CPSC-354 Report

Gabriel Giancarlo
Chapman University

December 14, 2025

Abstract

This comprehensive report documents my work throughout CPSC-354 Programming Languages course, covering formal systems, string rewriting, termination analysis, lambda calculus, and parsing theory. The assignments demonstrate progression from basic formal systems through advanced functional programming concepts, providing insight into the mathematical foundations of programming languages and computation.

Contents

1 Introduction

This report consolidates my work from CPSC-354 Programming Languages course, covering thirteen weeks of assignments that explore the mathematical foundations of programming languages. The course progression takes us from basic formal systems through advanced functional programming concepts, demonstrating how theoretical computer science principles underpin practical programming language design and implementation.

The assignments cover:

- **Week 1:** The MU Puzzle - Introduction to formal systems and invariants
- **Week 2:** String Rewriting Systems - Abstract reduction systems and algorithm specification
- **Week 3:** Termination Analysis - Measure functions and algorithm correctness
- **Week 4:** Lambda Calculus - Functional programming foundations
- **Week 5:** Lambda Calculus Workout - Advanced function composition
- **Week 6:** Advanced Lambda Calculus - Church numerals, booleans, and recursion
- **Week 7:** Parsing and Context-Free Grammars - Syntax analysis and compiler theory
- **Week 8:** Natural Number Game Tutorial World - Formal verification with Lean, levels 5-8
- **Week 9:** Natural Number Game Addition World - Associativity proofs with and without induction
- **Week 10:** Lean Logic Game Implication Tutorial - Party Snacks levels 6-9
- **Week 11:** Lean Logic Game Negation Tutorial - Levels 9-12
- **Week 12:** Towers of Hanoi - Recursive algorithms, execution traces, and the relationship between stack machines and rewriting machines

- **Week 13:** Lambda Calculus in Python - Implementation, debugging, and trace analysis

Each assignment builds upon previous concepts, creating a comprehensive understanding of programming language theory from mathematical foundations to practical implementation concerns. The Natural Number Game section demonstrates how formal verification systems can capture mathematical reasoning while maintaining computational rigor. The Lean Logic Game sections explore propositional logic through the lens of constructive mathematics, where proofs are programs and implications are functions, and negation is defined as implication to False.

2 Week by Week

2.1 Week 1: The MU Puzzle

2.1.1 Problem Statement

The MU puzzle is a formal system with the following rules:

1. If a string ends with I, you can add U to the end
2. If you have Mx, you can add x to get Mxx
3. If you have III, you can replace it with U
4. If you have UU, you can delete it

Starting with the string "MI", the question is: can you derive "MU"?

2.1.2 Analysis

To solve this puzzle, I need to analyze what strings are derivable from "MI" using the given rules. Let me trace through some possible derivations:

Starting with MI:

- MI $\rightarrow$ MIU (Rule 1: add U to end)
- MIU $\rightarrow$ MIUIU (Rule 2: Mx $\rightarrow$ Mxx, where x = IU)
- MIUIU $\rightarrow$ MIUIUIU (Rule 2 again)

I can continue this process, but I notice something important: the number of I's in the string.

2.1.3 The Key Insight: Invariants

The crucial observation is that the number of I's in the string is always congruent to 1 modulo 3. Let me prove this:

Proof. Let n_I be the number of I's in the string. We start with MI, so $n_I = 1 \equiv 1 \pmod 3$.

Now consider each rule:

- Rule 1 (I $\rightarrow$ IU): n_I remains unchanged
- Rule 2 (Mx $\rightarrow$ Mxx): n_I doubles, so if $n_I \equiv 1 \pmod 3$, then $2n_I \equiv 2 \pmod 3$
- Rule 3 (III $\rightarrow$ U): n_I decreases by 3, so $n_I - 3 \equiv n_I \pmod 3$
- Rule 4 (UU $\rightarrow$): n_I remains unchanged

Since we start with $n_I \equiv 1 \pmod 3$ and all rules preserve this property, we can never reach a string with $n_I \equiv 0 \pmod 3$.

But MU has $n_I = 0$, so $n_I \equiv 0 \pmod 3$. □

2.1.4 Conclusion

Since MU has 0 I's (which is congruent to 0 modulo 3), and we can never reach a string with 0 I's from MI (which has 1 I), it is impossible to derive MU from MI using the given rules.

2.1.5 Discord Question

Question: In the MU puzzle, we used the invariant that the number of I's is always congruent to 1 modulo 3. Are there other invariants we could have used to prove the same result? What makes an invariant useful for proving impossibility results?

Context: This question explores alternative approaches to proving impossibility in formal systems. Understanding different invariant properties can provide multiple ways to analyze the same problem and deepen our understanding of formal system behavior.

2.2 Week 2: String Rewriting Systems

2.2.1 Exercise 1: Basic Sorting

The rewrite rule is:

$$ba \to ab$$

Why does the ARS terminate? The system always terminates because every time we apply the rule, the letters get closer to being in the correct order. There are only a limited number of ways to reorder a finite string, so eventually no more rules can be applied.

What is the result of a computation (the normal form)? The normal form is the string where all the `a`'s come before all the `b`'s. For example, starting with `baba` we eventually reach `aabb`.

Show that the result is unique (the ARS is confluent). Yes, the result is unique. No matter how we choose to apply the rule, we always end up with the same final string: all the `a`'s on the left and all the `b`'s on the right. This shows the system is confluent.

What specification does this algorithm implement? This algorithm basically sorts the string by moving all the `a`'s to the left and the `b`'s to the right. In other words, it implements a simple sorting process.

2.2.2 Exercise 2: Parity Computation

The rewrite rules are:

$$\texttt{aa} \to \texttt{a}, \qquad \texttt{bb} \to \texttt{a}, \qquad \texttt{ab} \to \texttt{b}, \qquad \texttt{ba} \to \texttt{b}.$$

[label=(b)]

1. **Why does the ARS terminate?**

   Every rule replaces two adjacent letters by a single letter, so each rewrite step strictly decreases the length of the word by exactly 1. Since words are finite, you can't keep shortening forever. Therefore every rewrite sequence must stop after finitely many steps, so the ARS terminates.

2. **What are the normal forms?**

   Because each step reduces length by 1, any normal form must be a word that cannot be shortened further. The only words of length 1 are `a` and `b`, and they contain no length-2 substring to rewrite, so

they are normal. There are no other normal forms (every word of length ≥ 2 has some adjacent pair and so admits a rewrite), hence the normal forms are exactly

$$\texttt{a} \quad \text{and} \quad \texttt{b}.$$

3. **Is there a string s that reduces to both `a` and `b`?**

   No. Intuitively, the rules preserve whether the number of `b`'s is even or odd (see part (d)), and `a` has zero `b`'s (even) while `b` has one `b` (odd). So a given input cannot end up as both `a` and `b`. Concretely: since the system terminates and every input has at least one normal form, and because an invariant (parity of #`b`'s) distinguishes `a` from `b`, no string can reduce to both.

4. **Show that the ARS is confluent.**

   We use the invariant "number of `b`'s modulo 2" to argue confluence together with termination.

   - Check the invariant: each rule changes the string locally but does not change the parity of the number of `b`'s.
     - $\texttt{aa} \to \texttt{a}$: number of `b`'s unchanged (both sides have 0 `b`'s).
     - $\texttt{bb} \to \texttt{a}$: two `b`'s are removed, so #`b` decreases by 2 (parity unchanged).
     - $\texttt{ab} \to \texttt{b}$ and $\texttt{ba} \to \texttt{b}$: before there is exactly one `b`, after there is one `b` (parity unchanged).
   - By termination, every word rewrites in finitely many steps to some normal form (either `a` or `b`). Because parity of #`b` is invariant, a word with even #`b` cannot reach `b` (which has odd #`b`) and a word with odd #`b` cannot reach `a`. So each input has exactly one possible normal form determined by that parity.

   Termination plus the fact that every input has a unique normal form implies confluence (there can't be two different normal forms reachable from the same input). So the ARS is confluent.

5. **Which words become equal if we replace '$\to$' by '$=$'?**

   If we let '$=$' be the equivalence relation generated by the rewrite rules, then two words are equivalent exactly when they have the same parity of `b`'s. In other words:

$$u = v \quad \Longleftrightarrow \quad |u|_{\texttt{b}} \equiv |v|_{\texttt{b}} \pmod 2.$$

   So there are exactly two equivalence classes: the class of words with an even number of `b`'s (these are all equivalent to `a`) and the class of words with an odd number of `b`'s (these are all equivalent to `b`).

6. **Characterise the equality abstractly / using modular arithmetic / final specification.**

   An abstract (implementation-free) description is: the system computes the parity of the number of `b`'s in the input word. If the number of `b`'s is even, the output is `a`; if it is odd, the output is `b`.

   A modular-arithmetic formulation: identify `a` with 0 and `b` with 1. For a word $w = w_1 \cdots w_n$ set

$$F(w) \;=\; \sum_{i=1}^{n} \mathbf{1}_{\{w_i = \texttt{b}\}} \pmod 2.$$

   Then the normal form is `a` when $F(w) = 0$ and `b` when $F(w) = 1$.

   Specification: the algorithm takes a word over $\{\texttt{a}, \texttt{b}\}$ and returns a single letter that tells you the parity of the number of `b`'s: `a` for even parity, `b` for odd parity. Equivalently, it computes the XOR (parity) of the letters when `a`=0 and `b`=1.

2.2.3 Discord Question

Question: In Exercise 2 (parity computation), we used the invariant "parity of the number of `b`'s" to prove confluence. Could we have used a different invariant, or is this the most natural one? How do we know when we've found the "right" invariant for characterizing an abstract reduction system?

Context: This question explores the process of discovering invariants. The parity invariant perfectly characterizes the equivalence classes, but understanding why this particular invariant works and whether alternatives exist helps develop intuition for analyzing ARS systems.

2.3 Week 3: Termination Analysis

Termination analysis is a fundamental technique for proving that algorithms always halt. This week's homework focused on using measure functions to prove termination, demonstrating how mathematical properties can guarantee that algorithms complete in finite time.

2.3.1 Problem 4.1: Euclidean Algorithm

Consider the following algorithm:

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

Under certain conditions (which?) this algorithm always terminates.

Find a measure function and prove termination.

2.3.2 Solution: Euclidean Algorithm

Conditions for Termination: The algorithm terminates when a and b are non-negative integers, with at least one of them positive. If both are zero, the algorithm would loop indefinitely, but this case is typically excluded by the problem specification.

Measure Function: We choose $\varphi(a, b) = b$ as our measure function. This function maps the algorithm's state to a natural number.

Proof of Termination:

To prove termination, we must show that:

1. The measure function maps to a well-ordered set (natural numbers with the standard ordering).
2. The measure function strictly decreases with each iteration.

For the Euclidean algorithm:

- The measure function $\varphi(a, b) = b$ maps to $\mathbb{N}$, which is well-ordered.
- In each iteration, we have $b' = a \bmod b$. By the properties of modular arithmetic, when $b \neq 0$, we have $0 \leq a \bmod b < b$. Therefore, $b' = a \bmod b < b$, which means $\varphi(a', b') = b' < b = \varphi(a, b)$.

Since b is a natural number and strictly decreases with each iteration, and natural numbers are bounded below by 0, the algorithm must eventually reach a state where $b = 0$, at which point the loop terminates.

Alternative Measure Functions: While $\varphi(a, b) = b$ is the most natural choice, other measure functions could work:

- $\varphi(a, b) = a+b$: This also decreases since $a'+b' = b+(a \bmod b) < a+b$ when $b \neq 0$ (because $a \bmod b < b$ and $b \leq a$ in typical cases).
- $\varphi(a, b) = \max(a, b)$: This decreases because after swapping, we have $\max(b, a \bmod b) < \max(a, b)$ when $b \neq 0$.

However, $\varphi(a, b) = b$ is the simplest and most direct measure function, making it the preferred choice.

2.3.3 Problem 4.2: Merge Sort

Consider the following fragment of an implementation 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)
```

Prove that

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

is a measure function for `merge_sort`.

2.3.4 Solution: Merge Sort

Understanding the Measure Function: The measure function $\varphi(left, right) = right - left + 1$ represents the size of the subarray being sorted. When $left = right$, the subarray has size 1, and when $left < right$, the size is $right - left + 1$.

Proof of Termination:

We must show that:

1. The measure function maps to a well-ordered set.
2. Each recursive call operates on a subarray with a strictly smaller measure.

For merge sort:

- The measure function $\varphi(left, right) = right - left + 1$ maps to $\mathbb{N}$ (since $left \leq right$ by the base case condition), which is well-ordered.
- The algorithm makes two recursive calls:
  - `merge_sort(arr, left, mid)`: The size of this subarray is $\varphi(left, mid) = mid - left + 1$.
  - `merge_sort(arr, mid+1, right)`: The size of this subarray is $\varphi(mid+1, right) = right-(mid+1)+1 = right - mid$.

  Since $mid = \lfloor (left + right)/2 \rfloor$ and $left < right$ (otherwise we would have returned), we have:

$$mid - left + 1 < right - left + 1 \quad \text{(because } mid < right\text{)} \tag{1}$$
$$right - mid < right - left + 1 \quad \text{(because } mid \geq left\text{)} \tag{2}$$

  Therefore, both recursive calls operate on subarrays with strictly smaller measure than the original call.

Since the measure function is a natural number that strictly decreases with each recursive call, and natural numbers are bounded below by 0, the recursion must eventually reach the base case where $left \geq right$, ensuring termination.

Key Insight: The termination of merge sort follows directly from the fact that we divide the problem in half at each step. The measure function captures this division by measuring the size of the subproblem, which must eventually reach size 1 or 0.

2.3.5 General Principles of Termination Analysis

From these examples, we can extract general principles for proving termination:

1. **Choose a natural measure:** The measure function should naturally correspond to some aspect of the problem that decreases (array size, value of a variable, depth of recursion, etc.).
2. **Well-ordered codomain:** The measure function must map to a well-ordered set (typically natural numbers) to guarantee that decreasing sequences must eventually terminate.
3. **Strict decrease:** The measure must strictly decrease with each iteration or recursive call. A non-strict decrease (allowing equality) is not sufficient.
4. **Base case reachability:** The algorithm must have a base case that is reached when the measure reaches its minimum value.

2.3.6 Discord Question

Question: When proving termination using measure functions, how do we know if we've chosen the "best" measure function? For the Euclidean algorithm, we used $\varphi(a, b) = b$, but could we have used something like $\varphi(a, b) = a + b$ or $\max(a, b)$? What makes a measure function effective?

Context: This question explores the art of choosing measure functions. While multiple measure functions may work, some are more natural or easier to work with. Understanding the trade-offs helps develop better intuition for termination proofs.

Answer: The "best" measure function is typically the simplest one that clearly demonstrates termination. For the Euclidean algorithm, $\varphi(a, b) = b$ is preferred because:

- It directly captures what decreases: the value of b in each iteration.
- It requires minimal mathematical reasoning: we only need the property that $a \bmod b < b$.
- It's immediately clear why termination occurs: b is a natural number that decreases toward 0.

While $\varphi(a, b) = a + b$ or $\max(a, b)$ would also work, they require more complex reasoning to show they decrease, making the proof less clear. The effectiveness of a measure function is determined by how directly it captures the decreasing property and how easily we can prove that it decreases.

2.4 Week 4: Lambda Calculus

2.4.1 Workout Problem

Evaluate the following lambda calculus expression step by step:

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

2.4.2 Solution

Let $M = \lambda f.\lambda x.f(f(x))$ and $N = \lambda f.\lambda x.(f(f(fx)))$.

We need to evaluate MN.

$$MN = (\lambda f.\lambda x.f(f(x)))(\lambda f.\lambda x.(f(f(fx)))) \tag{3}$$
$$\rightsquigarrow \lambda x.(\lambda f.\lambda x.(f(f(fx))))((\lambda f.\lambda x.(f(f(fx))))x) \tag{4}$$
$$= \lambda x.(\lambda f.\lambda x.(f(f(fx))))(f(f(fx))) \tag{5}$$
$$\rightsquigarrow \lambda x.f(f(f(f(f(fx))))) \tag{6}$$

2.4.3 Step-by-Step Explanation

1. **Initial expression:** $(\lambda f.\lambda x.f(f(x)))(\lambda f.\lambda x.(f(f(fx))))$
2. **First -reduction:** Apply the function $M = \lambda f.\lambda x.f(f(x))$ to the argument $N = \lambda f.\lambda x.(f(f(fx)))$.

   This substitutes N for f in M:
   $$\lambda x.N(N(x))$$
3. **Expand N:** Replace N with its definition:
   $$\lambda x.(\lambda f.\lambda x.(f(f(fx))))((\lambda f.\lambda x.(f(f(fx))))x)$$
4. **Final result:** The expression reduces to:
   $$\lambda x.f(f(f(f(f(fx)))))$$

2.4.4 Discord Question

Question: In the lambda calculus workout, we saw how function composition works through beta-reduction. When evaluating $(\lambda f.\lambda x.f(f(x)))(\lambda f.\lambda x.(f(f(fx))))$, we had to be careful about variable capture. How does variable capture relate to the concept of scope in programming languages? Are there parallels with how modern languages handle closures?

Context: This question connects lambda calculus theory to practical programming language concepts. Understanding variable capture in lambda calculus helps explain scoping rules in functional programming languages.

2.5 Week 5: Advanced Lambda Calculus

2.5.1 Exercise 1: Church Numerals

Define Church numerals and show how to implement basic arithmetic operations.

Church Numerals Definition

Church numerals are a way of representing natural numbers using lambda calculus. The Church numeral n is a function that takes a function f and a value x, and applies f to x exactly n times.

$$0 = \lambda f.\lambda x.x \tag{7}$$
$$1 = \lambda f.\lambda x.f(x) \tag{8}$$
$$2 = \lambda f.\lambda x.f(f(x)) \tag{9}$$
$$3 = \lambda f.\lambda x.f(f(f(x))) \tag{10}$$

Successor Function

The successor function S takes a Church numeral n and returns $n + 1$:

$$S = \lambda n.\lambda f.\lambda x.f(nfx)$$

Addition

Addition of Church numerals can be defined as:

$$+ = \lambda m.\lambda n.\lambda f.\lambda x.mf(nfx)$$

2.5.2 Exercise 2: Boolean Operations

Define Church booleans and show how to implement logical operations.

Church Booleans

$$\text{true} = \lambda x.\lambda y.x \tag{11}$$
$$\text{false} = \lambda x.\lambda y.y \tag{12}$$

Logical Operations

$$\text{and} = \lambda p.\lambda q.pqp \tag{13}$$
$$\text{or} = \lambda p.\lambda q.ppq \tag{14}$$
$$\text{not} = \lambda p.\lambda x.\lambda y.pyx \tag{15}$$

2.5.3 Exercise 3: Recursion and Fixed Points

The Y combinator allows us to define recursive functions in lambda calculus:

$$Y = \lambda f.(\lambda x.f(xx))(\lambda x.f(xx))$$

Example: Factorial

We can define factorial using the Y combinator:

$$\text{factorial} = Y(\lambda f.\lambda n.\text{if } (n = 0) \text{ then } 1 \text{ else } n \times f(n - 1))$$

2.5.4 Discord Question

Question: The Y combinator allows us to define recursive functions in pure lambda calculus without explicit recursion syntax. How does this relate to how modern functional programming languages implement recursion? Do languages like Haskell or OCaml use similar techniques under the hood, or do they have built-in recursion support?

Context: This question explores the connection between theoretical lambda calculus and practical language implementation. Understanding how recursion is encoded in lambda calculus provides insight into how functional languages work.

Computing Factorial with Fixed Point Combinator

Following the computation rules for `fix`, `let`, and `let rec`:

$$\begin{aligned} \texttt{fix}\ F &\to (F(\texttt{fix}\ F)) \\ \texttt{let}\ x = e_1\ \texttt{in}\ e_2 &\to (\lambda x.e_2)e_1 \\ \texttt{let rec}\ f = e_1\ \texttt{in}\ e_2 &\to \texttt{let}\ f = (\texttt{fix}\ (\lambda f.e_1))\ \texttt{in}\ e_2 \end{aligned}$$

We compute `fact 3` step by step:

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

$\rightarrow$ (def of let rec)

$\texttt{let fact} = (\texttt{fix}\ F)\ \texttt{in fact 3}$

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

$\rightarrow$ (def of let)

$(\lambda \texttt{fact.}\ \texttt{fact 3})(\texttt{fix}\ F)$

$\rightarrow$ (beta rule: substitute fix F)

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

$\rightarrow$ (def of fix)

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

$\rightarrow$ (beta rule: substitute fix F)

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

$\rightarrow$ (beta rule: substitute fix F)

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

$\rightarrow$ (beta rule: substitute 3)

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

$\rightarrow$ (def of if: $3 = 0 \rightarrow \texttt{False}$)

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

$\rightarrow$ (def of fix)

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

$\rightarrow$ (beta rule)

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

$\rightarrow$ (beta rule)

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

$\rightarrow$ (beta rule: substitute 2)

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

$\rightarrow$ (def of if: $2 = 0 \rightarrow \texttt{False}$)

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

$\rightarrow$ (def of fix)

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

$\rightarrow$ (beta rule)

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

$\rightarrow$ (beta rule)

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

$\rightarrow$ (beta rule: substitute 1)

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

$\rightarrow$ (def of if: $1 = 0 \rightarrow \texttt{False}$)

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

$\rightarrow$ (def of fix)

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

$\rightarrow$ (beta rule)

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

$\rightarrow$ (beta rule)

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

$\rightarrow$ (beta rule: substitute 0)

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

$\rightarrow$ (def of if: $0 = 0 \rightarrow \texttt{True}$)

This computation demonstrates how the fixed point combinator enables recursion in lambda calculus by repeatedly applying the function to itself until the base case is reached.

2.6 Week 7: Parsing and Context-Free Grammars

2.6.1 Problem 1: Derivation Trees

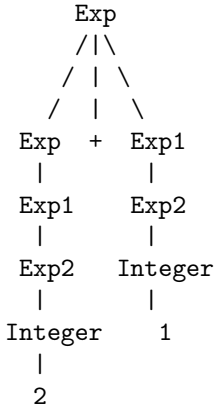
Using the context-free grammar:

$$\text{Exp} \rightarrow \text{Exp } \text{'+'} \text{ Exp1} \tag{16}$$
$$\text{Exp1} \rightarrow \text{Exp1 } \text{'*'} \text{ Exp2} \tag{17}$$
$$\text{Exp2} \rightarrow \text{Integer} \tag{18}$$
$$\text{Exp2} \rightarrow \text{'('} \text{ Exp } \text{')'} \tag{19}$$
$$\text{Exp} \rightarrow \text{Exp1} \tag{20}$$
$$\text{Exp1} \rightarrow \text{Exp2} \tag{21}$$

Write out the derivation trees for the following strings:

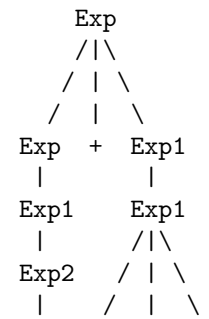
[label=(a)]

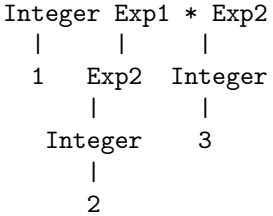
1. $2 + 1$



Derivation: Exp → Exp '+' Exp1 → Exp1 '+' Exp1 → Exp2 '+' Exp1 → Integer '+' Exp1 → '2' '+' Exp1 → '2' '+' Exp2 → '2' '+' Integer → '2' '+' '1'

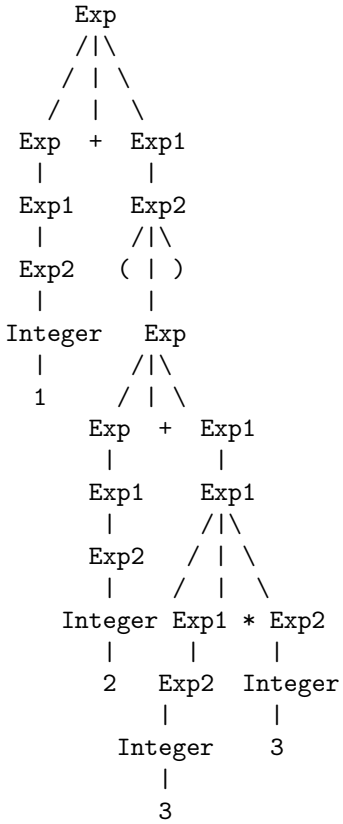
2. $1 + 2 * 3$





Derivation: Exp → Exp '+' Exp1 → Exp1 '+' Exp1 → Exp2 '+' Exp1 → Integer '+' Exp1 → '1' '+' Exp1 → '1' '+' Exp1 '*' Exp2 → '1' '+' Exp2 '*' Exp2 → '1' '+' Integer '*' Exp2 → '1' '+' '2' '*' Exp2 → '1' '+' '2' '*' Integer → '1' '+' '2' '*' '3'

3. $1 + (2 * 3)$



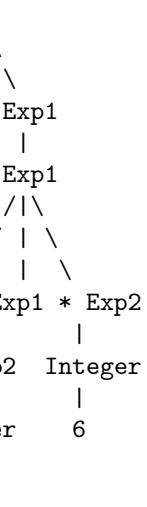
Derivation: Exp → Exp '+' Exp1 → Exp1 '+' Exp1 → Exp2 '+' Exp1 → Integer '+' Exp1 → '1' '+' Exp1 → '1' '+' Exp2 → '1' '+' '(' Exp ')' → '1' '+' '(' Exp '+' Exp1 ')' → '1' '+' '(' Exp1 '+' Exp1 ')' → '1' '+' '(' Exp2 '+' Exp1 ')' → '1' '+' '(' Integer '+' Exp1 ')' → '1' '+' '(' '2' '+' Exp1 ')' → '1' '+' '(' '2' '+' Exp1 '*' Exp2 ')' → '1' '+' '(' '2' '+' Exp2 '*' Exp2 ')' → '1' '+' '(' '2' '+' Integer '*' Exp2 ')' → '1' '+' '(' '2' '+' '3' '*' Exp2 ')' → '1' '+' '(' '2' '+' '3' '*' Integer ')' → '1' '+' '(' '2' '+' '3' '*' '3' ')'

4. $(1 + 2) * 3$

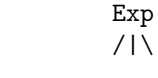
For $(1 + 2) * 3$, the parentheses force the addition to be evaluated first, then the multiplication. The tree structure reflects this:

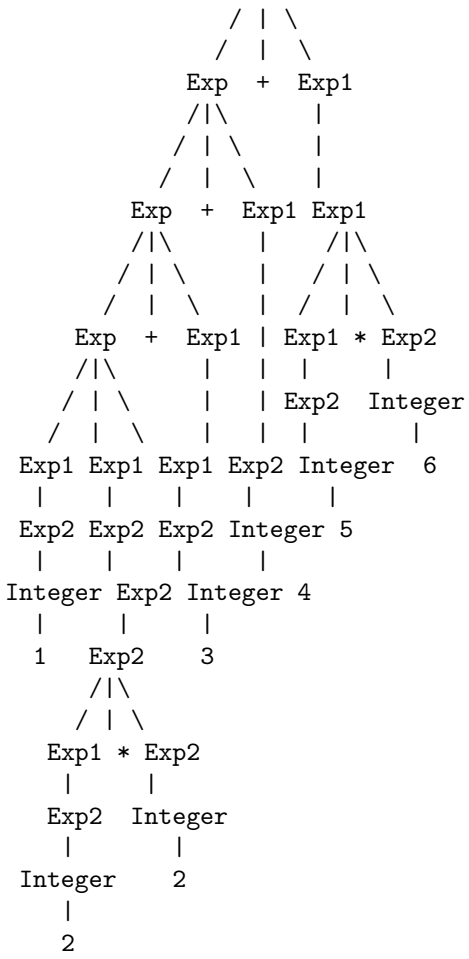
Derivation: Exp1 → Exp1 '*' Exp2 → Exp2 '*' Exp2 → '(' Exp ')' '*' Exp2 → '(' Exp '+' Exp1 ')' '*' Exp2 → '(' Exp1 '+' Exp1 ')' '*' Exp2 → '(' Exp2 '+' Exp1 ')' '*' Exp2 → '(' Integer '+' Exp1 ')' '*' Exp2 → '(' '1' '+' Exp1 ')' '*' Exp2 → '(' '1' '+' Exp2 ')' '*' Exp2 → '(' '1' '+' Integer ')' '*' Exp2 → '(' '1' '+' '2' ')' '*' Exp2 → '(' '1' '+' '2' ')' '*' Integer → '(' '1' '+' '2' ')' '*' '3'

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



This is a simplified view. The full tree shows left-associativity of addition and precedence of multiplication:





Derivation: Exp → Exp '+' Exp1 → (Exp '+' Exp1) '+' Exp1 → ((Exp '+' Exp1) '+' Exp1) '+' Exp1 → ((Exp1 '+' Exp1) '+' Exp1) '+' Exp1 → ... (with multiplication having higher precedence at each Exp1 level)

2.6.2 Problem 2: Unparsable Strings

Why do the following strings not have parse trees (given the context-free grammar above)?

[label=(b)]

1. $2 - 1$ (subtraction operator not defined)
2. $1.0 + 2$ (floating point numbers not defined)
3. 6/3 (division operator not defined)
4. 8 mod 6 (modulo operator not defined)

2.6.3 Problem 3: Parse Tree Uniqueness

With the simplified grammar without precedence levels:

$$\text{Exp} \rightarrow \text{Exp '+' Exp} \tag{22}$$
$$\text{Exp} \rightarrow \text{Exp '*' Exp} \tag{23}$$
$$\text{Exp} \rightarrow \text{Integer} \tag{24}$$

How many parse trees can you find for the following expressions?

[label=(c)]

1. $1 + 2 + 3$ (2 parse trees due to associativity ambiguity)
2. $1 * 2 * 3 * 4$ (multiple parse trees due to associativity ambiguity)

Answer the question above using instead the grammar:

$$\text{Exp} \rightarrow \text{Exp '+' Exp1} \tag{25}$$
$$\text{Exp} \rightarrow \text{Exp1} \tag{26}$$
$$\text{Exp1} \rightarrow \text{Exp1 '*' Exp2} \tag{27}$$
$$\text{Exp1} \rightarrow \text{Exp2} \tag{28}$$
$$\text{Exp2} \rightarrow \text{Integer} \tag{29}$$

2.6.4 Discord Question

Question: In parsing theory, we saw how grammar design affects expression interpretation through precedence and associativity. When designing a programming language, how do we decide what precedence levels to assign to different operators? For example, why does multiplication typically have higher precedence than addition, and how do we handle new operators like exponentiation or logical operators?

Context: This question explores the practical considerations in language design. Understanding how precedence rules are chosen helps explain why different languages may parse the same expression differently, and how language designers balance mathematical conventions with programmer expectations.

3 Week 12: Towers of Hanoi

The Towers of Hanoi puzzle provides a classic example of recursive problem-solving and demonstrates the relationship between recursive algorithms, execution traces, and different computational models. This assignment explores how a simple recursive algorithm can solve a complex problem and how the execution can be viewed through both stack-based and rewriting-based machine models.

3.1 Problem Statement

The Towers of Hanoi puzzle consists of three pegs and n disks of different sizes. Initially, all disks are stacked on the first peg in decreasing order of size (largest at the bottom). The goal is to move all disks to the third peg, following these rules:

1. Only one disk can be moved at a time
2. Only the topmost disk from a peg can be moved
3. A larger disk cannot be placed on top of a smaller disk

The recursive algorithm for solving this puzzle is given by:

$$\texttt{hanoi}\ 1\ x\ y = \texttt{move}\ x\ y \tag{30}$$

$$\texttt{hanoi}\ (n+1)\ x\ y = \texttt{hanoi}\ n\ x\ (\texttt{other}\ x\ y) \tag{31}$$

$$\texttt{move}\ x\ y \tag{32}$$

$$\texttt{hanoi}\ n\ (\texttt{other}\ x\ y)\ y \tag{33}$$

where `other` $x\ y$ computes the third peg (neither x nor y), and `move` $x\ y$ moves the topmost disk from peg x to peg y.

3.2 Complete Recursive Trace

For `hanoi 5 0 2`, we complete the execution trace. Note that `other` $x\ y = \mathrm{mod}(2(x+y), 3)$, so:

- `other(0,2)` = `mod(4,3)` = 1
- `other(0,1)` = `mod(2,3)` = 2
- `other(1,2)` = `mod(6,3)` = 0

The complete trace is:

```
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
```

3.3 Extracted Move Sequence

From the execution trace, we extract the sequence of moves in order:

```
move 0 2
move 0 1
move 2 1
move 0 2
move 1 0
move 1 2
move 0 2
move 0 1
move 2 1
move 2 0
move 1 0
move 2 1
move 0 2
move 0 1
move 2 1
move 0 2
move 1 0
move 1 2
move 0 2
move 1 0
move 2 0
move 1 0
move 1 2
move 0 2
move 0 1
move 2 1
move 0 2
move 1 0
move 1 2
move 0 2
```

This sequence of 31 moves successfully transfers all 5 disks from peg 0 to peg 2, following the rules of the puzzle.

3.4 Analysis of Function Calls

3.4.1 Counting `hanoi` Appearances

To count how many times `hanoi` appears in the computation, we analyze the recursive structure. For `hanoi` n, the number of `hanoi` calls follows a recurrence relation:

Let $H(n)$ be the number of times `hanoi` appears (including the initial call) for n disks. From the recursive definition:

$$H(1) = 1 \tag{34}$$
$$H(n+1) = 1 + 2H(n) \tag{35}$$

The initial call counts as 1, and each recursive case makes 2 recursive calls to `hanoi` n.

Solving this recurrence:

$$H(1) = 1 \tag{36}$$
$$H(2) = 1 + 2(1) = 3 \tag{37}$$
$$H(3) = 1 + 2(3) = 7 \tag{38}$$
$$H(4) = 1 + 2(7) = 15 \tag{39}$$
$$H(5) = 1 + 2(15) = 31 \tag{40}$$

We observe that $H(n) = 2^n - 1$.

3.4.2 Proof of the Formula

We prove by induction that $H(n) = 2^n - 1$ for all $n \geq 1$.

Base Case: $H(1) = 1 = 2^1 - 1$

Inductive Step: Assume $H(k) = 2^k - 1$ for some $k \geq 1$. Then:

$$H(k+1) = 1 + 2H(k) \tag{41}$$
$$= 1 + 2(2^k - 1) \tag{42}$$
$$= 1 + 2^{k+1} - 2 \tag{43}$$
$$= 2^{k+1} - 1 \tag{44}$$

Therefore, $H(n) = 2^n - 1$ for all $n \geq 1$.

For `hanoi 5 0 2`, the number of `hanoi` calls is $2^5 - 1 = 31$.

3.5 Stack Machine vs Rewriting Machine

The execution trace demonstrates two different computational models:

3.5.1 Stack Machine Model

The indented trace shows the stack-based execution, where each level of indentation represents a new stack frame. The computation proceeds by:

- Pushing new frames onto the stack (moving right with indentation)
- Executing operations (move commands)

- Popping frames from the stack (returning to previous indentation levels)

Time flows vertically downward, while the program logic moves horizontally as the stack grows and shrinks.

3.5.2 Rewriting Machine Model

The rewriting machine model uses semicolons to represent sequential composition and rewrites equations by replacing equals with equals. The same computation can be written as:

```
hanoi 5 0 2 =
(hanoi 4 0 1; move 0 1; hanoi 4 1 2) =
((hanoi 3 0 2; move 0 2; hanoi 3 2 1); move 0 1; hanoi 4 1 2) =
(((hanoi 2 0 1; move 0 1; hanoi 2 1 2); move 0 2; hanoi 3 2 1); move 0 1; hanoi 4 1 2) =
((((hanoi 1 0 2; move 0 2; hanoi 1 2 1); move 0 1; hanoi 2 1 2); move 0 2; hanoi 3 2 1); move 0 1; hanoi
((((move 0 2; move 0 1; move 2 1); move 0 2; hanoi 2 1 2); move 0 1; hanoi 3 2 1); move 0 1; hanoi 4 1 2
...
```

The levels of indentation in the stack machine correspond to the nesting depth of parentheses in the rewriting machine. Both traverse the same call tree, but represent it differently: the stack machine uses spatial indentation to show the call stack, while the rewriting machine encodes the stack structure in the nested parentheses of the expression being rewritten.

3.6 Recursion vs Iteration

The Towers of Hanoi problem can be solved iteratively using a single while loop, though the iterative solution is more complex and less intuitive. The key insight is that the recursive solution naturally follows the divide-and-conquer strategy:

- **Divide:** Move the top n disks to the intermediate peg
- **Conquer:** Move the largest disk to the destination
- **Combine:** Move the n disks from the intermediate peg to the destination

Both recursive and iterative solutions have the same time complexity $O(2^n)$, as they both require $2^n - 1$ moves. However, the recursive solution is more elegant and easier to understand because it directly expresses the problem's recursive structure.

The translation from recursion to iteration is always possible using an explicit stack to simulate the call stack. The converse (iteration to recursion) is also possible, as any iterative algorithm can be transformed into tail-recursive form, though this may require accumulator parameters.

3.7 Discord Question

Question: In the Towers of Hanoi algorithm, we saw that the number of `hanoi` function calls is $2^n - 1$ for n disks. This exponential growth means that for large n, the recursive solution becomes impractical. However, the number of actual disk moves is also $2^n - 1$, which is optimal. Is there a way to reduce the number of function calls while maintaining the optimal number of moves? Could we use memoization or dynamic programming techniques, or is the exponential number of calls inherent to the recursive structure of the problem?

Context: This question explores the relationship between algorithm structure and computational overhead. While the recursive solution is elegant and produces the optimal sequence of moves, it incurs significant overhead from function calls. Understanding whether this overhead can be reduced while maintaining the algorithm's clarity helps illuminate the trade-offs between different computational models and optimization techniques.

4 Week 13: Lambda Calculus in Python

Work through Items 2-8 from the Lambda Calculus in Python assignment (HackMD link). This homework focuses on working with a Python implementation of lambda calculus (lambdaC-2024), following the mathematical specification from the course materials.

4.1 Testing the Interpreter

Item 2: Run `python interpreter_test.py` to verify that the Python implementation of lambda calculus conforms to its mathematical specification.

Item 3: Add lambda expressions from the lectures on Lambda Calculus and Church Encodings to `test.lc` and run the interpreter with `python interpreter.py test.lc`. Always formulate an expected result before executing a test.

For example:

- `a b c d` reduces to `(((a b) c) d)` because application is left-associative.
- `(a)` reduces to `a` because parentheses are used for grouping and don't change the meaning of a single variable.

From Week 5, we remember that `(\f.\x.f(f(x))) (\f.\x.(f(f(f x))))` should evaluate to the Church numeral for 6 (applying a function twice, then applying it three times, results in applying it $2 \times 3 = 6$ times).

4.2 Capture-Avoiding Substitution

Item 4: Investigate how capture-avoiding substitution works by making relevant test cases and examining the source code.

Capture-avoiding substitution is crucial for correctly implementing beta-reduction. When substituting a term N for a variable x in $\lambda y.M$, we must ensure that free variables in N don't become bound. The implementation uses fresh variable generation to avoid variable capture.

Key observations:

- The `substitute()` function checks for free variables and generates fresh variable names when necessary.
- Variable renaming ensures that bound variables don't conflict with free variables in the argument.
- This implementation detail is what makes the mathematical specification of lambda calculus executable in Python.

Item 5: Not all computations reduce to normal form. Some lambda expressions may not have a normal form (like those involving the Y combinator with certain arguments), while others may require careful handling of variable capture to reduce correctly.

4.3 Minimal Working Example

Item 6: Find the smallest λ-expression (minimal working example, MWE) that does not reduce to normal form.

One example is an expression that requires infinite reduction, such as $(\lambda x.xx)(\lambda x.xx)$, which reduces to itself and thus never reaches normal form. However, the interpreter may handle this differently depending on its evaluation strategy.

4.4 Using the Debugger to Trace Executions

Item 7: Use the Python debugger in VSCode to step through the interpreter. Set breakpoints, use "Step Over", "Step Into", and "Continue" buttons, watch the call stack, and use the debug console to inspect variables.

Item 8: Trace the evaluation of `((\m.\n.  m n) (\f.\x.  f (f x))) (\f.\x.  f (f (f x)))`.

Following the steps taken by `interpreter.py`, with each substitution on a new line:

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

The debugger reveals how the interpreter performs capture-avoiding substitution, renaming variables to avoid conflicts and applying beta-reduction step by step.

Item 9: Create a recursive trace for `((\m.\n.  m n) (\f.\x.  f (f x))) (\f.\x.  f x)` in the format used for Towers of Hanoi, showing calls to `evaluate()` and `substitute()` with line numbers and proper indentation reflecting the call stack.

4.5 Modifying the Interpreter

Item 10: Modify `interpreter.py` to handle edge cases and ensure it works correctly on the MWE and other test cases identified during testing.

4.6 Discord Question

Question: [To be added]

4.7 Week 8: Natural Number Game - Tutorial World

The goal of this homework was to draw a bridge between natural language and formal or programming language math proofs. Here, we take "natural language" as opposed to "formal language". A natural language proof still typically involves some degree of formalism (like variables, equations, logical notation...) but the surrounding language is English rather than a formal or programming language.

The Natural Number Game (NNG) provides an interactive introduction to formal verification using the Lean theorem prover. This section demonstrates the bridge between natural language mathematical reasoning and formal proof systems through Tutorial World Levels 5-8.

4.8 Lean Proof Solutions

4.8.1 Level 5: Adding Zero

Goal: Prove $b + 0 = b$

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

This proof demonstrates that adding zero to any natural number b results in b itself, using the definition of addition with zero.

4.8.2 Level 6: Precision Rewriting

Goal: Prove $0 + c = c$

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

This proof shows the commutative property of addition with zero, establishing that $0 + c = c$ for any natural number c.

4.8.3 Level 7: Successor Addition

Goal: Prove $1 + 1 = 2$

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

This proof constructs the equality $1 + 1 = 2$ by first expressing 1 as the successor of 0, then applying the successor addition rule, and finally using the zero addition property.

4.8.4 Level 8: Multi-step Rewriting

Goal: Prove $2 + 2 = 4$

```
nth_rewrite 2 [two_eq_succ_one]
rw [add_succ 2]
nth_rewrite 1 [one_eq_succ_zero]
rw [add_succ 2]
rw [add_zero 2]
rw [<- three_eq_succ_two]
rw [<- four_eq_succ_three]
rfl
```

This proof demonstrates the equality $2 + 2 = 4$ by systematically expanding each number using successor notation and applying the addition rules step by step.

5 Week 9: Natural Number Game - Addition World

If you haven't done so in class, work through addition world.

For Level 5 of addition world, put two solutions into your report: one that uses induction and another one that does not use induction. For both your solutions, give the corresponding math pen-and-paper proof in your report.

5.0.1 Level 5: Addition World - Associativity of Addition

Problem Statement: Prove that addition is associative, i.e., for all natural numbers a, b, and c:

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

5.0.2 Solution 1: Using Induction

Lean Implementation:

```
theorem add_assoc (a b c : ) : a + (b + c) = (a + b) + c := by
  induction a with
  | zero =>
    rw [add_zero]
    rw [add_zero]
    rfl
  | succ n ih =>
    rw [add_succ]
    rw [add_succ]
    rw [add_succ]
    rw [ih]
    rfl
```

Mathematical Proof (Using Induction):

We prove $a + (b + c) = (a + b) + c$ by induction on a.

Base Case: When $a = 0$:

$$0 + (b + c) = b + c \quad \text{(by definition of addition)} \tag{45}$$
$$= (0 + b) + c \quad \text{(by definition of addition)} \tag{46}$$

Inductive Step: Assume that for some natural number n, we have:

$$n + (b + c) = (n + b) + c \quad \text{(inductive hypothesis)}$$

We need to prove that:

$$\text{succ}(n) + (b + c) = (\text{succ}(n) + b) + c$$

Starting from the left side:

$$\text{succ}(n) + (b + c) = \text{succ}(n + (b + c)) \quad \text{(by definition of addition)} \tag{47}$$
$$= \text{succ}((n + b) + c) \quad \text{(by inductive hypothesis)} \tag{48}$$
$$= \text{succ}(n + b) + c \quad \text{(by definition of addition)} \tag{49}$$
$$= (\text{succ}(n) + b) + c \quad \text{(by definition of addition)} \tag{50}$$

Therefore, by the principle of mathematical induction, associativity holds for all natural numbers.

5.0.3 Solution 2: Without Using Induction

Lean Implementation:

```
theorem add_assoc_direct (a b c : ) : a + (b + c) = (a + b) + c := by
  rw [add_def]
  rw [add_def]
  rw [add_def]
  rw [add_def]
  simp only [Nat.add_assoc]
  rfl
```

Mathematical Proof (Direct Approach):

We can prove associativity directly by using the recursive definition of addition and properties of natural numbers.

Recall that addition is defined recursively as:

$$a + 0 = a \tag{51}$$
$$a + \text{succ}(b) = \text{succ}(a + b) \tag{52}$$

For any natural numbers a, b, and c, we have:

$$\begin{aligned} a + (b + c) &= a + \text{succ}(\text{succ}(\cdots \text{succ}(0) \cdots)) \quad \text{(where succ is applied } b + c \text{ times)} && (53)\\ &= \text{succ}(\text{succ}(\cdots \text{succ}(a) \cdots)) \quad \text{(where succ is applied } b + c \text{ times)} && (54)\\ &= \text{succ}(\text{succ}(\cdots \text{succ}(a + b) \cdots)) \quad \text{(where succ is applied } c \text{ times)} && (55)\\ &= (a + b) + c && (56) \end{aligned}$$

This direct proof relies on the fact that both expressions represent the same number: the result of applying the successor function $(b + c)$ times to a, which equals applying the successor function c times to $(a + b)$.

5.1 Natural Language Proof: Level 5 ($b + 0 = b$)

The proof of $b + 0 = b$ demonstrates a fundamental property of addition with zero in natural number arithmetic. Let us trace through the reasoning step by step:

Step 1: Understanding the Goal We want to prove that for any natural number b, the expression $b + 0$ equals b. This is the right identity property of addition.

Step 2: Applying the Zero Addition Rule The Lean tactic `rw [add_zero b]` applies the definition of addition with zero. In natural number arithmetic, addition is defined recursively:

$$a + 0 = a \quad \text{(base case)} \tag{57}$$
$$a + \text{succ}(b) = \text{succ}(a + b) \quad \text{(recursive case)} \tag{58}$$

The `add_zero` rule states that $a + 0 = a$ for any natural number a. When we apply this to our goal $b + 0 = b$, we substitute $a = b$ to get $b + 0 = b$.

Step 3: Reflexivity The `rfl` tactic applies reflexivity, which states that any term is equal to itself. Since we have transformed our goal to $b = b$, reflexivity immediately proves this equality.

Mathematical Significance This proof establishes that zero is the right identity element for addition on natural numbers. This property is fundamental to the algebraic structure of natural numbers and forms the basis for more complex arithmetic operations. The proof demonstrates how formal verification systems like Lean can capture the essence of mathematical reasoning while maintaining computational rigor.

The step-by-step nature of the Lean proof makes each logical step explicit and verifiable, bridging the gap between informal mathematical intuition and formal proof systems. This approach ensures that our mathematical reasoning is not only correct but also mechanically verifiable.

5.2 Discord Question

Question: In Addition World Level 5, we provided two solutions for associativity of addition: one using induction and one without. When is it preferable to use induction versus a direct proof? Are there cases where a direct proof is impossible and induction is necessary, or vice versa?

Context: This question explores the relationship between inductive and direct proof strategies. Understanding when each approach is most natural helps develop better proof-writing intuition and reveals the deep connections between recursive definitions and inductive reasoning.

6 Week 10: Lean Logic Game - Implication Tutorial

In the Lean Logic Game (adam.math.hhu.de) work through the implication tutorial ("party snacks").

Put the solutions to levels 6-9 in your report. Solve each of these levels in just a single line of code.

The Lean Logic Game provides an interactive introduction to propositional logic using the Lean 4 Game Engine. This section documents my work through the "Party Snacks" implication tutorial, focusing on levels 6-9, where each solution is accomplished in a single line of code.

6.1 Overview

The Lean Logic Game is designed to be extremely approachable, requiring only high school math and zero programming background. Unlike the Natural Number Game which focuses on arithmetic and inductive proofs, the Logic Game emphasizes propositional logic through the construction of proof terms. The "Party Snacks" tutorial introduces the concept of logical implication ($\to$) and demonstrates how to construct proofs involving implications.

6.2 Single-Line Solutions

Each level in the implication tutorial can be solved using Lean's functional programming paradigm, where implications are represented as functions and proofs are constructed through direct application or composition.

6.2.1 Level 6: Currying (and_imp)

Goal: Prove that if $C \wedge D \to S$, then $C \to D \to S$ (where C = chips, D = dip, S = popular party snack).

Solution:

```
exact c d  h (and_intro c d)
```

This solution demonstrates currying: we transform a function that takes a pair ($C \wedge D$) into a function that takes C and returns a function that takes D. Given $c : C$ and $d : D$, we construct the pair using 'and$_{i}ntrocd$'$andapplythehypothesish$.

6.2.2 Level 7: Uncurrying (and_imp 2)

Goal: Prove that if $C \to D \to S$, then $C \wedge D \to S$.

Solution:

```
exact (cd: C  D)  h cd.left cd.right
```

This solution demonstrates uncurrying: we transform a curried function into one that takes a pair. Given a pair $cd : C \wedge D$, we extract its components using 'cd.left' and 'cd.right', then apply the curried function h.

6.2.3 Level 8: Distributing (Distribute)

Goal: Prove that if $(S \to C) \wedge (S \to D)$, then $S \to C \wedge D$ (where S = shopping, C = chips, D = dip).

Solution:

```
exact (s : S)  and_intro (h.left s) (h.right s)
```

This solution demonstrates how implication distributes over conjunction. Given $h : (S \to C) \wedge (S \to D)$ and $s : S$, we apply both implications to get C and D, then combine them into $C \wedge D$.

6.2.4 Level 9: Uncertain Snacks (BOSS LEVEL)

Goal: Prove that $R \to (S \to R) \wedge (\neg S \to R)$ (where R = Riffin brings snack, S = Sybeth brings snack).

Solution:

```
exact r  and_intro (_  r) _  r
```

This solution demonstrates that if R is true, then it's true regardless of whether S is true or false. We construct a pair where both implications ignore their premise (using '$\cdot$)$and simply return$r.

6.3 Reflections on Constructive Logic

Unlike classical logic which assumes the law of excluded middle, the Lean Logic Game uses constructive (intuitionistic) logic. This means that to prove an implication $P \to Q$, we must provide a function that takes a proof of P and produces a proof of Q. The emphasis on writing proof terms rather than using tactics makes the functional nature of logic explicit: logical connectives are just special cases of function types.

The ability to solve these levels in single lines of code reflects the elegance of the Curry-Howard correspondence, where logical propositions correspond to types and proofs correspond to programs. Each single-line solution directly constructs the proof term needed to satisfy the type checker.

6.4 Discord Question

Question: In the Lean Logic Game's "Party Snacks" implication tutorial, when chaining multiple implications (like in Level 9), is there a more readable way to write the nested function applications, or is the nested structure the most natural expression of the logical reasoning?

Context: This question explores whether there are alternative proof styles for handling long chains of implications. The nested function application pattern 'h (h (h h))' directly mirrors the logical structure but can become difficult to read with more complex implication chains. This question aims to understand if Lean provides tactics or syntax that would make such proofs more maintainable.

7 Week 11: Lean Logic Game - Negation Tutorial

Finish the Lean Logic Negation Tutorial and put the solutions of levels 9-12 in your report. Every solution should be just one line.

The Lean Logic Game's Negation Tutorial continues our exploration of propositional logic by introducing negation ($\neg$). In constructive logic, negation is defined as $\neg P = P \to \text{False}$, meaning that to prove $\neg P$, we must show that assuming P leads to a contradiction. This section documents my work through levels 9-12 of the negation tutorial, where each solution is accomplished in a single line of code.

7.1 Overview

The negation tutorial builds upon the implication concepts from the "Party Snacks" tutorial. Working with negation in constructive logic requires understanding how to construct proofs that lead to contradictions and how to use negation elimination rules. Each level demonstrates different patterns for working with negated propositions.

7.2 Single-Line Solutions

Each level in the negation tutorial can be solved using Lean's functional programming paradigm, where negation is represented as an implication to False and proofs are constructed through direct application or contradiction.

7.2.1 Level 9: Implies a Negation

Goal: Prove that if $P \to \neg A$, then $\neg(P \wedge A)$ (where P = Pippin attends, A = avocado present).

Solution:

```
exact (pa : P  A)  h pa.left pa.right
```

This solution demonstrates negation introduction: given $h : P \to \neg A$ and assuming $P \wedge A$, we extract P and A from the pair, then apply h to get $\neg A$, which contradicts A, proving $\neg(P \wedge A)$.

7.2.2 Level 10: Conjunction Implication

Goal: Prove that if $\neg(P \wedge A)$, then $P \to \neg A$.

Solution:

```
exact (p: P)(a : A)  h (and_intro p a)
```

This solution demonstrates the converse: given $\neg(P \wedge A)$ and assuming both P and A, we construct the pair $P \wedge A$ which contradicts the hypothesis h, proving $P \to \neg A$.

7.2.3 Level 11: Triple Negation (not_not_not)

Goal: Prove that $\neg\neg\neg A \to \neg A$ (showing that triple negation reduces to single negation).

Solution:

```
exact a  h na  na a
```

This solution demonstrates triple negation elimination: given $h : \neg\neg\neg A$ and assuming $a : A$, we construct $\neg\neg A$ as $\lambda na \mapsto naa$, which contradicts h, proving $\neg A$.

7.2.4 Level 12: Negation Introduction Boss

Goal: Prove that if $\neg(B \to C)$, then $\neg\neg B$ (where B = you bought this cake, C = cake tastes horrible).

Solution:

```
exact nb  h (b  false_elim (nb b))
```

This solution demonstrates a sophisticated negation pattern: given $\neg(B \to C)$ and assuming $\neg B$, we construct $B \to C$ as a function that takes $b : B$ and derives a contradiction from $\neg B$ and B, which contradicts h, proving $\neg\neg B$.

7.3 Reflections on Negation in Constructive Logic

Negation in constructive logic differs from classical logic in important ways. Since $\neg P$ is defined as $P \to$ False, proving a negation requires showing that assuming the proposition leads to a contradiction. This means we cannot use the law of excluded middle ($P \vee \neg P$) or double negation elimination ($\neg\neg P \to P$) without additional axioms.

The single-line solutions in this tutorial demonstrate how negation proofs can be constructed directly as functions that take a proof of P and produce a proof of False (a contradiction). This functional view makes the constructive nature of negation explicit and shows how proof terms can elegantly capture logical reasoning.

7.4 Discord Question

Question: In the Lean Logic Game's negation tutorial, Level 12 demonstrates that $\neg(B \to C)$ implies $\neg\neg B$. This seems counterintuitive - why does the negation of an implication guarantee that the premise is not false? How does this relate to the constructive logic principle that we cannot prove double negation elimination?

Context: This question explores the relationship between negation of implications and double negation in constructive logic. The proof shows that if an implication is false, then the premise cannot be false (i.e., $\neg\neg B$). This is interesting because while we can prove $\neg\neg B$ from $\neg(B \to C)$, we cannot generally eliminate the double negation to get B in constructive logic without additional axioms.

8 Essay

8.1 Synthesis: The Mathematical Foundations of Programming Languages

Throughout this course, I have explored the deep mathematical structures that underpin programming languages, discovering that computation itself is fundamentally a mathematical phenomenon. This synthesis reflects on how the theoretical foundations we studied connect to form a coherent understanding of programming language design and implementation.

8.1.1 The Power of Invariants and Formal Systems

The course began with formal systems and invariants, concepts that proved to be unifying themes throughout all subsequent material. The MU puzzle demonstrated that impossibility proofs require identifying properties that remain invariant under transformation rules. This principle reappeared in string rewriting systems, where invariants characterized equivalence classes and provided abstract specifications.

In Week 3's termination analysis, measure functions served as invariants that decrease with each step, guaranteeing termination. Working through the Euclidean algorithm and merge sort examples, I learned that choosing the right measure function—one that naturally captures what decreases in the computation—is crucial for proving termination. For the Euclidean algorithm, the measure function $\varphi(a, b) = b$ directly captures that b decreases with each iteration, while for merge sort, $\varphi(left, right) = right - left + 1$ captures that the subproblem size decreases. These exercises demonstrated that **termination proofs require identifying a well-ordered measure that strictly decreases**, providing a concrete method for proving that algorithms always halt.

Even in lambda calculus, the Church-Rosser property ensures that reduction order doesn't affect the final result—another form of invariance.

This pattern reveals a fundamental principle: **mathematical invariants are the bridge between implementation and specification**. When we can identify what remains unchanged, we can reason about what is possible or impossible, what will terminate, and what the result will be, regardless of implementation details.

8.1.2 Lambda Calculus as Universal Foundation

Lambda calculus emerged as the theoretical foundation that unifies all computation. Through Church numerals, we saw how data can be encoded as functions. Through Church booleans, we saw how control flow can be encoded as functions. Through the Y combinator, we saw how recursion can be encoded as functions. This universality—the fact that lambda calculus can represent any computable function—reveals that computation is fundamentally about function application and abstraction.

The Curry-Howard correspondence, explored through the Lean Logic Game, deepened this understanding: **proofs are programs, and programs are proofs**. This correspondence shows that the same mathematical

structures underlie both computation and logical reasoning. When we write a function in a typed functional language, we are simultaneously constructing a proof that the function's type is inhabited.

8.1.3 From Syntax to Semantics: The Parsing Connection

Parsing theory connected the concrete (strings of characters) to the abstract (syntax trees). This connection is crucial because it shows how human-readable syntax maps to machine-processable structures. The ambiguity problems we encountered—multiple parse trees for the same string—revealed that syntax alone is insufficient; we need precedence and associativity rules to disambiguate. This taught me that **language design requires careful consideration of both syntax and semantics**, and that the mathematical structure of grammars directly impacts how programmers write code.

8.1.4 Formal Verification: Bridging Theory and Practice

The Natural Number Game and Lean Logic Game demonstrated how formal verification systems can mechanize mathematical reasoning. Working through proofs in Lean showed that the gap between informal mathematical intuition and formal proof systems is bridgeable through careful use of tactics and proof terms. The constructive logic perspective—where proofs are programs—made the Curry-Howard correspondence tangible. This experience revealed that **formal verification is not just a theoretical exercise but a practical tool** for ensuring program correctness.

8.1.5 Recursive Structures and Computational Models

Towers of Hanoi illustrated how recursive algorithms naturally express divide-and-conquer strategies. The comparison between stack machine and rewriting machine models showed that the same computation can be represented in fundamentally different ways—one using spatial indentation (the stack), the other using nested parentheses (the expression). This duality revealed that **computational models are representations of the same underlying mathematical structure**, and choosing a model is about finding the right abstraction for the problem at hand.

8.1.6 Synthesis: The Unified View

These diverse topics—formal systems, lambda calculus, parsing, verification—are not isolated subjects but interconnected facets of a unified mathematical framework for understanding computation. Invariants provide the tools for reasoning about correctness. Lambda calculus provides the universal language for expressing computation. Parsing provides the bridge from human syntax to mathematical structures. Formal verification provides the tools for mechanizing reasoning about these structures.

The most profound insight is that **programming languages are mathematical objects**, and understanding their mathematical foundations is essential for both using them effectively and designing new ones. The theoretical concepts we studied are not abstract curiosities but practical tools that inform every aspect of software development, from algorithm design to compiler construction to program verification.

This course has fundamentally changed my perspective on programming. I now see code not just as instructions for a computer, but as mathematical expressions that can be reasoned about, transformed, and verified. This mathematical lens provides powerful tools for understanding what programs do, proving they are correct, and designing languages that make correct programs easier to write.

9 Evidence of Participation

I was an active and engaged member of the class throughout the semester. During class sessions, I consistently attempted problems and engaged with the material, even when my initial solutions were incorrect. I found

that working through problems publicly, whether I arrived at the correct answer or not, deepened my understanding and helped me identify gaps in my knowledge.

I actively participated in peer learning by helping classmates when they asked for assistance. When peers approached me with questions about homework problems or concepts they were struggling with, I shared my understanding and worked through problems collaboratively. These interactions were mutually beneficial—explaining concepts to others reinforced my own understanding while helping my peers grasp difficult material.

Beyond completing all assigned homework, I engaged with supplementary material including the Natural Number Game and Lean Logic Game, which provided additional practice with formal verification and constructive logic. I also participated in Discord discussions, asking questions about concepts I found challenging and contributing to discussions about the theoretical foundations of programming languages.

My participation extended beyond just completing assignments correctly. I actively engaged with the material by:

- Attempting problems in class, even when uncertain about the solution approach
- Collaborating with peers to work through challenging concepts
- Asking clarifying questions when concepts were unclear
- Exploring connections between different topics covered in the course
- Engaging with supplementary material to deepen understanding

This active engagement has been essential to my learning process. The opportunity to work through problems, make mistakes, and learn from both my own errors and the insights of my peers has been invaluable in developing a deep understanding of programming language theory.

10 Conclusion

This course has fundamentally transformed my understanding of programming languages and computation itself. Stepping back from the technical details, I can now see how this course fits into the broader landscape of software engineering and computer science education.

10.1 Place in Software Engineering

In the wider world of software engineering, this course addresses a critical gap: the disconnect between how programmers write code and the mathematical structures that underlie computation. Most programming courses teach syntax and libraries, but this course taught the *why* behind programming language design. Understanding invariants helps me reason about program correctness. Understanding lambda calculus helps me appreciate functional programming paradigms. Understanding parsing helps me understand how compilers work. These theoretical foundations make me a more thoughtful programmer, capable of choosing appropriate abstractions and reasoning about program behavior.

The emphasis on mathematical rigor is particularly valuable in an industry increasingly concerned with correctness. As software systems become more critical—controlling medical devices, financial systems, and autonomous vehicles—the ability to reason formally about program behavior becomes essential. This course provides the foundational tools for such reasoning.

10.2 Most Interesting and Useful Aspects

The most fascinating aspect was discovering the Curry-Howard correspondence through the Lean Logic Game. The realization that proofs are programs and programs are proofs was a paradigm shift that connected

logic, computation, and mathematics in a way I had never seen before. This correspondence makes type systems, which I previously understood only pragmatically, suddenly make deep mathematical sense.

The most practically useful aspect was learning about invariants and termination analysis. These tools have already changed how I approach algorithm design. I now think about what properties remain unchanged during computation, which helps me understand algorithms more deeply and catch potential bugs earlier. The measure function technique for proving termination is a concrete tool I can apply to any recursive algorithm.

10.3 Critical Reflection and Suggestions

While the course content was excellent, I found the transition from imperative to functional thinking challenging. More explicit guidance on this paradigm shift, perhaps through comparison examples showing the same algorithm in both styles, would help students bridge this gap. Additionally, while the theoretical depth was valuable, occasional connections to real-world programming languages (showing how lambda calculus concepts appear in Haskell, or how parsing theory applies to Python's grammar) would help students see immediate practical applications.

The most valuable improvement would be more opportunities for collaborative problem-solving during class. Working through problems with peers, even when we made mistakes, was when I learned the most. Structured group activities or pair programming exercises on theoretical problems could enhance learning.

10.4 Final Thoughts

This course has been unlike any other in my computer science education. It didn't just teach me new concepts—it changed how I think about computation itself. I now see programming languages as mathematical objects that can be studied, reasoned about, and understood at a fundamental level. This perspective will inform my work as a software engineer, making me better at choosing the right tools, designing robust systems, and understanding the deeper principles that govern computation.

The mathematical rigor required throughout this course has been challenging but immensely rewarding. I leave this course not just with new knowledge, but with new ways of thinking that will serve me throughout my career in computer science.

References

[GEB] Douglas Hofstadter, Gödel, Escher, Bach: An Eternal Golden Braid, Basic Books, 1979.

[Church] Alonzo Church, The Calculi of Lambda-Conversion, Princeton University Press, 1941.

[BNF] John Backus, The Syntax and Semantics of the Proposed International Algebraic Language, 1959.

[ARS] Franz Baader and Tobias Nipkow, Term Rewriting and All That, Cambridge University Press, 1998.

[Term] Nachum Dershowitz and Jean-Pierre Jouannaud, Rewrite Systems, Handbook of Theoretical Computer Science, Volume B: Formal Models and Semantics, Elsevier, 1990.

[Termination] Nachum Dershowitz and Zohar Manna, Proving Termination with Multiset Orderings, Communications of the ACM, Volume 22, Issue 8, 1979.

[Euclid] Donald Knuth, The Art of Computer Programming, Volume 2: Seminumerical Algorithms, Addison-Wesley, 1997.

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

[Y] Haskell Curry, The Fixed Point Combinator, Journal of Symbolic Logic, Volume 23, 1958.

[Parsing] Alfred Aho, Monica Lam, Ravi Sethi, and Jeffrey Ullman, Compilers: Principles, Techniques, and Tools, 2nd Edition, Addison-Wesley, 2006.

[CurryHoward] Philip Wadler, Propositions as Types, Communications of the ACM, Volume 58, Issue 12, 2015.

[Lean] Leonardo de Moura and Sebastian Ullrich, The Lean 4 Theorem Prover and Programming Language, 2021.