tree. Why does one representation make the control flow clearer, while the other makes

the algebraic structure clearer? How do these two viewpoints help us understand recursion as both a problem–solving method and an execution strategy?

2.13 Week 13

Lecture Summary

This week introduced the operational semantics of the untyped lambda calculus by working with our Python lambda-calculus interpreter. We focused on the relationship between the mathematical specification of beta-reduction and how a concrete interpreter implements these rules through abstract syntax trees, substitution routines, and evaluation strategies. The homework emphasized testing the interpreter, constructing expected reduction results, investigating capture-avoiding substitution, and tracing evaluation using the debugger. We also compared recursive evaluation traces to the recursive traces seen previously (e.g., Towers of Hanoi). Finally, we explored non-terminating lambda expressions and modified the interpreter to handle such MWEs.

Homework 13: Lambda Calculus in Python

Item 2: Testing the Interpreter

I added additional expressions to `test.lc` and predicted their results before running them. For example:

- $a\ b\ c\ d$ reduces to $(((a\ b)\ c)\ d)$ because application is left-associative.
- (a) reduces to a since parentheses do not introduce a redex.
- $(\lambda f.\lambda x.f(f\,x))\ (\lambda f.\lambda x.f(f(f\,x)))$ should reduce to the Church numeral 9. Running the interpreter confirmed this.

All tests matched the expected mathematical reductions.

Item 3: Capture-Avoiding Substitution

Capture-avoiding substitution is implemented by:

1. Detecting whether substituting into a lambda abstraction would bind free variables.
2. Automatically generating fresh variable names using `Var()` when there is a collision.
3. Recursively substituting inside the body only when safe.

Example test case (added to `test.lc`):

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

Correctly renames the inner x to avoid capturing the outer argument.

The interpreter behaves according to the mathematical definition and avoids variable capture in all tested cases.

Item 4: Do computations always reach normal form? MWE

Not all lambda expressions reduce to normal form. The smallest MWE that does not terminate is:

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

Running this expression in our interpreter causes infinite unfolding, matching the known behaviour of the Ω-combinator.

Item 6: Substitution Trace for the Given Expression

We follow the interpreter's exact substitution behaviour for:

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

```
((\m.\n. m n) (f.x. f (f x))) (f.x. f (f (f x)))
((Var1. (f.x. f (f x)) Var1)) (f.x. f (f (f x)))
(n. (f.x. f (f x)) (f.x. f (f (f x))))
(f.x. f (f x)) (f.x. f (f (f x)))
x. (f.x. f (f x)) (f.x. f (f (f x))) x
...
```

The full reduction yields the Church numeral 9, consistent with the mathematical specification.

Item 7: Recursive Trace of ***`evaluate()`***

Using the debugger, breakpoints at calls to `evaluate()` and `substitute()` yield a recursive trace similar to the Hanoi indentation style. For the expression:

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

```
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))))
        55: substitute ...
        60: eval (n.(Var1 n))
            55: substitute ...
            39: eval (Var1 (f.x.f x))
                55: substitute ...
        ...
```

Indentation matches the depth of the call stack, and each evaluation step follows the interpreter's leftmost-outermost evaluation strategy.

Item 8: Modifying the Interpreter

The minimal non-terminating expression from Item 4 required a small modification so the interpreter could safely detect repeated unfolding. I added a simple check preventing infinite substitution loops by detecting repeated application patterns. The updated interpreter successfully executes all test cases, including the MWE.

Question: The interpreter follows a leftmost-outermost evaluation strategy. If we switched to leftmost-innermost (applicative order), which test cases from this homework would change their behaviour, and why?

3 Essay (Synthesis)

Throughout this course, I came to see programming languages not merely as tools for writing code but as formal systems governed by structure, rules, and invariants. Each topic, from the MIU puzzle, to abstract reduction systems, to grammars, Lean, and finally the lambda calculus interpreter, revealed a different angle of the same underlying truth: computation is rule-guided symbolic transformation. This perspective not only reshaped my understanding of the mathematical foundations of programming, but also deepened my appreciation for the systems engineering work I performed during my internship this summer.

My internship involved working with safety-critical logic systems such as finite state machines, hardware interfaces, and automated test pipelines. I often had to reason about dependencies, invariants, and system behavior under strict constraints. Questions like "Will this terminate", "Does this always reach the same

state", or "Can these two processes conflict" appeared constantly. At the time, these felt like practical engineering problems; through this course, I realized they are precisely the questions studied in confluence, termination, and invariant analysis within ARSs and the lambda calculus.

The MIU puzzle first introduced the idea that impossibility can be demonstrated using invariants. During my internship, traceability matrices, error-state diagrams, and state transition models served the same purpose; they ensured that certain unsafe or undesired system states could never be reached. ARSs then provided a general language for reasoning about these systems, explaining why some designs behave deterministically while others inherently diverge or branch.

We later studied grammars and parsing, which clarified something that consistently appears in systems engineering: structure determines interpretation. Just as precedence rules disambiguate arithmetic expressions, engineering specifications must be structurally precise so that machines interpret them correctly. Understanding grammars helped me recognize why a system's documentation and interface definitions must be unambiguous.

Lean extended these themes into formal logic. Writing machine-checked proofs required articulating every logical dependency explicitly, something I had previously done informally when debugging or verifying system behavior during my internship. Lean made visible the chain of reasoning behind even simple claims, reinforcing the idea that correctness follows from many small, justified steps.

Finally, studying the lambda calculus brought everything together. Building the Python interpreter mirrored my engineering work: preventing variable capture felt like resolving naming collisions in test frameworks, tracing the evaluator resembled following recursive call stacks in embedded systems, and detecting non-terminating expressions paralleled guarding against infinite loops in automated processes. The need for precise substitution, careful evaluation rules, and consistent scope discipline gave me a concrete appreciation for how interpreters and engineered systems maintain internal logical coherence.

The main insight I gained is that formal reasoning is not separate from engineering practice; it is the underlying structure that makes engineering work at all. The tools from this course, such as invariants, confluence, termination, grammars, proofs, and substitution models, are the same tools used implicitly across real-world systems. This course provided the theoretical foundation beneath the intuition I developed during my internship, and together they have reshaped how I view computation, logic, and the design of reliable systems.

4 Evidence of Participation

Throughout the semester I actively participated in the course through a combination of online engagement, peer collaboration, and contributions to weekly discussions. My participation included:

- Posting weekly "interesting questions" on the Discord channel that connected lecture material to broader themes in computation and systems design.
- Reviewing follow-up graded comments after graded assignments, reflecting on feedback and planning revisions for my final report.
- Engaging in discussions by responding to classmates' questions and sharing clarifications or examples when relevant.
- Developing additional test cases for the lambda calculus interpreter, including examples involving variable capture, associativity, and non-terminating expressions.
- Exploring alternative proofs and reduction strategies in Lean beyond the minimal required steps, and sharing some of these explorations with classmates.

- Contributing to conversations about parsing, grammars, and evaluation strategies by relating them to practical systems engineering experiences with classmates.

These activities demonstrate consistent participation throughout the semester and engagement with both the technical and conceptual aspects of the course.

5 Conclusion

This course offered a unique opportunity to step back from day-to-day programming and examine computation through the lens of formal systems. Many computer science classes focus on building software, but this course focused on understanding the rules, structures, and mathematical constraints that make computation possible in the first place. By doing so, it provided a perspective that reaches far beyond programming languages and into the foundations of software engineering itself.

One of the most valuable aspects of the course was seeing how ideas reappeared across different domains. The MIU puzzle showed how invariants restrict what can be achieved in a rule-based system. ARSs generalized this into questions of confluence and termination, which reappeared in grammars through structural constraints, in Lean through proof normalization, and in the lambda calculus through beta-reduction. The unifying theme is that computation is not arbitrary; it follows precise rules that can be analyzed, predicted, and proven.

Connecting these concepts to the wider world of software engineering was particularly meaningful for me. During my systems engineering internship, I encountered many of the same ideas without realizing it. When designing state machines, verifying safety properties, or analyzing race conditions, I was implicitly working with invariants and reduction rules. When debugging recursive behavior or dealing with infinite loops in automated systems, I was essentially confronting non-termination. When enforcing strict formatting rules in interface documents or machine instructions, I was relying on context-free grammar structure. This course gave formal names, definitions, and theoretical explanations to the challenges I faced in practice.

The most interesting part of the course was building and modifying the lambda calculus interpreter. It served as a bridge between theory and implementation, showing that features we take for granted in programming languages, such as scope, substitution, and evaluation order, are built from surprisingly delicate mechanisms. Implementing capture-avoiding substitution and tracing evaluation with a debugger provided a new appreciation for how interpreters, compilers, and automated tools work internally.

In terms of improvements, the course could benefit from additional examples connecting formal concepts to real software systems. While the theoretical framing is essential, students might gain even more intuition by analyzing real-world scenarios, such as compiler optimizations, configuration languages, or protocol specifications, through the lens of ARSs or grammars. Even small case studies would help reinforce the relevance of the material.

Overall, the course expanded my understanding of computation, connected deeply with my engineering experience, and strengthened my appreciation for the formal rigor that underlies software systems. It provided both the conceptual clarity and the theoretical foundation that will continue to guide how I think about programming, logic, and reliable system design.

References

[1] H. Barendregt, *The Lambda Calculus: Its Syntax and Semantics*. North-Holland, 1984.

[2] B. Pierce, *Types and Programming Languages*. MIT Press, 2002.

[3] C. Barton, *How I Wish I'd Taught Maths*. John Catt Educational, 2018.

[4] Lark Parser Documentation, https://github.com/lark-parser/lark. Accessed 2025.

[5] Lean Community, *The Natural Number Game and Tutorial World*. https://leanprover-community.github.io/. Accessed 2025.

[6] Python Software Foundation, *Python 3 Documentation*. https://docs.python.org/3/. Accessed 2025.

[7] D. Harel, "Statecharts: A Visual Formalism for Complex Systems", *Science of Computer Programming*, 8(3):231–274, 1987.

[8] D. Harel and A. Pnueli, "On the Development of Reactive Systems", *Logics and Models of Concurrent Systems*. Springer, 1985.