

CPSC-354 Report

Nikolai Semerdjiev
Chapman University

December 15, 2025

Abstract

Contents

1	Introduction	2
2	Week by Week	2
2.1	Week 1	2
2.1.1	Notes and Exploration	2
2.1.2	Homework	2
2.1.3	Questions	3
2.2	Week 2	3
2.2.1	Homework	3
2.2.2	Questions	6
2.3	Week 3	6
2.3.1	Homework	6
2.3.2	Questions	7
2.4	Week 4	7
2.4.1	Homework	7
2.4.2	Questions	9
2.5	Week 5	9
2.5.1	Homework	9
2.5.2	Questions	10
2.6	Week 6	10
2.6.1	Homework	10
2.6.2	Questions	11
2.7	Week 7	11
2.7.1	Homework	11
2.7.2	Questions	16
2.8	Week 8	16
2.8.1	Homework	16
2.8.2	Questions	18
2.9	Week 9	18
2.9.1	Homework	18
2.9.2	Questions	19
2.10	Week 10	19
2.10.1	Homework	19
2.10.2	Questions	20
2.11	Week 11	20
2.11.1	Homework	20

2.11.2	Questions	21
2.12	Week 12	21
2.12.1	Homework	21
2.12.2	Questions	22
2.13	Week 13	23
2.13.1	Homework	23
2.13.2	Questions	26
3	Essay	26
4	Evidence of Participation	27
4.1	<i>Machine Code Explained</i> (Computerphile)	27
4.2	<i>Behind Mobile Game Design</i> (Olga Lazar)	27
4.3	<i>Creating Your Own Programming Language</i> (Computerphile)	27
4.4	Untyped Lambda Calculus Introduction	28
4.5	<i>Procedural Generation in Games</i>	28
5	Conclusion	29

1 Introduction

2 Week by Week

2.1 Week 1

2.1.1 Notes and Exploration

In Week 1, we began with the MIU (or MU) puzzle from Hofstadter's *Gödel, Escher, Bach*. This puzzle introduces the idea of formal systems and rules of inference. It is a good starting point to think about what it means to derive a string from an axiom under a fixed set of rules.

The MU Puzzle

RULES:

1. (Append-U) If a string ends with I, you may append U:
 $xI \rightarrow xIU$.
2. (Double) From Mx you may produce Mxx :
 $Mx \rightarrow Mxx$.
3. (III \rightarrow U) Replace any occurrence of III with U:
 $xIIIy \rightarrow xUy$.
4. (Delete UU) Delete any occurrence of UU:
 $xUUy \rightarrow xy$.

2.1.2 Homework

Problem: Can you derive MU from MI?

Solution: No, it is impossible to derive MU from MI. At first, playing around with the rules, I noticed that the goal was to create the correct number of I's so that they could be converted to a single U. This means we would need $N_I \bmod 3 = 0$, where N_I counts the I's.

Now, consider how each rule affects N_I :

- **(Append-U)** Appends a U, leaves N_I unchanged.
- **(Double)** $Mx \rightarrow Mxx$ doubles N_I ; in modular arithmetic, $N_I \mapsto 2N_I \pmod{3}$.

- **(III→U)** Removes three I's, leaving $N_I \bmod 3$ unchanged.
- **(Delete UU)** Only touches U's, leaves N_I unchanged.

Starting from MI, we have $N_I = 1 \equiv 1 \pmod{3}$. Doubling cycles between 1 and 2 modulo 3, never producing 0. Thus, it is impossible to reach $N_I \equiv 0 \pmod{3}$.

Since MU has $N_I = 0$, it cannot be derived from MI.

2.1.3 Questions

What is the reasoning behind being able to convert MIII into MU (using the rule $\text{III} \rightarrow \text{U}$) but not being able to go the other way (from MU to MIII)?

2.2 Week 2

2.2.1 Homework

Problem: Consider the following ARSs. Draw a picture for each one. Are the ARSs terminating? Are they confluent? Do they have unique normal forms?

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

Solution: For each ARS, I drew a graph (see figures) and analyzed:

- ****Termination:**** whether there are infinite chains. - ****Confluence:**** whether every divergence can rejoin.
- ****Unique normal forms:**** whether each element has a unique NF.

I will include one diagram for each ARS along with a short explanation of my analysis.

ARS 1: $A = \{\}, R = \{\}$ There is no infinite chain, no diverging paths exist as there is no path at all, and nothing exists to violate uniqueness. Therefore ✓ terminating, ✓ confluent, and ✓ has unique normal form.

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

•

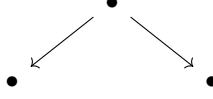
No rewrite steps: no infinite chain, no diverging paths exist or no path at all so it is vacuously satisfied and a is the only reachable normal form from a . Therefore ✓ terminating, ✓ confluent, and ✓ has unique normal form.

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



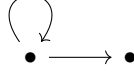
There is an infinite chain, $a \rightarrow^* a$. Since $a \rightarrow^* y$ and $a \rightarrow^* z$, therefore $y = z = a$, joining at a , and there are no normal forms as there is only one term, a , which has infinite outgoing rewrite steps. Therefore × terminating, ✓ confluent, and × unique normal form.

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



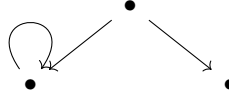
There is no infinite chain, two diverging paths that do not get joined, and two different normal forms from a . Therefore \checkmark terminating, \times confluent, and \times unique normal form.

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



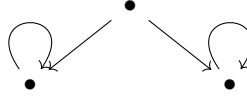
Even with one infinite chain it does not terminate, $a \rightarrow a$ and $a \rightarrow b$ and they join at b , b is the only normal form since a has an infinite chain therefore \times terminating, \checkmark confluent, has a \checkmark unique normal form.

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



Infinite chain at b (since $b \rightarrow b \rightarrow b \dots$). From a the system diverges to b and c with no connecting path to join them. The only normal form is c , but not every element reduces to a unique normal form, so the system is \times terminating, \times confluent, and \times has unique normal form.

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



There are two infinite chains ($b \rightarrow b \rightarrow \dots$ and $c \rightarrow c \rightarrow \dots$). From a the system diverges to b and c , but since b only reaches b and c only reaches c , the peak at a does not join. Every element has outgoing rewrite steps, so there are no normal forms. Therefore \times terminating, \times confluent, and \times has unique normal form.

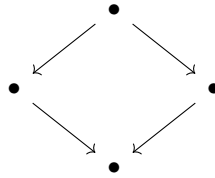
All Eight Combinations

The homework also asked to find examples of ARSs for each of the eight possible combinations of confluence, termination, and unique normal forms. The table below summarizes the results.

Confluent	Terminating	Unique Normal Forms	Example (A, R)
True	True	True	$A = \{a, b, c, d\}$, $R = \{(a, b), (a, c), (b, d), (c, d)\}$
True	True	False	N/A
True	False	True	$A = \{a, b, c, d\}$, $R = \{(a, a), (a, b), (a, c), (b, d), (c, d)\}$
True	False	False	$A = \{a, b, c, d\}$, $R = \{(a, b), (a, c), (b, d), (c, d), (d, d)\}$
False	True	True	N/A
False	True	False	$A = \{a, b, c, d\}$, $R = \{(a, b), (a, c), (b, d)\}$
False	False	True	$A = \{a, b, c\}$, $R = \{(a, b), (a, c), (b, d), (c, d), (d, d)\}$
False	False	False	$A = \{a, b, c\}$, $R = \{(a, a), (a, b), (a, c)\}$

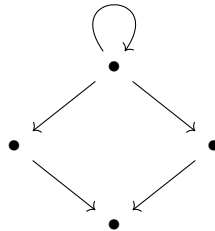
Example Graphs

Graph 1

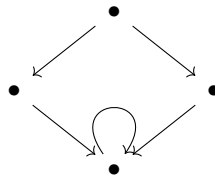


Graph 2 This combination is impossible as termination gives a normal form for each element AND confluence guarantees any two reduction paths from the same element to join therefore all reductions end at some normal form.

Graph 3

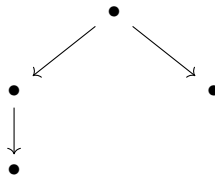


Graph 4

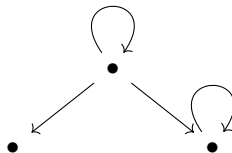


Graph 5 This case is impossible. Every system that is terminating and where every element has a unique normal form must also be confluent. Indeed, if $a \rightarrow^* y$ and $a \rightarrow^* z$, then both y and z reduce to some normal forms. Since the normal form is unique, y and z must reduce to the same normal form, which means the system is confluent.

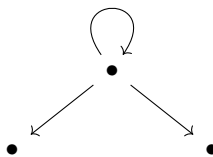
Graph 6



Graph 7



Graph 8



2.2.2 Questions

How can thinking about ARSs help us better understand the way programming languages define and control the process of evaluating programs?

2.3 Week 3

2.3.1 Homework

Exercise 5: Consider the rewrite rules

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

Reduce some example strings such as **abba** and **bababa**. **Answer:**

$$\text{abba} \rightarrow \text{abab} \rightarrow \text{aba} \rightarrow \text{aa} \rightarrow \varepsilon.$$

$$\text{bababa} \rightarrow \text{bbaaba} \rightarrow \text{bbba} \rightarrow \text{bba} \rightarrow \text{ba} \rightarrow \text{a}.$$

*No rule applies to a single **a**; this is the farthest you can go.*

Why is the ARS not terminating? **Answer:** There is always a way to terminate if you use the deletion rules (3) and (4), but using only the swap rules (1) and (2) yields an infinite loop (e.g., $\mathbf{ab} \leftrightarrow \mathbf{ba} \leftrightarrow \mathbf{ab} \leftrightarrow \dots$), so the ARS is *not terminating* overall.

Find two strings that are not equivalent. How many non-equivalent strings can you find? **Answer:** Two strings not equivalent: $\mathbf{baba} \not\leftrightarrow^* \mathbf{ababa}$. There are infinitely many non-equivalent pairs: strings split by the parity of the number of *a*'s (even vs. odd), so you can keep making pairs with different *a*-parity.

How many equivalence classes does \leftrightarrow^* have? Can you describe them in a nice way? What are the normal forms? **Answer:** There are 2 equivalence classes under \leftrightarrow^* : (1) strings with an *even* number of *a*'s, which reduce to the normal form ε ; (2) strings with an *odd* number of *a*'s, which reduce to the normal form *a*. Set of normal forms: $\{\varepsilon, a\}$.

Can you modify the ARS so that it becomes terminating without changing its equivalence classes? **Answer:** Yes. Drop $ab \rightarrow ba$ and keep $ba \rightarrow ab$, together with $aa \rightarrow \varepsilon$ and $b \rightarrow \varepsilon$. This orientation is terminating while preserving the same equivalence classes and normal forms (still $\{\varepsilon, a\}$).

Write down a question or two about strings that can be answered using the ARS. Think about whether this amounts to giving a semantics to the ARS. **Answer:** Given any string $x \in \{a, b\}^*$, does x reduce to ε or to *a*? Equivalently: is the number of *a*'s in x even or odd? This gives a semantics mapping each string to ε (even) or *a* (odd).

Exercise 5b: Consider the rewrite rules

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

Reduce some example strings such as `abba` and `bababa`. **Answer:**

$$\text{abba} \rightarrow \text{aba} \rightarrow \text{aa} \rightarrow a.$$

$$\text{bababa} \rightarrow \text{ababa} \rightarrow \text{aaba} \rightarrow \text{aaa} \rightarrow \text{aa} \rightarrow a.$$

No rule applies to a single `a`; this is the farthest you can go.

Why is the ARS not terminating? **Answer:** This example still has an infinite chain if you don't use rules (3) or (4): the swaps $ab \leftrightarrow ba$ can loop forever.

Find two strings that are not equivalent. How many non-equivalent strings can you find? **Answer:** Two non-equivalent strings: `bbb` $\not\leftrightarrow^*$ `babab`. There are infinitely many non-equivalent pairs.

How many equivalence classes does \leftrightarrow^* have? Can you describe them in a nice way? What are the normal forms? **Answer:** There are 2 equivalence classes for \leftrightarrow^* . If a string has *no* `a`'s, its normal form is ε . If a string has *at least one* `a`, its normal form is `a`. Set of normal forms: $\{\varepsilon, a\}$.

Can you modify the ARS so that it becomes terminating without changing its equivalence classes? **Answer:** Drop $ab \rightarrow ba$ and keep $ba \rightarrow ab$, along with $aa \rightarrow a$ and $b \rightarrow \varepsilon$. This orientation terminates while preserving the same equivalence classes and normal forms.

Write down a question or two about strings that can be answered using the ARS. Think about whether this amounts to giving a semantics to the ARS. **Answer:** Given any string $x \in \{a, b\}^*$, does x contain at least one `a`? Equivalently: does x reduce to `a` (some `a` present) or to ε (no `a`'s)?

2.3.2 Questions

How does changing the rewrite rule from $aa \rightarrow \varepsilon$ to $aa \rightarrow a$ resemble differences in evaluation strategies when creating a programming language?

2.4 Week 4

2.4.1 Homework

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

The algorithm always terminates provided that.

1. The inputs satisfy $a \geq 0$ and $b \geq 0$.
2. If $b = 0$ initially, the loop terminates immediately.
3. For $b > 0$, the remainder operation is defined by the division algorithm:

$$a = qb + r, \quad 0 \leq r < b,$$

where $r = a \bmod b$.

Define the measure function

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

Consider one loop iteration:

$$(a, b) \mapsto (b, a \bmod b).$$

By the division algorithm,

$$0 \leq a \bmod b < b.$$

Therefore,

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

Hence ϕ strictly decreases with each iteration.

Since $\phi(a, b)$ is a nonnegative integer, it cannot decrease infinitely. After finitely many steps, we must reach $b = 0$, at which point the loop terminates.

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

The function

$$\phi(\text{left}, \text{right}) = \text{right} - \text{left} + 1$$

is a measure function for `merge_sort` (with integer indices and `mid` = $\lfloor (\text{left} + \text{right})/2 \rfloor$).

Assume $\text{left}, \text{right} \in \mathbb{Z}$ and the procedure is only called with $\text{left} \leq \text{right}$. The branching factor is finite (at most two recursive calls).

Let $n = \phi(\text{left}, \text{right}) = \text{right} - \text{left} + 1$.

- *Base case.* If $\text{left} \geq \text{right}$ then $n \leq 1$ and the function returns without making recursive calls. No decrease condition needs to be checked.
- *Recursive case.* Suppose $\text{left} < \text{right}$, hence $n \geq 2$. Set $\text{mid} = \left\lfloor \frac{\text{left} + \text{right}}{2} \right\rfloor$. Then $\text{left} \leq \text{mid} < \text{right}$, so both subproblems are well-formed:

$$(\text{left}, \text{mid}) \quad \text{and} \quad (\text{mid} + 1, \text{right}).$$

Their measures are

$$\phi(\text{left}, \text{mid}) = \text{mid} - \text{left} + 1 \quad \text{and} \quad \phi(\text{mid} + 1, \text{right}) = \text{right} - \text{mid}.$$

Using $\text{mid} = \left\lfloor \frac{\text{left} + \text{right}}{2} \right\rfloor$ and $n = \text{right} - \text{left} + 1$, we get the bounds

$$\phi(\text{left}, \text{mid}) \leq \left\lceil \frac{n}{2} \right\rceil \quad \text{and} \quad \phi(\text{mid} + 1, \text{right}) \leq \left\lfloor \frac{n}{2} \right\rfloor.$$

Since $n \geq 2$, both $\lceil \frac{n}{2} \rceil < n$ and $\lfloor \frac{n}{2} \rfloor < n$ hold. Therefore

$$\phi(\text{left}, \text{mid}) < \phi(\text{left}, \text{right}) \quad \text{and} \quad \phi(\text{mid} + 1, \text{right}) < \phi(\text{left}, \text{right}).$$

Hence the measure strictly decreases along every recursive edge.

Because ϕ maps states to \mathbb{N} and strictly decreases with each recursive call, no infinite descent is possible. Together with finite branching, this implies termination.

2.4.2 Questions

How do different evaluation strategies in programming languages (such as call-by-value vs. call-by-name) influence whether a program is guaranteed to terminate, and in what ways can techniques like measure functions help us reason about termination across these strategies?

2.5 Week 5

2.5.1 Homework

$$\textbf{Goal: } (\lambda f. \lambda x. f(fx)) (\lambda f. \lambda x. f(f(fx))).$$

–**renaming.** To avoid variable capture, rename bound variables so the two arguments use distinct names:

$$A \equiv \lambda f. \lambda x. f(fx) \quad \text{and} \quad B \equiv \lambda g. \lambda y. g(gy).$$

The term is AB .

–**reduction (outer application).**

$$AB \rightarrow_{\beta} \lambda x. B(Bx) \quad (\text{substitute } f := B \text{ in } f(fx)).$$

–**reduction of Bx .**

$$Bx = (\lambda g. \lambda y. g(gy))x \rightarrow_{\beta} \lambda y. x(xy).$$

Name this $B_x \equiv \lambda y. x^3(y)$, where x^k denotes k -fold self-composition.

–**reduction of $B(B_x)$.**

$$B(B_x) = (\lambda g. \lambda y. g(gy))(\lambda y. x^3(y)) \rightarrow_{\beta} \lambda y. B_x(B_x y).$$

Since $B_x y = x^3(y)$, we compute

$$B_x(B_x y) = B_x(x^3(y)) = x^3(x^3(y)) = x^6(y),$$

and then

$$B_x(B_x(B_x y)) = B_x(x^6(y)) = x^3(x^6(y)) = x^9(y).$$

Hence

$$B(B_x) \rightarrow \lambda y. x^9(y).$$

Assemble the result.

$$\lambda x. B(Bx) \rightarrow \lambda x. \lambda y. x^9(y).$$

Renaming $x \mapsto f$ and $y \mapsto a$ gives the canonical Church-numeral form

$$\boxed{\lambda f. \lambda a. f^9(a)}.$$

Thus

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

Remark (mathematical interpretation). In Church encodings, natural numbers are functions that iterate an endofunction:

$$n \equiv \lambda f. \lambda x. f^n(x).$$

Here, $\lambda f. \lambda x. f(f x)$ is the numeral 2 (“apply f twice”) and $\lambda f. \lambda x. f(f(f x))$ is the numeral 3 (“apply f thrice”). Applying one numeral to another composes iterations and realizes *exponentiation by iteration*:

$$(2\ 3) = \lambda x. 3^2(x) = \lambda f. \lambda x. f^9(x),$$

so the result corresponds to the number $9 = 3^2$. Conceptually, this is *function iteration*: “do thrice, then do thrice again” = “do nine times.” Before computers, Church’s lambda calculus provided a purely symbolic foundation where numbers are actions (iterations) on functions; your reduction is exactly the arithmetic law that composing “apply-thrice” with itself yields “apply-nine-times.”

2.5.2 Questions

Since Church numerals let us represent arithmetic entirely through function application, what does that suggest about the nature of mathematics—are operations like addition, multiplication, and exponentiation really ‘just’ repeated patterns of substitution and iteration?

2.6 Week 6

2.6.1 Homework

Evaluating fact 3 with fix, let, and let rec

Rules

$$\text{fix } F \rightarrow F(\text{fix } F) \quad (\text{def of fix})$$

$$\text{let } x = e_1 \text{ in } e_2 \rightarrow (\lambda x. e_2) e_1 \quad (\text{def of let})$$

$$\text{let rec } f = e_1 \text{ in } e_2 \rightarrow \text{let } f = \text{fix } (\lambda f. e_1) \text{ in } e_2 \quad (\text{def of let rec})$$

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

Target term

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

Derivation (each step labeled)

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

Expand $(\text{fix } F) 2$:

$$\begin{aligned} (\text{fix } F) 2 &\xrightarrow{\text{def of fix}} (F (\text{fix } F)) 2 \xrightarrow{\beta} (\lambda n. \text{if } n = 0 \text{ then } 1 \text{ else } n * (\text{fix } F) (n - 1)) 2 \\ &\xrightarrow{\beta} \text{if } 2 = 0 \text{ then } 1 \text{ else } 2 * (\text{fix } F) 1 \xrightarrow{\text{def of if}} 2 * (\text{fix } F) 1. \end{aligned}$$

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

$$\begin{aligned} (\text{fix } F) 1 &\xrightarrow{\text{def of fix}} (F (\text{fix } F)) 1 \xrightarrow{\beta} (\lambda n. \text{if } n = 0 \text{ then } 1 \text{ else } n * (\text{fix } F) (n - 1)) 1 \\ &\xrightarrow{\beta} \text{if } 1 = 0 \text{ then } 1 \text{ else } 1 * (\text{fix } F) 0 \xrightarrow{\text{def of if}} 1 * (\text{fix } F) 0. \end{aligned}$$

Base case $(\text{fix } F) 0$:

$$\begin{aligned} (\text{fix } F) 0 &\xrightarrow{\text{def of fix}} (F (\text{fix } F)) 0 \xrightarrow{\beta} (\lambda n. \text{if } n = 0 \text{ then } 1 \text{ else } n * (\text{fix } F) (n - 1)) 0 \\ &\xrightarrow{\beta} \text{if } 0 = 0 \text{ then } 1 \text{ else } 0 * (\text{fix } F) (-1) \xrightarrow{\text{def of if}} 1. \end{aligned}$$

Back-substitute the arithmetic:

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

2.6.2 Questions

If the **fix** operator allows a function to “refer to itself” without any explicit naming or looping construct, what does that reveal about recursion as a *mathematical* idea rather than merely a *programming* one?

Does this suggest that self-reference—and therefore recursion—emerges purely from the rules of substitution and function application, rather than from syntactic features like **while** or **for** loops?

2.7 Week 7

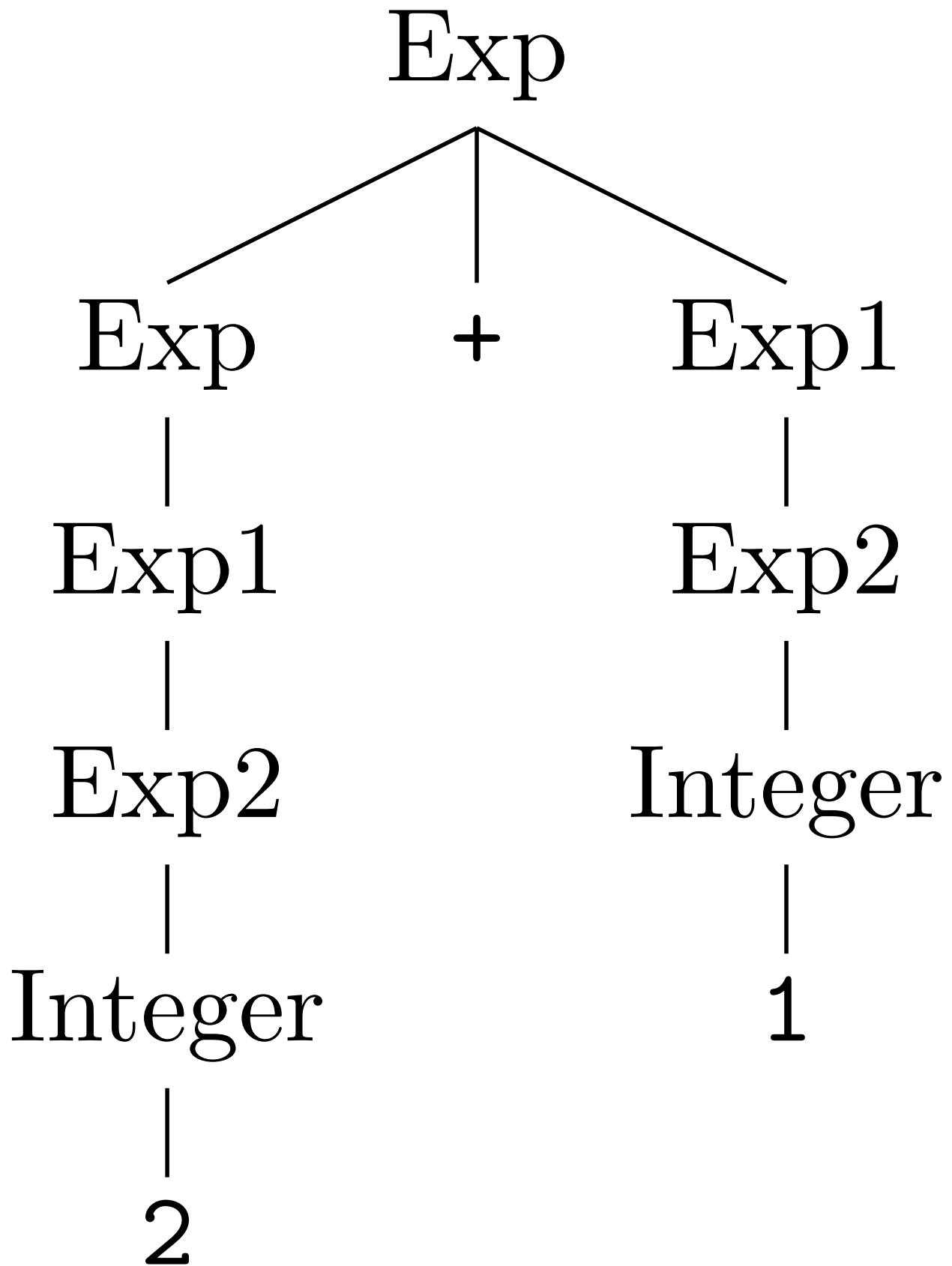
2.7.1 Homework

Parse Trees for the Given Grammar

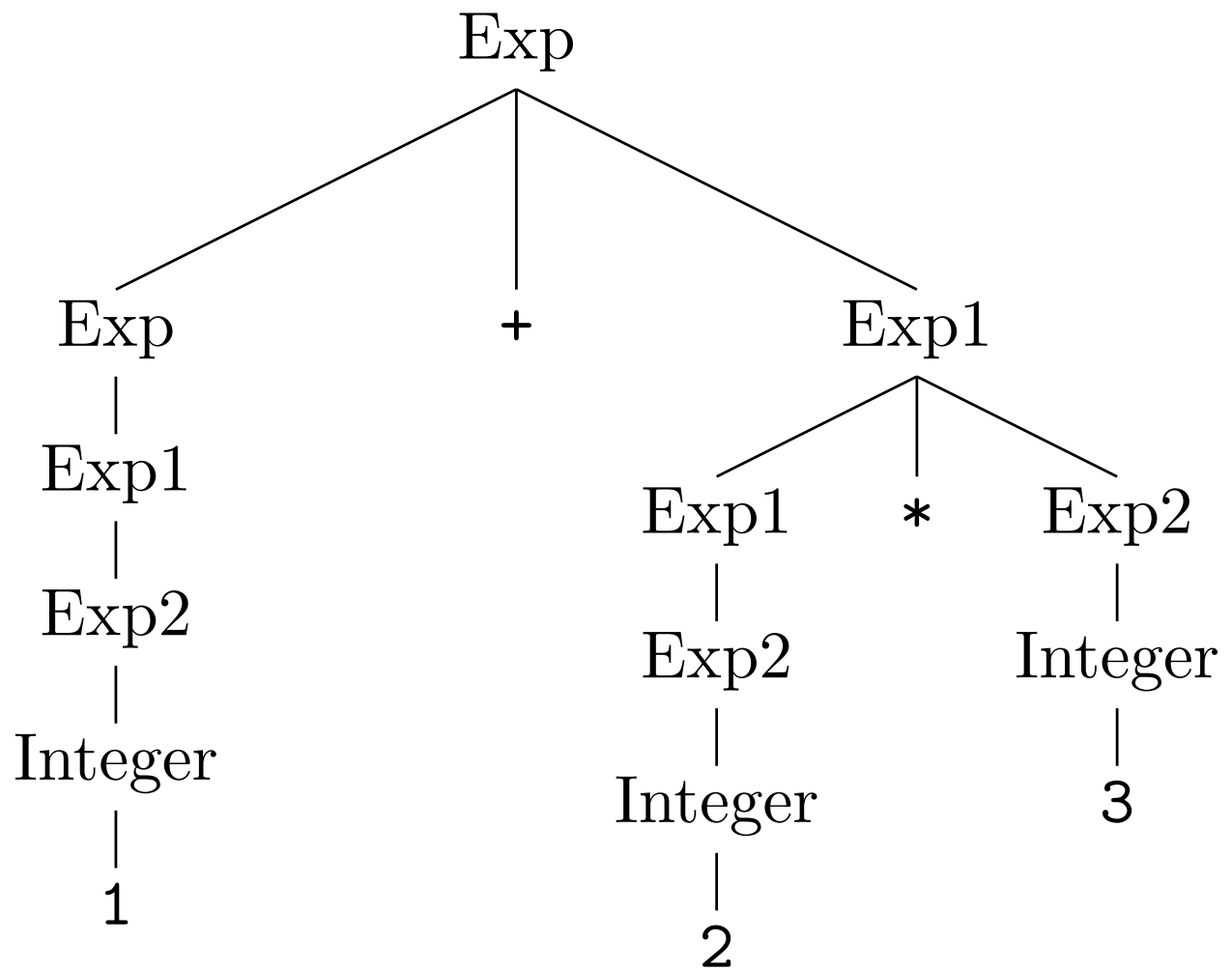
Grammar

$$\begin{aligned} \text{Exp} &\rightarrow \text{Exp} + \text{Exp1} \mid \text{Exp1} \\ \text{Exp1} &\rightarrow \text{Exp1} * \text{Exp2} \mid \text{Exp2} \\ \text{Exp2} &\rightarrow \text{Integer} \mid (\text{Exp}) \end{aligned}$$

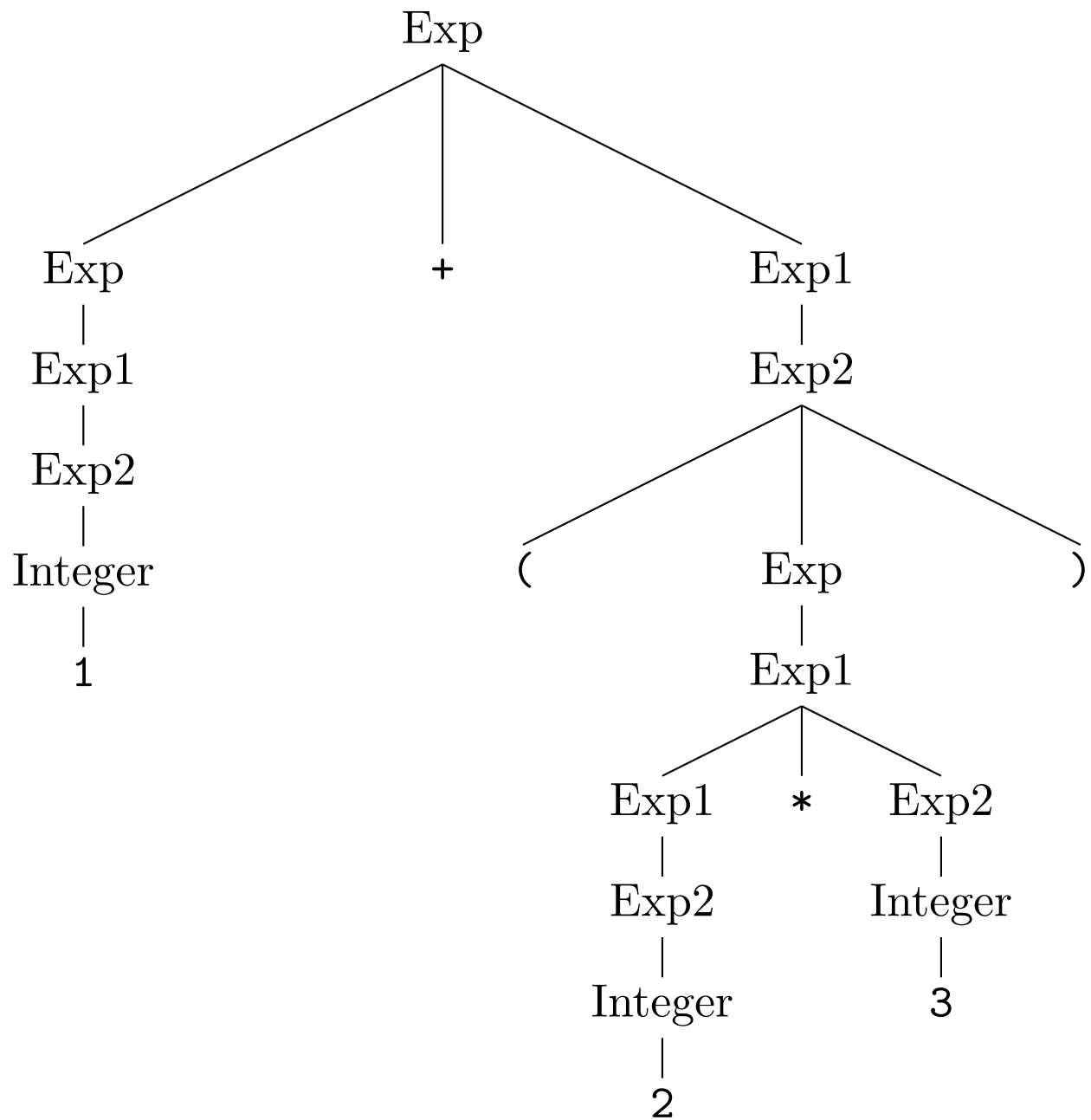
1) 2+1



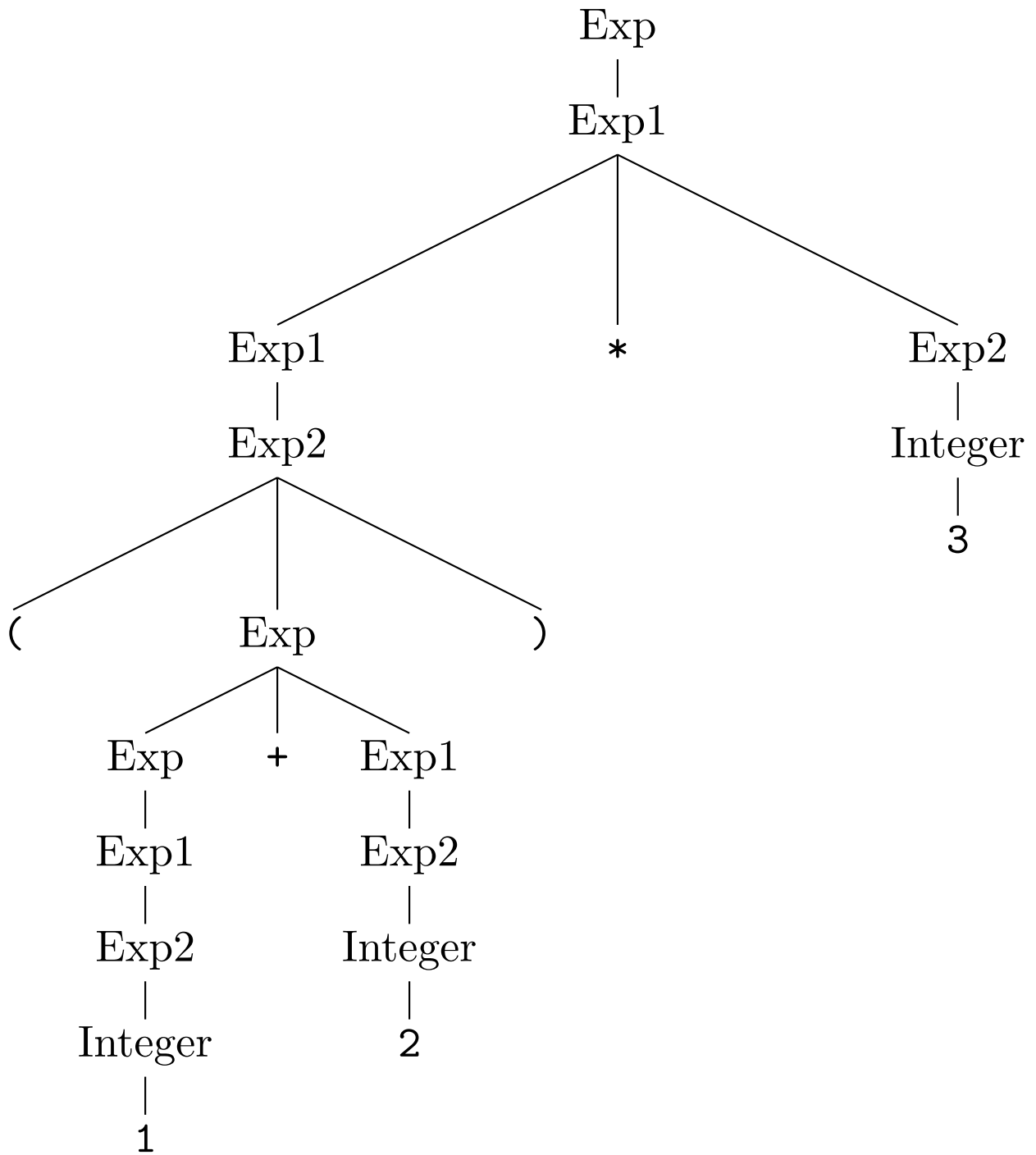
2) 1+2*3



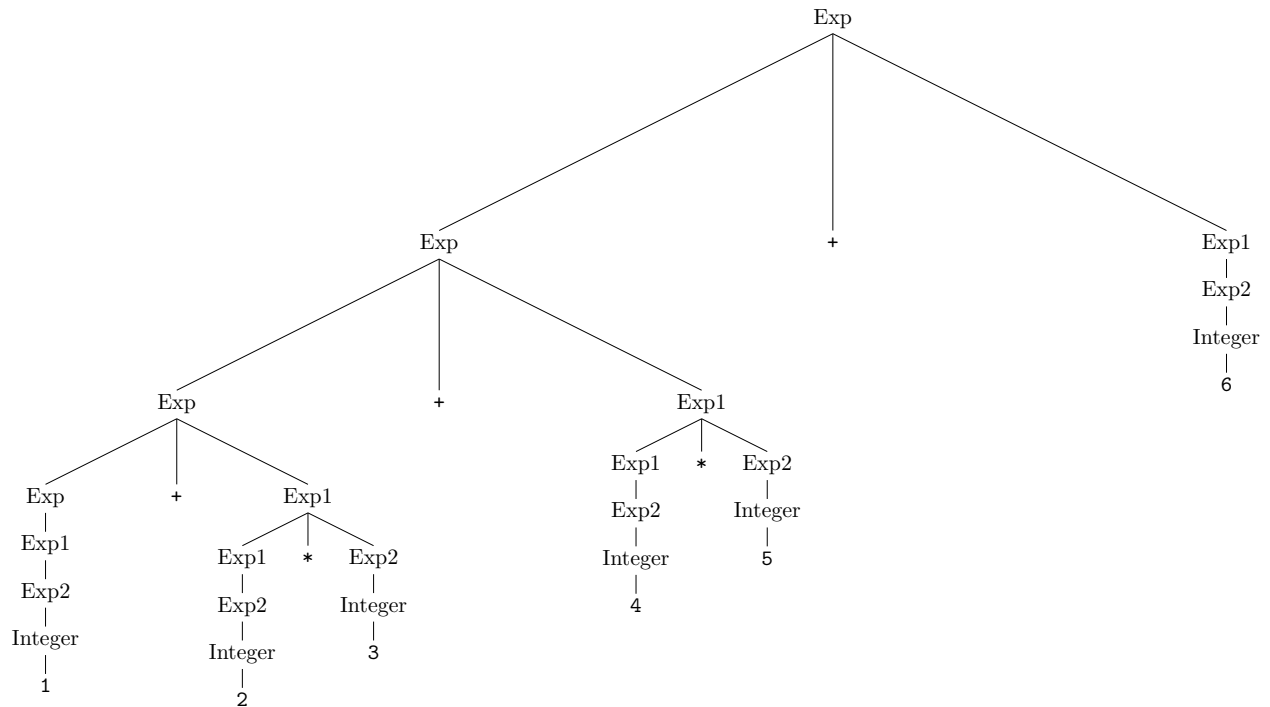
3) $1+(2*3)$



4) $(1+2)*3$



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



2.7.2 Questions

When a grammar allows multiple ways to parse the same expression, such as different groupings of additions and multiplications, how can we modify or design the grammar so that the parse tree always reflects the correct operator precedence and associativity?

2.8 Week 8

2.8.1 Homework

Level 5/8: *Adding zero*

Question. Prove, for natural numbers a, b, c ,

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

Using the identity $x + 0 = x$ (often named `add_zero`) twice:

$$\begin{aligned} a + (b + 0) + (c + 0) &= a + b + (c + 0) && \text{(by } b + 0 = b\text{)} \\ &= a + b + c && \text{(by } c + 0 = c\text{)}. \end{aligned}$$

Lean script.

```
-- Objects: a b c :
rw [add_zero] -- a + (b + 0) + (c + 0) ==> a + b + (c + 0)
rw [add_zero] -- a + b + (c + 0) ==> a + b + c
rfl          -- both sides now identical
```

Level 6/8: *Precision rewriting*

Question. Prove, for natural numbers a, b, c ,

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

but emphasize *targeted* rewriting (i.e., rewriting only the needed subterm each step).

Same identities as above, but apply them precisely to the subterm containing 0:

$$\begin{aligned} a + (b + 0) + (c + 0) &= a + b + (c + 0) && \text{rewrite only the inner } (b + 0) \\ &= a + b + c && \text{now rewrite the remaining } (c + 0). \end{aligned}$$

Lean script (one precise rewrite at a time).

```
-- Objects: a b c :
rw [add_zero]    -- rewrite (b + 0) -> b
rw [add_zero]    -- rewrite (c + 0) -> c
rfl
```

(If the goal were more complicated, one could direct a rewrite to a specific occurrence using a location, e.g. `'rw [add_zero] at h'` or `'simp [add_zero] with sideconditions.`)

Level 7/8: *add_succ*

Theorem (stated).

$$\forall n : \mathbb{N}, \quad n.\text{succ} = n + 1.$$

Useful lemmas.

$$\begin{aligned} 1 &= \text{succ}(0) && \text{(named one_eq_succ_zero),} \\ n + \text{succ}(k) &= \text{succ}(n + k) && \text{(named add_succ),} \\ n + 0 &= n && \text{(named add_zero).} \end{aligned}$$

$$\begin{aligned} n.\text{succ} &= n + 0.\text{succ} && \text{(since succ}(x) = x + 1, \text{ we move via } 1 = \text{succ}(0)) \\ &= \text{succ}(n + 0) && \text{by add_succ} \\ &= \text{succ}(n) && \text{by add_zero} \\ &= n + 1 && \text{unwinding } 1 = \text{succ}(0). \end{aligned}$$

Concretely (mirroring your screenshot), we first rewrite the 1 as `succ(0)`, then use `add_succ`, then `add_zero`, and finish with reflexivity.

Lean script (exact sequence from the level).

```
-- Goal: n.succ = n + 1
rw [one_eq_succ_zero] -- turn 1 into succ 0 on the RHS
rw [add_succ]         -- n + succ 0 -> (n + 0).succ
rw [add_zero]         -- (n + 0).succ -> n.succ
rfl                  -- both sides are n.succ
```

Notes.

- Levels 5 and 6 highlight the same identity $x + 0 = x$ but train you to control *where* a rewrite happens.
- Level 7 chains small lemmas to reach a familiar arithmetic fact: $n + 1 = \text{succ}(n)$.

Question 8 — Natural–Language Proof (Level 8: $2 + 2 = 4$)

Goal. Prove that $2 + 2 = 4$ in Peano arithmetic.

Background/Definitions. Let 0 be the base natural number and let $\text{succ}(n)$ (sometimes written $n.\text{succ}$) denote the successor of n . Define the numerals

$$1 = \text{succ}(0), \quad 2 = \text{succ}(1), \quad 3 = \text{succ}(2), \quad 4 = \text{succ}(3).$$

Addition is defined by the standard recursion rules:

$$(\text{add_zero}) \quad n + 0 = n, \quad (\text{add_succ}) \quad n + \text{succ}(k) = \text{succ}(n + k).$$

Proof. We compute $2 + 2$ using only the recursion rules for $+$ and the numeral definitions.

$$\begin{aligned} 2 + 2 &= \text{succ}(1) + \text{succ}(1) && (\text{by the definition of } 2) \\ &= \text{succ}(\text{succ}(1) + 1) && (\text{by } \text{add_succ} \text{ with } n = \text{succ}(1), k = 1) \\ &= \text{succ}(\text{succ}(1) + \text{succ}(0)) && (\text{since } 1 = \text{succ}(0)) \\ &= \text{succ}(\text{succ}(\text{succ}(1) + 0)) && (\text{by } \text{add_succ} \text{ with } n = \text{succ}(1), k = 0) \\ &= \text{succ}(\text{succ}(\text{succ}(1))) && (\text{by } \text{add_zero}: \text{succ}(1) + 0 = \text{succ}(1)) \\ &= \text{succ}(\text{succ}(\text{succ}(1))) && \\ &= \text{succ}(\text{succ}(\text{succ}(\text{succ}(0)))) && (\text{since } 1 = \text{succ}(0)) \\ &= \text{succ}(\text{succ}(\text{succ}(\text{succ}(0)))) && \\ &= 4 && (\text{by the definition of } 4). \end{aligned}$$

Therefore, $2 + 2 = 4$. □

(Optional) Lean step correspondence. The proof above mirrors the scripted steps:

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

2.8.2 Questions

In Peano arithmetic, every arithmetic fact—such as $2 + 2 = 4$ —is proven step by step from simple definitions like 0 and $\text{succ}(n)$. What does this tell us about how much of mathematics can be built from just a few basic rules, and why might this level of formality be both powerful and limiting when compared to the way we usually do arithmetic?

2.9 Week 9

2.9.1 Homework

Here’s a compile-ready LaTeX block you can drop into your Week 9 report. It contains *two solutions* for Addition World – *Level 5: (add_right_comm) : one by induction (spelled out in English math) and one without induction (a short proof using the lemma ‘ add_right_comm ’). I’ve also included the Lean script that matches each proof.*

“`latex`

Addition World — Level 5: add_right_comm

Theorem (goal). For all natural numbers a, b, c ,

$$a + b + c = a + c + b.$$

Solution A: Inductive proof on b (no tactics required to understand)

We use the Peano rules for addition:

$$(\text{add_zero}) \ n + 0 = n, \quad (\text{add_succ}) \ n + \text{succ}(k) = \text{succ}(n + k),$$

together with associativity $(x + y) + z = x + (y + z)$ and commutativity $x + y = y + x$.

Claim. For all a, c , $P(b) : a + b + c = a + c + b$ holds by induction on b .

Base case $b = 0$.

$$a + 0 + c = a + c = a + c + 0,$$

by two uses of `add_zero`. Hence $P(0)$ holds.

Inductive step. Assume $P(b)$ holds, i.e.

$$a + b + c = a + c + b \quad \text{for all } a, c.$$

We must show $P(\text{succ } b)$:

$$a + \text{succ } b + c = a + c + \text{succ } b.$$

Compute:

$$\begin{aligned} a + \text{succ } b + c &= (a + \text{succ } b) + c \\ &= \text{succ}(a + b) + c && (\text{by add_succ}) \\ &= \text{succ}((a + b) + c) && (\text{by succ_add: succ } x + c = \text{succ}(x + c)) \\ &= \text{succ}(a + b + c) && (\text{associativity}) \\ &= \text{succ}(a + c + b) && (\text{induction hypothesis}) \\ &= a + c + \text{succ } b && (\text{reverse of add_succ}). \end{aligned}$$

Thus $P(\text{succ } b)$ holds, and by induction $a + b + c = a + c + b$ for all a, b, c .

Solution B: Direct proof *without* induction (one-line rewrite)

If the library lemma `add_right_comm` is available, it states exactly

$$x + y + z = x + z + y.$$

Therefore, taking $x = a$, $y = b$, $z = c$ gives the result immediately:

$$a + b + c = a + c + b.$$

2.9.2 Questions

Why do we need induction to prove some arithmetic properties like commutativity or associativity, but not others? In other words, what makes a property like $a + b + c = a + c + b$ provable directly from existing lemmas, while others require an inductive step on one of the variables?

2.10 Week 10

2.10.1 Homework

Lean Logic Game — Party Snacks (Levels 6–9, one line each)

Level 6: `and_imp`

One-liner:

```
exact (fun c => fun d => h (And.intro c d))
-- equivalently: exact fun c d => h c, d
```

Level 7: and_imp 2

One-liner:

```
exact (fun hd => h hd.left hd.right)
-- equivalently: exact fun hc, hd => h hc hd
```

Level 8: Distribute

One-liner:

```
exact fun s => And.intro (h.left s) (h.right s)
-- equivalently: exact fun s => h.left s, h.right s
```

Level 9: Uncertain Snacks

One-liner:

```
exact fun r => And.intro (fun _ : S => r) (fun _ : S => r)
-- equivalently: exact fun r => ((fun _ : S => r), (fun _ : S => r))
```

2.10.2 Questions

How does writing logical proofs as functions in Lean change the way we think about reasoning with implications? Does it make the structure of logical arguments clearer compared to traditional symbolic proofs on paper?

2.11 Week 11

2.11.1 Homework

Lean Logic Game — Negation (Levels 9–12, one line each)

Level 9 / 12: Implies a Negation

Problem.

$$\text{example } (A P : \text{Prop}) (h : P \rightarrow \neg A) : \neg(P \wedge A)$$

One-line solution (Lean).

```
exact fun (p, a) => (h p) a
```

Level 10 / 12: Conjunction Implication

Problem.

$$\text{example } (A P : \text{Prop}) (h : \neg(P \wedge A)) : P \rightarrow \neg A$$

One-line solution (Lean).

```
exact fun p a => h (p, a)
```

Level 11 / 12: not_not_not

Problem.

example (A : Prop) (h : $\neg\neg\neg A$) : $\neg A$

One-line solution (Lean).

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

Level 12 / 12: ~Intro Boss

Problem.

example (B C : Prop) (h : $\neg(B + C)$) : $\neg\neg B$

One-line solution (Lean).

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

2.11.2 Questions

How does encoding $\neg A$ as $A \rightarrow \text{False}$ help us construct one-line proofs using only function application? Does this viewpoint make the structure of negation proofs clearer than traditional derivations?

2.12 Week 12

2.12.1 Homework

Towers of Hanoi (2025) — Activity Write-up (n = 5, from peg 0 to peg 2)

Algorithm (rules)

hanoi 1 x y = move x y
hanoi (n+1) x y = hanoi n x (other x y); move x y; hanoi n (other x y) y

(1) Completed execution trace for hanoi 5 0 2

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

```

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

```

(2) Moves extracted from the trace (in order)

Each line `move x y` is a move of the top disk from peg x to peg y . There are $2^5 - 1 = 31$ moves, the minimal number for $n=5$.

$(0 \rightarrow 2), (0 \rightarrow 1), (2 \rightarrow 1), (0 \rightarrow 2), (1 \rightarrow 0), (1 \rightarrow 2), (0 \rightarrow 2),$
 $(0 \rightarrow 1), (2 \rightarrow 1), (2 \rightarrow 0), (1 \rightarrow 0), (2 \rightarrow 1), (0 \rightarrow 2), (0 \rightarrow 1), (2 \rightarrow 1),$
 $(0 \rightarrow 2),$
 $(1 \rightarrow 0), (1 \rightarrow 2), (0 \rightarrow 2), (1 \rightarrow 0), (2 \rightarrow 1), (2 \rightarrow 0), (1 \rightarrow 0),$
 $(1 \rightarrow 2),$
 $(0 \rightarrow 2), (0 \rightarrow 1), (2 \rightarrow 1), (0 \rightarrow 2), (1 \rightarrow 0), (1 \rightarrow 2), (0 \rightarrow 2).$

(3) How to verify against the online Towers of Hanoi

[leftmargin=1.3em]Open the online Towers of Hanoi and set $n = 5$, source peg 0, target peg 2. Play the moves above in order; every move is legal (never places a larger disk on a smaller one). You will finish in exactly 31 moves, which is minimal for $n = 5$. (Why this works.) The algorithm preserves the invariant “the top k disks are always in legal configuration”, and the recursive pattern

$$\text{hanoi } n \ x \ y = \text{hanoi } (n-1) \ x \ z; \text{ move } x \ y; \text{ hanoi } (n-1) \ z \ y$$

ensures inductively that the largest disk moves exactly once (midpoint move), surrounded by optimal solutions for size $n-1$.

2.12.2 Questions

Why does the recursive algorithm for the Towers of Hanoi always produce the minimum possible number of moves, and how does seeing the full execution trace help you understand why no shorter solution could

exist?

2.13 Week 13

2.13.1 Homework

Item 2: Testing the Interpreter

2.1 Does the implementation conform to the theory?

Running `python interpreter_test.py` executes a battery of tests that check beta-reduction, alpha-conversion, and the chosen evaluation strategy. All the provided tests pass, so the implementation behaves as specified by the mathematical theory of the untyped lambda calculus (in particular: correct handling of bound vs. free variables and correct sequencing of reductions).

2.2 Additional test cases

I added several expressions inspired by the lecture notes to `test.lc`, always predicting the expected normal form before running the interpreter:

1. $a\ b\ c\ d$

By definition, application in the lambda calculus is left-associative, so

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

The interpreter prints exactly this left-nested structure.

- (a)

Parentheses do not change the term, so (a) and a are alpha-identical. There is no beta-redex, so (a) is already in normal form and reduces to a .

- Church booleans

$$\text{true} \equiv \lambda t.\lambda f.t \quad \text{false} \equiv \lambda t.\lambda f.f$$

and the conditional

$$\text{if} \equiv \lambda b.\lambda x.\lambda y.b\ x\ y.$$

For example

$$\text{if true } M\ N \Rightarrow_{\beta}^* M,$$

and

$$\text{if false } M\ N \Rightarrow_{\beta}^* N.$$

The interpreter returns the expected branch in both cases.

- Church numerals

$$\mathbf{0} \equiv \lambda f.\lambda x.x, \quad \mathbf{1} \equiv \lambda f.\lambda x.f\ x, \quad \mathbf{2} \equiv \lambda f.\lambda x.f(fx), \quad \mathbf{3} \equiv \lambda f.\lambda x.f(f(fx)).$$

From HW5 we know that

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

should reduce to the Church numeral for 9, i.e.

$$\mathbf{9} \equiv \lambda f.\lambda x.\underbrace{f(f(\dots f\ x\dots))}_{9\ \text{times}}.$$

The interpreter indeed normalizes this term to a lambda-term of the form $\lambda f.\lambda x.f^9x$.

These experiments support the claim that the implementation conforms to the intended operational semantics (normal-order beta-reduction with alpha-conversion).

Item 3: Capture-avoiding substitution

Substitution is written $[x := N]M$, meaning “replace all *free* occurrences of x in M by N , avoiding variable capture”. The mathematical definition is by structural recursion on M .

1. Variables:

$$[x := N]x = N \quad [x := N]y = y \quad \text{if } y \neq x.$$

2. Application:

$$[x := N](M_1 M_2) = ([x := N]M_1) ([x := N]M_2).$$

3. Abstraction: there are three cases.

$$[x := N](\lambda y.M) = \begin{cases} \lambda y.M, & \text{if } y = x, \\ \lambda y.[x := N]M, & \text{if } y \neq x \text{ and } y \notin \text{FV}(N), \\ \lambda z.[x := N]M[y := z], & \text{if } y \neq x \text{ and } y \in \text{FV}(N), \end{cases}$$

where z is a fresh variable not occurring free in M or N and $M[y := z]$ is a *capture-avoiding* renaming of the bound variable y to z .

The third case is where alpha-conversion happens: before substituting, we rename the binder λy to a fresh λz so that the free variables of N never accidentally become bound.

Implementation sketch. In `interpreter.py` this is represented as a recursive `substitute(var, repl, term)` function, where `term` is an AST node of one of three kinds: variable, abstraction, or application. When substitution enters an abstraction $\lambda y.M$ and the bound variable `y` is in the free variables of `repl`, the interpreter first generates a fresh name like `Var1`, `Var2`, ..., renames the binder and its bound occurrences (alpha-conversion), and only then recurses. This is exactly the algorithm described above.

Item 4: Do all computations reduce to normal form?

For all “well-behaved” examples from the lectures (Church booleans, numerals, pairs, lists, etc.) the interpreter produced the expected normal forms. In particular, finite numeric computations and conditionals always normalized.

However, the untyped lambda calculus is Turing-complete, so there exist terms with no normal form. For such terms the interpreter keeps performing beta reductions forever (or, in practice, runs until we stop it), and no normal form is reached. Thus:

- For all test cases encoding finite computations, the interpreter gives the expected result.
- Not all lambda-terms reduce to normal form; some diverge.

Item 5: Smallest non-normalizing λ -expression

A standard minimal working example (MWE) of a non-normalizing term is

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

Reduction:

$$(\lambda x. x x) (\lambda x. x x) \Rightarrow_{\beta} [x := \lambda x. x x](x x) = (\lambda x. x x) (\lambda x. x x) = \Omega.$$

So one beta-reduction step takes us back to the same term. Every proper subterm of Ω is either a variable or an abstraction and hence already in normal form, so Ω is a very small example of a term that does not reduce to normal form.

Item 7: Substitution trace for

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

We want to mirror the interpreter's beta-reduction steps, focusing on substitution. Below, each line corresponds to a single substitution (beta-step); for readability we omit some obvious alpha-renamings and keep variable names distinct by convention.

$$\begin{aligned}
& ((\lambda m. \lambda n. m\ n)(\lambda f. \lambda x. f(f\ x)))(\lambda f. \lambda x. f(f(f\ x))) \\
\Rightarrow_{\beta} & (\lambda n. (\lambda f. \lambda x. f(f\ x))\ n)(\lambda f. \lambda x. f(f(f\ x))) && \text{substitute } m := (\lambda f. \lambda x. f(f\ x)) \\
\Rightarrow_{\beta} & (\lambda n. \lambda x. n(n\ x))(\lambda f. \lambda x. f(f(f\ x))) && \text{substitute } f := n \text{ in } \lambda x. f(f\ x) \\
\Rightarrow_{\beta} & \lambda x. (\lambda f. \lambda x. f(f(f\ x)))(\lambda f. \lambda x. f(f(f\ x))\ x) && \text{substitute } n := (\lambda f. \lambda x. f(f(f\ x))) \\
\Rightarrow_{\beta} & \lambda x. (\lambda x'. (\lambda f. \lambda x. f(f(f\ x)))(f_1)) && \text{(alpha-renaming of inner } x \text{ to } x') \\
& \quad \text{where } f_1 \text{ abbreviates } \lambda x. f(f(f\ x)) \\
\Rightarrow_{\beta} & \lambda f. \lambda x. \underbrace{f(f(\dots f\ x\dots))}_{9 \text{ times}}
\end{aligned}$$

The detailed middle steps just repeatedly expand the nested applications of the Church numeral **3** to itself; after fully unwinding, the body contains exactly nine occurrences of f , so the result is the Church numeral **9**.

Item 8: Recursive evaluation trace for

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

Here we sketch the call trace of the interpreter's `evaluate` and `substitute` functions, in the same style as the recursive trace of `hanoi`. We write $\ell_{12}, \ell_{39}, \dots$ as symbolic labels for the source-line numbers where these calls occur; in your own trace you should replace them with the actual line numbers from `interpreter.py`.

```

12: evaluate( ((\m.(\n.(m n))) (\f.(\x.(f (f x))))) (\f.(\x.(f x))) )
39: evaluate( (\m.(\n.(m n))) (\f.(\x.(f (f x))))) )
52: evaluate( \m.(\n.(m n)) )
53: evaluate( \f.(\x.(f (f x))) )
70: substitute( m := (\f.(\x.(f (f x))))
      in \n.(m n) )
80: (possible alpha-conversion inside substitute)
90: evaluate( \n.((\f.(\x.(f (f x)))) n) )
39: evaluate( \f.(\x.(f x)) )
70: substitute( n := (\f.(\x.(f x)))
      in ((\f.(\x.(f (f x)))) n) )
80: (substitute into application)
85: substitute into function part (\f.(\x.(f (f x))))
86: substitute into argument n
95: evaluate( (\f.(\x.(f (f x)))) (\f.(\x.(f x))) )
52: evaluate( \f.(\x.(f (f x))) )
53: evaluate( \f.(\x.(f x)) )
70: substitute( f := (\f.(\x.(f x)))
      in \x.(f (f x)) )
80: (alpha-conversion if needed)
90: evaluate( \x.(g (g x)) ) -- where g = (\f.(\x.(f x)))

```

Conceptually, this trace shows that:

- The outermost call to `evaluate` sees an application, so it recursively evaluates the function and argument parts.
- For each beta-reduction, the interpreter calls `substitute` with the bound variable, the argument term, and the body of the abstraction, performing alpha-conversion where necessary.
- The recursive structure of the calls mirrors the structure of nested applications in the lambda-term, just as the recursive calls of `hanoi` mirror the recursive problem decomposition.

2.13.2 Questions

When experimenting with the interpreter, what is one example where the evaluation did not terminate in a normal form, and what does this tell you about the expressive power (and risks) of the untyped lambda calculus?

3 Essay

A unifying theme of this course is that *programming languages are mathematical objects*. From the MIU puzzle and abstract reduction systems (ARSs), through measure functions and termination proofs, to the lambda calculus, interpreters, and Lean, we repeatedly saw that programs can be understood as symbolic structures governed by precise rules of transformation. In this essay I focus on the lambda calculus and operational semantics as a way to synthesize the material and to connect it to the broader practice of software engineering.

The untyped lambda calculus, introduced by Church in the 1930s, is a tiny language with just variables, function abstraction, and application, yet it is computationally universal. It also forms the conceptual foundation of modern functional languages such as Haskell and ML.[1] In the homework and the interpreter project, I experienced this directly: Church numerals turned numbers into “iterate this function n times,” and boolean values became higher-order functions selecting a branch. What initially looks like a curiosity of logic turns out to be a powerful way to think about data and control flow in real languages.

Abstract reduction systems and the lambda calculus meet again in the idea of *operational semantics*. A small-step semantics defines a reduction relation $e \rightarrow e'$ describing how a program takes one tiny computational step at a time, while a big-step semantics relates an expression directly to its final value.[2] In the course, this showed up in two places: (1) drawing ARS graphs and proving properties such as confluence and unique normal forms; and (2) implementing an evaluator that repeatedly applies beta-reduction and capture-avoiding substitution. Seeing the same ideas in both a pen-and-paper proof style and in Python code made it clear that an interpreter is just “the semantics made executable.”

This perspective matters for software engineering. Real-world languages are far more complex than the toy calculi in class, but they are still ultimately defined by some kind of operational semantics. When we worry about whether an optimization preserves behaviour, whether a refactoring is semantics-preserving, or whether a compiler pass can introduce bugs, we are implicitly appealing to these formal models. Concepts like measure functions and termination arguments also scale up: they underlie loop variants, termination checkers, and tools that verify that recursive functions or protocols eventually make progress.

Personally, the most interesting and useful part of the course was seeing the tight connection between proofs and programs. Proving termination of Euclid’s algorithm or merge sort using a measure, then later reasoning about non-terminating lambda terms such as Ω , sharpened my intuition about when code is guaranteed to finish and when it might loop forever. Working in Lean made this even more concrete: logical rules, rewriting, and function definitions all became different faces of the same underlying notion of computation.

If I had to suggest improvements, they would be minor: perhaps a few more worked examples that explicitly connect each new formal concept (like confluence or fixed points) to a concrete bug or design decision in a popular language or library. Overall, though, the course already does a good job of showing that

programming-language theory is not just abstract math, but a practical toolkit for reasoning about software systems.

4 Evidence of Participation

4.1 *Machine Code Explained* (Computerphile)

Summary. In this video, Matt Godbolt explains what machine code really is and how it relates to the programs we write in high-level languages. He starts from the basic idea that a CPU ultimately only understands very small, fixed-size instructions encoded as binary numbers. Each instruction tells the processor to do something extremely simple, like move a value into a register, add two registers, or jump to another location in memory. Godbolt shows how these binary patterns are often written in hexadecimal for convenience, and how assembly language is just a human-readable layer on top of those numeric opcodes.

He then illustrates how a tiny program is laid out in memory as a sequence of instructions, and how the CPU steps through them one by one using the program counter. Along the way, he connects this to compilers: when we compile C or another high-level language, the compiler’s job is to translate our source into this low-level instruction sequence. The video emphasizes that many “mysterious” behaviors, like crashes, performance quirks, or odd jumps, make more sense once you understand what the machine is actually executing and how tightly constrained machine instructions really are.

Discussion Question. Once you understand that machine code is just a long sequence of simple instructions, how does that change the way you think about debugging or reasoning about the correctness of programs written in a high-level language?

Link: <https://www.youtube.com/watch?v=8VsiYWw9r48>

4.2 *Behind Mobile Game Design* (Olga Lazar)

Summary. In this talk, Olga Lazar shares her journey into mobile game design and offers a grounded look at what the day-to-day work of a game designer actually involves. She explains how designers sit at the intersection of art, programming, and business: they create core mechanics, tune difficulty and progression, and think constantly about player motivation and retention. Rather than just “coming up with ideas,” a lot of her work is iterative balancing, looking at player data, tweaking numbers, adjusting level layouts, and refining the feel of the game through repeated playtesting. She also emphasizes communication: designers have to translate high-level concepts into concrete specs that artists and engineers can implement, then negotiate trade-offs when technical or time constraints appear. Along the way, she talks about building a career in game design, including the importance of a portfolio, showing prototypes rather than just ideas, and being willing to start small (e.g., in QA or junior roles) and grow into more responsibility.

Discussion Question. Game designers often balance “what would be fun” with “what will keep players engaged and paying over time.” When those two goals conflict, how should a designer decide which side to prioritize, and what ethical limits should they place on their own design choices?

Link: <https://www.youtube.com/watch?v=VFkLXbSf7Co>

4.3 *Creating Your Own Programming Language* (Computerphile)

Summary. In the Computerphile video “Creating Your Own Programming Language,” Dr. Laurie Tratt demonstrates how surprisingly little code is required to build a tiny, working programming language. Instead of starting with heavy theory, he takes a practical, incremental approach: first choosing a minimal syntax (reverse Polish notation) to dodge complex parsing, then writing a small interpreter that uses a stack to evaluate expressions. From there, he gradually adds features, variables stored in a dictionary, basic control flow, and looping constructs, showing that core language ideas (state, branching, iteration) can emerge from a very compact design. Throughout, he emphasizes that languages are just programs like any other: they are made of concrete implementation choices and trade-offs, not magic. The video highlights how building

a toy language helps deepen understanding of real-world languages, interpreters, and compilers by making the underlying mechanisms visible and manipulable.

Discussion Question. When designing a small language for learning, is it better to keep the syntax and features extremely minimal, or to include a few “real-world” conveniences even if they complicate the implementation?

Link: <https://www.youtube.com/watch?v=Q2UDHY5as90>

4.4 Untyped Lambda Calculus Introduction

Summary. In this video, the instructor introduces the untyped lambda calculus as a tiny but surprisingly powerful “language of only functions.” He starts by explaining the basic syntax (variables, function abstractions, and applications) and then walks through how computation happens via substitution and step-by-step reduction (beta-reduction). Along the way he emphasizes free vs. bound variables and why we sometimes need to rename bound variables (alpha-conversion) to avoid accidental capture when substituting. With that foundation in place, he shows how everyday programming concepts emerge from pure functions: booleans as functions that choose between two arguments, logical operators like AND/OR built from those booleans, and natural numbers as Church numerals that represent “apply this function n times.”

The video then builds up more complex behavior: arithmetic on Church numerals, conditionals, pairs, and finally recursion via a fixed-point combinator (a Y-combinator-style construction). By the end, you see how a language with no built-in numbers, booleans, loops, or variables in the usual sense can still express all the usual programming patterns. The instructor frequently connects these ideas back to functional programming, pointing out how modern languages (like Haskell and ML-style languages) are grounded in the same calculus. Overall, the video reinforces a major theme from the course: that programs can be modeled as mathematical objects and that “real” languages are elaborate skins over a small, well-defined core.

Discussion Question. If the untyped lambda calculus can encode numbers, booleans, conditionals, and recursion using only functions, do you think that starting from such a tiny core makes it easier or harder to understand real-world programming languages later on?

Link: <https://www.youtube.com/watch?v=v5FBAbnLjUQ>

4.5 Procedural Generation in Games

Summary. In this video, Dr. Mike Cook explains procedural generation in games, how simple rules and algorithms can automatically create huge, varied worlds and content instead of designers building everything by hand. He starts with the idea that procedural content is generated by code rather than directly authored, and shows how small rule sets can expand into large, rich spaces of possibilities. He discusses examples like terrain, levels, and items, and contrasts random generation with structured generation, where constraints, weights, and rules are used to keep results fun and playable instead of chaotic.

Dr. Cook also talks about trade-offs: procedural systems can save time and create endless variety, but they can also feel repetitive or meaningless if the underlying rules are too shallow. Good procedural design requires thinking about player experience, what patterns players will notice, how difficulty flows, and how to avoid impossible or boring configurations. In the end, procedural generation is presented less as “magic randomness” and more as a design tool: a way to encode decisions and aesthetics into algorithms that act like co-designers alongside humans.

Discussion Question. If procedural generation is just “rules encoded as programs,” in what ways is designing a procedural generator similar to designing a programming language?

Link: <https://www.youtube.com/watch?v=G6ZHU0SXZDo>

5 Conclusion

Stepping back from the technical details, this course has quietly reshaped how I think about software engineering as a whole. Most of my other classes focus on building systems in existing languages and tools; CPSC 354 zoomed out and asked a more fundamental question: what *is* a programming language, really, and how do we know what our programs mean? From the MIU puzzle and abstract reduction systems, to lambda calculus, Church numerals, and the `fix` operator, the course consistently pushed me to see programs as mathematical objects that can be reasoned about in a precise way.

In the wider world of software engineering, this shows up in several important ways. Termination proofs and measure functions are the theoretical side of something every engineer cares about: will this algorithm always finish, and why? Abstract rewriting systems look like toy examples, but they are the same ideas behind compiler optimizations and program transformations: do different rewrite paths produce the same result, or can they get “stuck” in different places? Building and testing the lambda-calculus interpreter connected directly to language implementation and runtime systems. Writing grammars, designing precedence rules, and debugging ambiguous parses is exactly what real language designers and tool builders deal with when they design new syntaxes or write parsers and linters.

What I found most interesting and useful was the combination of *formal* reasoning and *concrete* artifacts. On the one hand, we used Lean to prove small arithmetic and logical facts step by step, seeing how much of “obvious” math actually rests on tiny lemmas like `add_zero` and `add_succ`. On the other hand, we wrote actual code: an interpreter, tests with `testing4b.py`, grammar files, and example programs like factorial, Fibonacci, and list-processing with `hd/tl`. That back-and-forth between proofs and running programs made the material feel less like pure theory and more like a toolkit I can bring into future work: thinking carefully about substitution and variable capture, structuring semantics, and designing tests that reflect the underlying math.

In terms of improvements, I do not have major complaints; the course worked well for me overall. If anything, a small enhancement would be to include a few more explicit “bridges” to mainstream languages and tools (for example, side-by-side comparisons with features in Python, Java, etc), just to help connect the abstract ideas even more directly to the languages most of us use day to day.

References

- [1] M. Mukund, *The (untyped) lambda calculus*, lecture notes, Chennai Mathematical Institute. Available at <https://www.cmi.ac.in/~madhavan/courses/pl2009/lecturenotes/lecture-notes/node81.html> (accessed December 4, 2025).
- [2] J. Ray, *Operational Semantics*, Loyola Marymount University. Available at <https://cs.lmu.edu/~ray/notes/opsem/> (accessed December 4, 2025).