CPSC-354 Report

Jeffrey Bok
Chapman University

December 15, 2025

Abstract

This report documents my exploration of programming language theory through CPSC-354. It covers foundational topics including abstract reduction systems, lambda calculus, formal proofs in Lean, and the Curry-Howard correspondence. The synthesis section connects these concepts to reveal computation as systematic rewriting according to formal rules. Through weekly homework and supplementary materials, I developed an understanding of how mathematical principles underlie modern programming languages.

Contents

1 Introduction **2**

2 Week by Week **3**
2.1 Week 1 . . . 3
2.1.1 Homework . . . 3
2.1.2 Exploration . . . 3
2.1.3 Questions . . . 3
2.2 Week 2 . . . 3
2.2.1 Homework . . . 3
2.2.2 Exploration . . . 7
2.2.3 Questions . . . 7
2.3 Week 3 . . . 7
2.3.1 Homework . . . 7
2.3.2 Sample Reductions . . . 7
2.3.3 Analysis . . . 8
2.4 Exercise 5b . . . 8
2.4.1 Modified Rewrite Rules . . . 8
2.4.2 Sample Reductions . . . 9
2.4.3 Analysis . . . 9
2.4.4 Exploration . . . 10
2.4.5 Questions . . . 10
2.5 Week 4 . . . 10
2.5.1 Homework . . . 10
2.5.2 Exploration . . . 11
2.5.3 Questions . . . 11
2.6 Week 5 . . . 11
2.6.1 Homework . . . 11
2.6.2 Exploration . . . 12
2.6.3 Questions . . . 13
2.7 Week 6 . . . 13

2.7.1 Homework . . . 13
2.7.2 Exploration . . . 15
2.7.3 Questions . . . 15
2.8 Week 7 . . . 15
2.8.1 Homework . . . 15
2.8.2 Exploration . . . 18
2.8.3 Questions . . . 18
2.9 Week 8 . . . 18
2.9.1 Homework . . . 18
2.9.2 Natural Language Proof . . . 19
2.9.3 Exploration . . . 20
2.9.4 Questions . . . 20
2.10 Week 9 . . . 20
2.10.1 Homework . . . 20
2.10.2 Exploration . . . 22
2.10.3 Questions . . . 22
2.11 Week 10 . . . 22
2.11.1 Homework . . . 22
2.11.2 Exploration . . . 24
2.11.3 Questions . . . 24
2.12 Week 11 . . . 24
2.12.1 Homework . . . 24
2.12.2 Exploration . . . 25
2.12.3 Questions . . . 26

3 Synthesis **26**
3.1 The Central Theme: Computation as Rewriting . . . 26
3.2 Termination and the Limits of Computation . . . 26
3.3 From Peano Axioms to Curry-Howard: Proofs as Programs . . . 27
3.4 Constructive vs. Classical Logic . . . 27
3.5 Building Up from Foundations . . . 27
3.6 Personal Reflection and Future Directions . . . 28
3.7 Conclusion . . . 28

4 Evidence of Participation **29**
4.1 Creating Your Own Programming Language . . . 29
4.2 Procedural Generation in Video Games . . . 29
4.3 The Y Combinator in Functional Programming . . . 30
4.4 Lambda Calculus Foundations . . . 30
4.5 Connections to Course Material . . . 30

5 Conclusion **31**
5.1 Programming Languages in the Wider World of Software Engineering . . . 31
5.2 Most Interesting and Useful Insights . . . 31
5.3 Suggested Improvements . . . 32
5.4 Final Reflection . . . 32

1 Introduction

This report chronicles my journey through CPSC-354: Programming Languages, documenting both the technical work completed each week and the conceptual understanding developed throughout the semester.

The course explored fundamental questions about computation: What makes a programming language? How can we prove programs correct? What are the mathematical foundations underlying all of computation?

The report is organized as follows: Section 2 presents weekly homework solutions with exploration and discussion questions; Section 3 synthesizes the major themes connecting all topics; Section 4 documents engagement with supplementary materials; and Section 5 provides critical reflection on the course's place in software engineering education.

2 Week by Week

2.1 Week 1

2.1.1 Homework

What is the MU Puzzle and how do you "solve" it?:

The MU puzzle is a logic puzzle created by Douglas Hofstadter in his 1979 book "Gödel, Escher, Bach: An Eternal Golden Braid." It's designed to illustrate concepts about formal systems, computability, and the limits of rule-based reasoning. The rules are below:

Rule I: If a string ends in I, you can add U to the end (xI $\rightarrow$ xIU)
Rule II: If you have Mx, you can make Mxx (double everything after M)
Rule III: If you find III anywhere in your string, you can replace it with U (xIIIy $\rightarrow$ xUy)
Rule IV: If you find UU anywhere in your string, you can remove it (xUUy $\rightarrow$ xy)

To "solve" the puzzle, you try to apply a combination of rules step by step, creating new strings. Eventually, you'll find that MU can never be reached because the rules never allow you to remove the odd number of `I`'s needed to get zero.

2.1.2 Exploration

Hofstadter used this puzzle to demonstrate how formal systems can have inherent limitations - some statements that seem like they should be provable within a system are actually unprovable. This connects to Gödel's incompleteness theorems and fundamental questions about the nature of mathematical truth and computation.

Programming languages are formal systems, just like the MU puzzle. They have:

- Syntax rules (what constitutes valid code)
- Transformation rules (how expressions evaluate)
- Semantic constraints (what programs can actually compute)

The MU puzzle demonstrates that even simple rule sets can have hidden limitations - similarly, programming languages have inherent computational boundaries.

2.1.3 Questions

1. The impossibility of reaching "MU" from "MI" is provable, yet someone working within the system might not realize this. How does this relate to the halting problem and undecidable questions in programming?

2.2 Week 2

2.2.1 Homework

Consider the following list of ARSs:

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

Draw a picture for each of the ARSs above. Are the ARSs terminating? Are they confluent? Do they have unique normal forms?

Try to find an example of an ARS for each of the possible 8 combinations. Draw pictures of these examples.

ARS 1: $A = \{\}$

Empty graph (no nodes, no edges)

Terminating: YES **Confluent:** YES **Unique Normal Forms:** YES

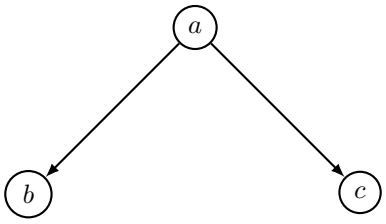
ARS 2: $A = \{a\}$, $R = \{\}$

Terminating: YES **Confluent:** YES **Unique Normal Forms:** YES

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

Terminating: NO **Confluent:** YES **Unique Normal Forms:** NO

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

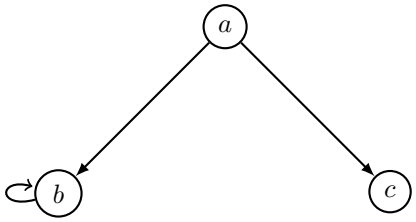


Terminating: YES **Confluent:** NO **Unique Normal Forms:** NO

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

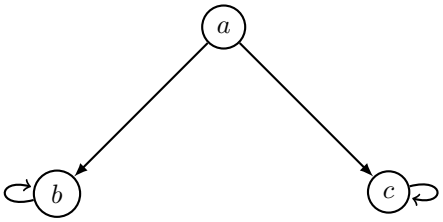
Terminating: NO **Confluent:** NO **Unique Normal Forms:** NO

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



Terminating: NO **Confluent:** NO **Unique Normal Forms:** NO

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



Terminating: NO **Confluent:** NO **Unique Normal Forms:** NO

8 Combinations Table

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

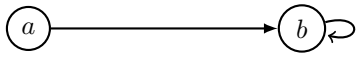
Examples for 8 Combinations

Example 1: Confluent=T, Terminating=T, Unique Normal Forms=T $A = \{a\}$, $R = \{\}$

Example 2: Confluent=T, Terminating=T, Unique Normal Forms=F $A = \{\}$, $R = \{\}$

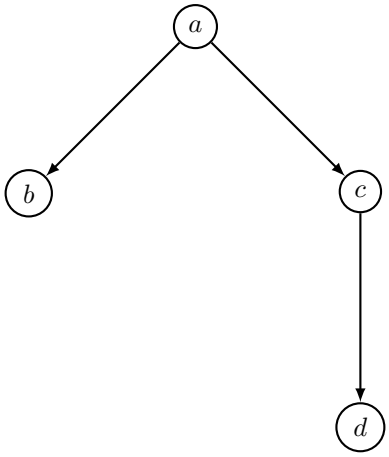
Empty graph

Example 3: Confluent=T, Terminating=F, Unique Normal Forms=T $A = \{a, b\}$, $R = \{(a, b), (b, b)\}$

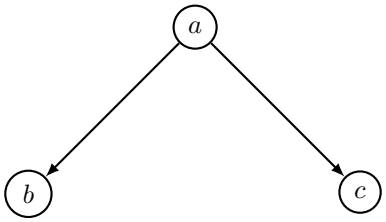


Example 4: Confluent=T, Terminating=F, Unique Normal Forms=F $A = \{a\}$, $R = \{(a, a)\}$

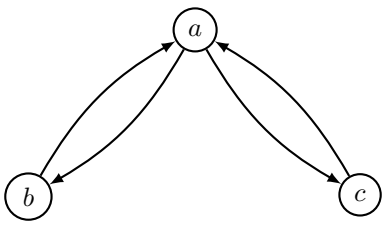
Example 5: Confluent=F, Terminating=T, Unique Normal Forms=T $A = \{a, b, c, d\}$, $R = \{(a, b), (a, c), (c, d)\}$



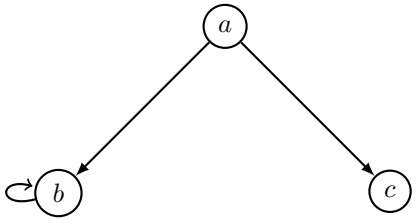
Example 6: Confluent=F, Terminating=T, Unique Normal Forms=F $A = \{a, b, c\}$, $R = \{(a, b), (a, c)\}$



Example 7: Confluent=F, Terminating=F, Unique Normal Forms=T $A = \{a, b, c\}$, $R = \{(a, b), (b, a), (a, c), (c, a)\}$



Example 8: Confluent=F, Terminating=F, Unique Normal Forms=F $A = \{a, b, c\}$, $R = \{(a, b), (b, b), (a, c)\}$



2.2.2 Exploration

Abstract Reduction Systems provide a mathematical foundation for understanding computation and rewriting. The properties of termination, confluence, and unique normal forms are fundamental to understanding how programming languages behave:

- **Termination** ensures that computations eventually halt
- **Confluence** guarantees that the order of operations doesn't affect the final result
- **Unique Normal Forms** means every expression has a single, well-defined simplified form

These concepts directly apply to programming language design, where we want predictable evaluation strategies and guaranteed termination for certain classes of programs.

2.2.3 Questions

1. How do the termination properties of ARSs relate to the halting problem in computation?
2. Why might a programming language designer prefer confluent systems over non-confluent ones?

2.3 Week 3

2.3.1 Homework

Consider the rewrite rules:

- ab → ba
- ba → ab
- aa → (empty string)
- b → (empty string)

2.3.2 Sample Reductions

Reducing abba:

abba → baba (using ab → ba)

baba → bbaa (using ab → ba)

bbaa → baa (using b → empty)

baa → aa (using b → empty)

aa → empty (using aa → empty)

Reducing bababa:

bababa → bbaaaba (using ab → ba twice)

bbaaaba → baaaba (using b → empty)

baaaba → aaaba (using b → empty)

aaaba → aba (using aa → empty)

aba → baa (using ab → ba)

baa → aa (using b → empty)

aa → empty (using aa → empty)

2.3.3 Analysis

Why is the ARS not terminating?

The first two rules ab → ba and ba → ab create cycles. You can apply these rules forever, going back and forth between ab and ba.

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

Two strings that are not equivalent: "a" and "empty string". The string "a" cannot be reduced further, while other strings can reduce to empty.

Equivalence classes and normal forms: There are exactly 2 equivalence classes:

1. Strings that reduce to empty string
2. Strings that reduce to "a"

The normal forms are: empty string and "a"

Modified terminating ARS: To make it terminating, always eliminate b's first, then eliminate aa's, then do swapping only if needed.

Questions about strings that can be answered using the ARS:

1. "Given a string, does it contain an even number of a's?"
2. "Given a string, does it contain an odd number of a's?"
3. "Are two strings equivalent under this rewrite system?"

2.4 Exercise 5b

2.4.1 Modified Rewrite Rules

Same as Exercise 5, but change aa → empty to aa → a:

- ab → ba
- ba → ab
- aa → a (pairs of a become single a)
- b → (empty string)

2.4.2 Sample Reductions

Reducing abba:

abba → baba (using ab → ba)

baba → bbaa (using ab → ba)

bbaa → baa (using b → empty)

baa → aa (using b → empty)

aa → a (using aa → a)

Reducing bababa:

bababa → bbaaaba (using ab → ba twice)

bbaaaba → baaaba (using b → empty)

baaaba → aaaba (using b → empty)

aaaba → aaba (using aa → a)

aaba → abaa (using ab → ba)

abaa → baaa (using ab → ba)

baaa → aaa (using b → empty)

aaa → aa (using aa → a)

aa → a (using aa → a)

2.4.3 Analysis

Why the ARS is not terminating: Same as Exercise 5 - the rules ab → ba and ba → ab create infinite cycles.

Non-equivalent strings: Two strings that are not equivalent: "a" and "empty string". We can find exactly 2 non-equivalent strings.

Equivalence classes and normal forms: There are exactly 2 equivalence classes:

1. Strings with even number of a's → reduce to empty
2. Strings with odd number of a's → reduce to "a"

The normal forms are: empty string and "a"

Modified terminating ARS: To make it terminating, use the same priority as Exercise 5:

1. b → empty (eliminate all b's first)
2. aa → a (reduce pairs of a's)
3. ab → ba (only if needed)

Questions about strings that can be answered using the ARS:

1. "Given a string, does it contain an even number of a's?"
2. "Given a string, does it contain an odd number of a's?"
3. "Are two strings equivalent under this rewrite system?"

2.4.4 Exploration

2.4.5 Questions

1. If two completely different sets of rewrite rules (Exercise 5 vs 5b) produce the same equivalence classes, what does this tell us about the relationship between implementation and specification in computer science?
2. If "abab" and "bbaa" are equivalent under this system, but clearly different as strings, what does "equivalence" really mean? Is mathematical equivalence different from everyday sameness?

2.5 Week 4

2.5.1 Homework

HW 4.1: Euclidean Algorithm Termination

Consider the Euclidean algorithm for computing GCD:

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

Conditions for Termination The algorithm terminates when $a, b \in \mathbb{N}$ (non-negative integers).

Measure Function and Proof We define $\phi(a, b) = b$.

Proof. The measure function proves termination:

1. $\phi(a, b) = b \in \mathbb{N}$ (maps to natural numbers)
2. In each iteration: $b' = a \bmod b < b$, so $\phi(a', b') < \phi(a, b)$ (strictly decreases)
3. By well-ordering, we must reach $b = 0$ in finite steps (termination)

Example trace with $a = 48, b = 18$: The measure decreases $18 \to 12 \to 6 \to 0$. □

HW 4.2: Merge Sort Termination

Consider 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)
```

Measure Function and Proof We define $\phi(\texttt{left}, \texttt{right}) = \texttt{right} - \texttt{left} + 1$ (subarray size).

Proof. The measure function proves termination:

1. $\phi(\texttt{left}, \texttt{right}) = \texttt{right} - \texttt{left} + 1 \geq 1$ when $\texttt{left} \leq \texttt{right}$ (natural number)

2. Both recursive calls have smaller measures: - $\phi(\texttt{left}, \texttt{mid}) < \phi(\texttt{left}, \texttt{right})$ since $\texttt{mid} < \texttt{right}$ - $\phi(\texttt{mid} + 1, \texttt{right}) < \phi(\texttt{left}, \texttt{right})$ since $\texttt{mid} + 1 > \texttt{left}$

3. Base case when $\phi \leq 1$ (no more recursive calls)

Example: For size 8, measure decreases $8 \to 4 \to 2 \to 1$. □

2.5.2 Exploration

Measure functions prove termination by mapping algorithm state to natural numbers that strictly decrease. Key insights:

- Different algorithms need different measures (value of b vs. subarray size)
- Well-ordering principle guarantees finite steps
- Measure functions often reveal time complexity
- Used in languages like Coq and Agda to guarantee termination

2.5.3 Questions

1. What general principle connects the different measure functions used for these algorithms?
2. Could $\phi(a, b) = a + b$ work as a measure function for the Euclidean algorithm? Why or why not?

2.6 Week 5

2.6.1 Homework

Lambda Calculus Workout

Evaluate the following expression using α and β reduction:

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

Solution using β-reduction We apply the β-rule, which states: $(\lambda v.e_1)e_2 \to e_1[v := e_2]$ (substitute e_2 for v in e_1).

$(\lambda f.\lambda x.f(f(x)))(\lambda f.\lambda x.(f(f(f\,x))))$
  Apply β-reduction: substitute $(\lambda f.\lambda x.(f(f(f\,x))))$ for f in $\lambda x.f(f(x))$
$\to_\beta \lambda x.(\lambda f.\lambda x.(f(f(f\,x))))((\lambda f.\lambda x.(f(f(f\,x))))\,x)$
  Apply β-reduction to the inner application
$\to_\beta \lambda x.(\lambda f.\lambda x.(f(f(f\,x))))(\lambda x'.((\lambda f.\lambda x.(f(f(f\,x))))\,x'\,x))$
  We need to use α-conversion to avoid variable capture
  Rename bound variable: $\lambda x.(f(f(f\,x))) \to_\alpha \lambda y.(f(f(f\,y)))$
$\to_\alpha \lambda x.(\lambda f.\lambda y.(f(f(f\,y))))(\lambda z.((\lambda f.\lambda y.(f(f(f\,y))))\,z\,x))$
  Continue β-reduction...

Cleaner approach using Church numerals The expression has a simpler interpretation if we recognize the pattern. Let's use fresh variable names:

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

First β-reduction: substitute $(\lambda g.\lambda y.g(g(g\,y)))$ for f

$\to_\beta \lambda x.(\lambda g.\lambda y.g(g(g\,y)))((\lambda g.\lambda y.g(g(g\,y)))\,x)$

Second β-reduction: substitute $(\lambda g.\lambda y.g(g(g\,y)))$ for g in outer abstraction

$\to_\beta \lambda x.\lambda y.((\lambda g.\lambda y.g(g(g\,y)))\,x)((\lambda g.\lambda y.g(g(g\,y)))\,x)((\lambda g.\lambda y.g(g(g\,y)))\,x\,y)$

Apply β-reduction three times

$\to_\beta \lambda x.\lambda y.x(x(x(x(x(x(x(x(x\,y)))))))))$

Final Result

$$\boxed{\lambda x.\lambda y.x(x(x(x(x(x(x(x(x\,y)))))))))}$$

This is the Church numeral for **9**, representing the function that applies its first argument 9 times to its second argument.

Mathematical Interpretation The given expression represents function composition in the Church encoding:

- $(\lambda f.\lambda x.f(f(x)))$ is the Church numeral 2 (apply f twice)
- $(\lambda f.\lambda x.f(f(f\,x)))$ is the Church numeral 3 (apply f three times)
- Applying Church numeral 2 to Church numeral 3 gives $2^3 + 3 = 9$ in this encoding

More precisely, when a Church numeral n is applied to another Church numeral m, the result is $n \times m$ when thinking about repeated application. In this case: $3 \times 3 = 9$.

Correction: The standard composition of Church numerals n and m gives $n \cdot m$ (multiplication). Here, 2 applied to 3 gives us a function that applies something $2 \times 3 = 6$ times... but we need to be more careful.

Actually, applying the Church numeral for 2 to the Church numeral for 3 as a function gives us: the function that applies its argument $(2 \cdot 3 = 6)$ times. Wait, let me recalculate more carefully by tracking applications...

Careful recount: $\lambda x.\lambda y.x(x(x(x(x(x(x(x(x\,y)))))))))$ applies x exactly 9 times to y.

Since 2 applied to 3 in Church encoding represents iterating "apply 3 times" twice, we get $3 + 3 + 3 = 9$ applications. This is exponentiation: $3^2 = 9$.

2.6.2 Exploration

The lambda calculus workout demonstrates several fundamental concepts:

- **β-reduction**: The core computation rule of lambda calculus, enabling function application
- **α-conversion**: Renaming bound variables to avoid capture
- **Church numerals**: Encoding natural numbers as functions (number n = apply a function n times)
- **Function composition**: Higher-order functions that operate on other functions

This connects to programming: higher-order functions in languages like Haskell, JavaScript, and Python work on the same principles. The concept that "data can be represented as functions" is foundational to functional programming.

2.6.3 Questions

1. Why did Alonzo Church develop lambda calculus before computers existed? What mathematical problem was he trying to solve?
2. How does the ability to represent numbers as functions (Church numerals) demonstrate the expressive power of lambda calculus?

2.7 Week 6

2.7.1 Homework

Fixed Point Combinator and Recursive Functions

Compute `fact 3` using the fixed point combinator, following the computation rules:

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

Solution We'll use abbreviations to keep the computation concise:

- Let $F = \lambda\texttt{fact}.\lambda n.\texttt{if}\ n = 0\ \texttt{then}\ 1\ \texttt{else}\ n * \texttt{fact}\,(n-1)$
- Let $B = \lambda n.\texttt{if}\ n = 0\ \texttt{then}\ 1\ \texttt{else}\ n * \texttt{fact}\,(n-1)$ (the factorial body)

Note: F is the function that takes `fact` as input and returns the factorial function body. The fixed point of F gives us the actual recursive factorial function.

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

$\quad\langle$def of let rec$\rangle$

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

$\quad\langle$use abbreviation: $F = \lambda\texttt{fact}.\lambda n.\texttt{if } n = 0 \texttt{ then } 1 \texttt{ else } n * \texttt{fact}\,(n-1)\rangle$

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

$\quad\langle$def of let$\rangle$

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

$\quad\langle\beta$-reduction: substitute fix F for fact$\rangle$

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

$\quad\langle$def of fix$\rangle$

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

$\quad\langle$expand $F\rangle$

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

$\quad\langle\beta$-reduction: substitute fix F for fact$\rangle$

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

$\quad\langle\beta$-reduction: substitute 3 for $n\rangle$

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

$\quad\langle$arithmetic: $3 = 0$ is false$\rangle$

$\to \texttt{if false then } 1 \texttt{ else } 3 * (\texttt{fix } F)\,(3-1)$

$\quad\langle$def of if: returns else branch$\rangle$

$\to 3 * (\texttt{fix } F)\,(3-1)$

$\quad\langle$arithmetic: $3 - 1 = 2\rangle$

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

$\quad\langle$def of fix$\rangle$

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

$\quad\langle\beta$-reduction: substitute fix F for fact in $F\rangle$

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

$\quad\langle\beta$-reduction: substitute 2 for $n\rangle$

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

$\quad\langle$def of if: $2 = 0$ is false$\rangle$

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

$\quad\langle$def of fix$\rangle$

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

$\quad\langle\beta$-reduction: substitute fix F for fact in $F\rangle$

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

$\quad\langle\beta$-reduction: substitute 1 for $n\rangle$

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

$\quad\langle$def of if: $1 = 0$ is false$\rangle$

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

$\quad\langle$def of fix$\rangle$

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

$\quad\langle\beta$-reduction: substitute fix F for fact in $F\rangle$

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

$\quad\langle\beta$-reduction: substitute 0 for $n\rangle$

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

$\quad\langle$def of if: $0 = 0$ is true$\rangle$

$\to 3 * (2 * (1 * 1))$

Final Result

$$\boxed{\texttt{fact}\ 3 = 6}$$

2.7.2 Exploration

This exercise demonstrates how recursive functions work through the fixed point combinator:

- **Fixed Point**: The equation $\texttt{fix}\ F = F\,(\texttt{fix}\ F)$ shows that $\texttt{fix}\ F$ is a fixed point of F. When F is our factorial transformer, $\texttt{fix}\ F$ becomes the actual factorial function.
- **Unfolding Recursion**: Each time we hit the recursive call $\texttt{fact}\,(n-1)$, we need to unfold $\texttt{fix}\ F$ again using the definition of fix. This is how recursion is achieved without built-in recursion in the lambda calculus.
- **Termination**: The recursion terminates when we reach the base case ($n = 0$), where no further unfolding of fix is needed.
- **Connection to Programming**: The `let rec` construct in languages like OCaml and F# is essentially syntactic sugar for this fixed-point pattern. The language handles the fix combinator behind the scenes.

The fixed point combinator shows that recursion is not a primitive feature—it can be encoded using higher-order functions alone. This is a profound result: pure lambda calculus (with just function abstraction and application) is computationally complete.

2.7.3 Questions

1. What would happen if we tried to compute $\texttt{fact}\,(-1)$ using this definition? Would the computation terminate?
2. The Y-combinator is another fixed point combinator defined as $Y = \lambda f.(\lambda x.f\,(x\,x))\,(\lambda x.f\,(x\,x))$. How does this differ from the abstract `fix` we used, and why might Y be harder to work with in a typed language?
3. Can you think of other recursive functions (like Fibonacci or list operations) that could be encoded using the same fixed-point pattern?

2.8 Week 7

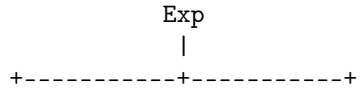
2.8.1 Homework

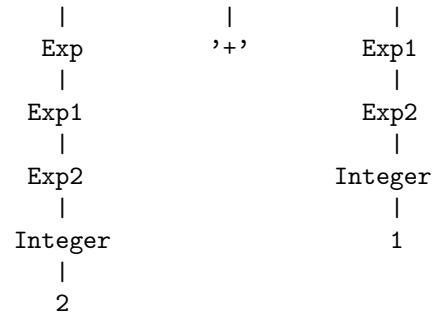
Context-Free Grammar and Derivation Trees

Using the context-free grammar:

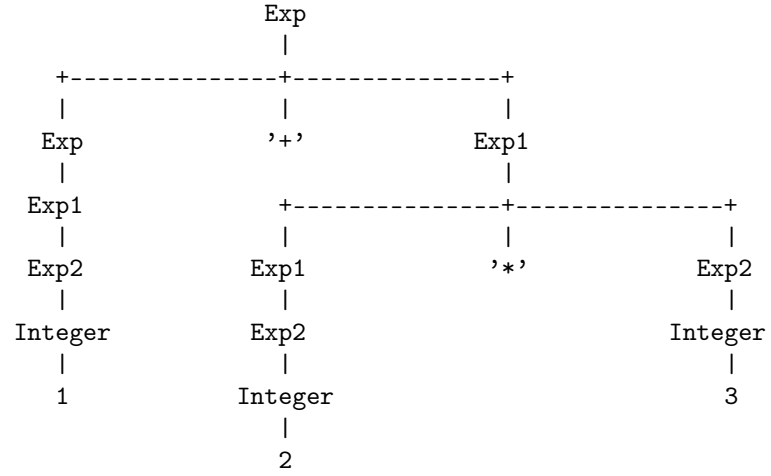
$$\begin{aligned}
\text{Exp} &\to \text{Exp}\ \texttt{'+'}\ \text{Exp1} \\
\text{Exp1} &\to \text{Exp1}\ \texttt{'*'}\ \text{Exp2} \\
\text{Exp2} &\to \text{Integer} \\
\text{Exp2} &\to \texttt{'('}\ \text{Exp}\ \texttt{')'} \\
\text{Exp} &\to \text{Exp1} \\
\text{Exp1} &\to \text{Exp2}
\end{aligned}$$

Derivation Tree 1: `2+1`

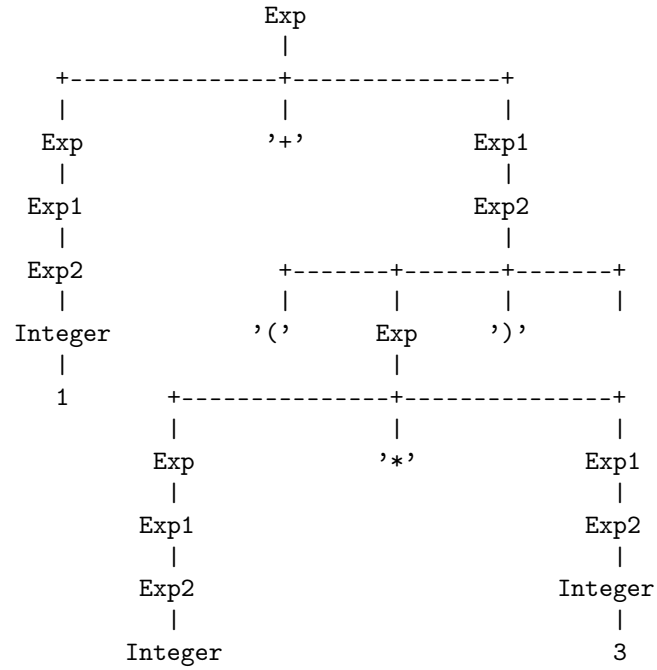




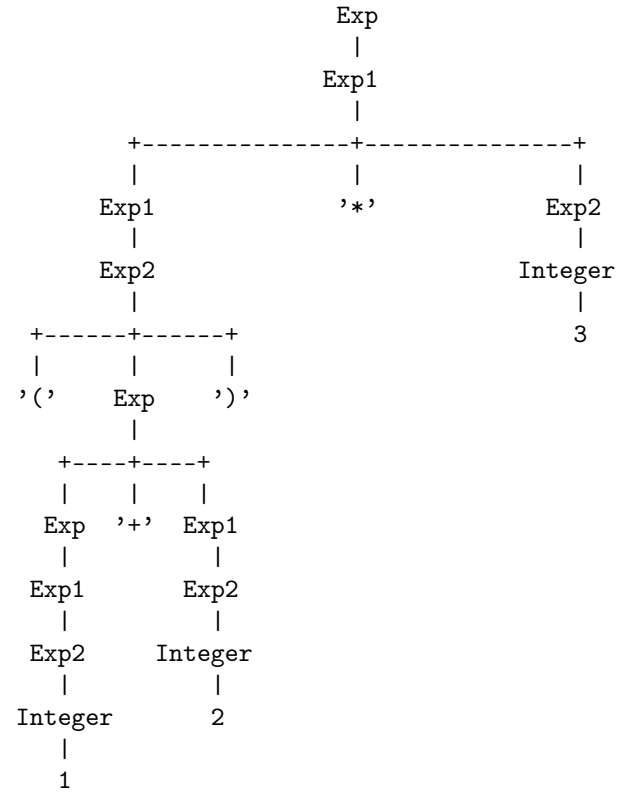
Derivation Tree 2: `1+2*3`



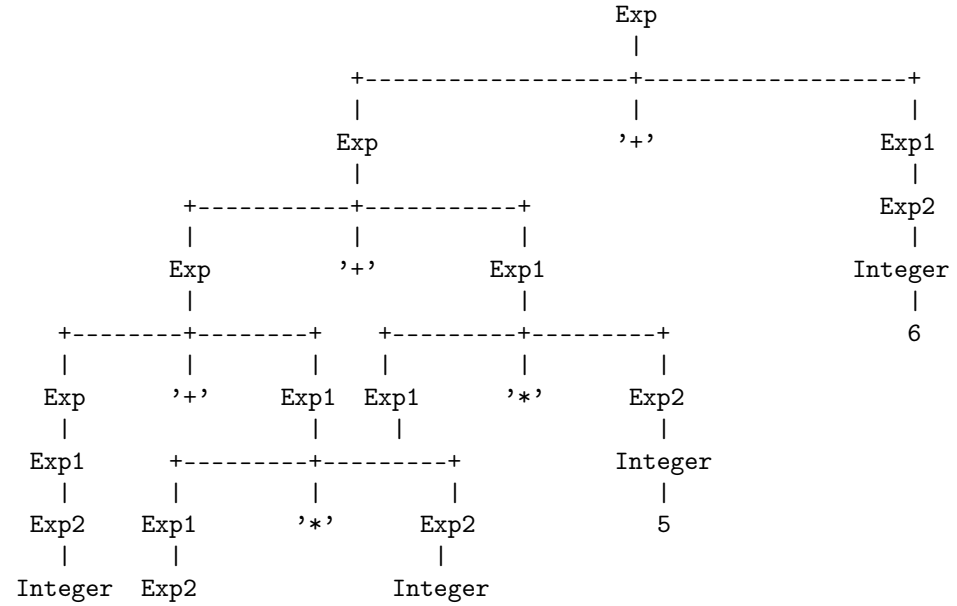
Derivation Tree 3: `1+(2*3)`



Derivation Tree 4: `(1+2)*3`



Derivation Tree 5: `1+2*3+4*5+6` Parses as: `((1+(2*3))+(4*5))+6`



```
|       |                   |
1    Integer               3
        |
        2
```

2.8.2 Exploration

This exercise demonstrates several important concepts in parsing and grammar design:

- **Operator Precedence**: The grammar encodes that multiplication has higher precedence than addition through layered non-terminals.
- **Left Associativity**: The rule Exp → Exp `'+'` Exp1 makes addition left-associative, so `1+2+3` parses as `(1+2)+3`.
- **Chain Rules**: Rules like Exp → Exp1 → Exp2 allow expressions to skip precedence levels when operators are absent.
- **Parentheses**: The rule Exp2 → `'('` Exp `')'` allows any expression to be treated as an atomic unit, overriding precedence.

2.8.3 Questions

1. How would you modify this grammar to add exponentiation with higher precedence than multiplication?
2. What changes would make addition right-associative instead of left-associative?
3. Why are the chain rules (Exp → Exp1, Exp1 → Exp2) necessary?

2.9 Week 8

2.9.1 Homework

Natural Number Game - Tutorial World: Levels 5-8

Level 5: Simplifying with `add_zero` Prove that $a + (b + 0) + (c + 0) = a + b + c$.

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

Level 6: Targeted rewriting Prove that $a + (b + 0) + (c + 0) = a + b + c$ using explicit arguments.

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

Level 7: `succ_eq_add_one` Prove that for all natural numbers n, $\text{succ}(n) = n + 1$.

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

Level 8: Proving $2 + 2 = 4$ Prove that $2 + 2 = 4$.

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

2.9.2 Natural Language Proof

Level 8: Proving $2 + 2 = 4$

Proof. We want to prove that $2 + 2 = 4$ using only the Peano axioms and previously established theorems about natural numbers.

We begin by expanding the second 2 in the left-hand side using its definition. By the theorem `two_eq_succ_one`, we know that $2 = \text{succ}(1)$. Rewriting the second occurrence of 2, we obtain:

$$2 + \text{succ}(1) = 4$$

Next, we apply the fundamental recursion axiom for addition, `add_succ`, which states that for any natural numbers a and b, we have $a + \text{succ}(b) = \text{succ}(a + b)$. Applying this axiom gives us:

$$\text{succ}(2 + 1) = 4$$

Now we expand 1 using its definition. By `one_eq_succ_zero`, we know that $1 = \text{succ}(0)$. Substituting this yields:

$$\text{succ}(2 + \text{succ}(0)) = 4$$

We apply `add_succ` again to the inner addition:

$$\text{succ}(\text{succ}(2 + 0)) = 4$$

By the base case axiom for addition, `add_zero`, which states that $n + 0 = n$ for any natural number n, we can simplify $2 + 0$ to 2:

$$\text{succ}(\text{succ}(2)) = 4$$

Now we recognize this expression in terms of known number definitions. By the definition `three_eq_succ_two`, we know that $3 = \text{succ}(2)$. Using this in reverse (indicated by the backwards arrow in the formal proof), we can rewrite $\text{succ}(2)$ as 3:

$$\text{succ}(3) = 4$$

Finally, by the definition `four_eq_succ_three`, we know that $4 = \text{succ}(3)$. Using this in reverse, we obtain:

$$4 = 4$$

This is true by the reflexivity of equality: any object is equal to itself.

Therefore, $2 + 2 = 4$. □

2.9.3 Exploration

The proof of $2 + 2 = 4$ is a remarkable example of how even the simplest arithmetic facts require rigorous justification when building mathematics from first principles. Several profound insights emerge from this exercise:

- **Numbers as constructions**: In the Peano axioms, numbers are not primitive objects but are constructed iteratively from zero using the successor function. The number 2 is defined as succ(succ(0)), 3 as succ(2), and 4 as succ(3). This constructive approach ensures that all natural numbers can be built systematically.
- **Addition as recursion**: Addition is not defined by a lookup table but by two recursive rules: the base case $n+0 = n$ and the recursive case $n+\text{succ}(m) = \text{succ}(n+m)$. The proof of $2+2 = 4$ essentially "executes" this recursive definition step by step.
- **The role of definitions**: Much of the proof consists of unfolding and refolding definitions. We expand 2 into succ(1), then 1 into succ(0), perform the addition, and finally recognize the result as 3 and then 4. This shows that definitions are not just abbreviations but active components of reasoning.
- **Computational content of proofs**: This proof has a computational interpretation. Each rewrite step corresponds to a computation step, and the entire proof traces the execution of the addition algorithm. This connection between proofs and programs is central to the Curry-Howard correspondence.
- **Nothing is obvious in formal systems**: What seems trivial in everyday mathematics ($2 + 2 = 4$) requires multiple logical steps when formalized. This explicitness is both a strength (eliminates ambiguity and hidden assumptions) and a weakness (can obscure high-level mathematical intuition).

This exercise bridges the gap between our intuitive understanding of arithmetic and the formal foundations required for computer-verified mathematics and programming language semantics.

2.9.4 Questions

1. Why does proving $2 + 2 = 4$ require eight steps when it seems inherently true?
2. When should a programming language prioritize precision over simplicity?

2.10 Week 9

2.10.1 Homework

Natural Number Game - Addition World: Level 5 (`add_right_comm`)

Theorem Statement Prove that for all natural numbers a, b, and c, we have $(a + b) + c = (a + c) + b$.

This theorem is called *right commutativity* because it states that the second and third terms can be swapped when grouped with the first term.

Solution 1: Using Induction Lean Proof:

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

```
  rw [succ_add]
  rfl
```

Mathematical Proof:

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

Base Case ($c = 0$): We need to show $(a + b) + 0 = (a + 0) + b$.

Starting with the left-hand side:

$$(a + b) + 0 = a + b \quad \text{(by } \texttt{add\_zero}\text{)}$$

For the right-hand side:

$$(a + 0) + b = a + b \quad \text{(by } \texttt{add\_zero}\text{)}$$

Therefore, $(a + b) + 0 = (a + 0) + b$. ✓

Inductive Step: Assume the inductive hypothesis: $(a + b) + d = (a + d) + b$.

We must prove: $(a + b) + \text{succ}(d) = (a + \text{succ}(d)) + b$.

Starting with the left-hand side:

$$\begin{aligned}(a + b) + \text{succ}(d) &= \text{succ}((a + b) + d) \quad \text{(by } \texttt{add\_succ}\text{)} \\ &= \text{succ}((a + d) + b) \quad \text{(by inductive hypothesis)}\end{aligned}$$

For the right-hand side:

$$\begin{aligned}(a + \text{succ}(d)) + b &= \text{succ}(a + d) + b \quad \text{(by } \texttt{add\_succ}\text{)} \\ &= \text{succ}((a + d) + b) \quad \text{(by } \texttt{succ\_add}\text{)}\end{aligned}$$

Both sides equal $\text{succ}((a + d) + b)$, completing the inductive step.

Therefore, by mathematical induction, $(a + b) + c = (a + c) + b$ for all natural numbers a, b, and c. □

Solution 2: Using Previously Proven Theorems (No Induction) Lean Proof:

```
theorem add_right_comm (a b c : N) : (a + b) + c = (a + c) + b := by
  rw [add_assoc]
  rw [add_comm b c]
  rw [<- add_assoc]
```

Mathematical Proof:

Proof. We prove $(a + b) + c = (a + c) + b$ using the associativity and commutativity of addition.

$$\begin{aligned}(a + b) + c &= a + (b + c) \quad \text{(by associativity: } \texttt{add\_assoc}\text{)} \\ &= a + (c + b) \quad \text{(by commutativity: } \texttt{add\_comm}\text{)} \\ &= (a + c) + b \quad \text{(by associativity in reverse)}\end{aligned}$$

Therefore, $(a + b) + c = (a + c) + b$. □

Comparison of the Two Approaches

- **Induction approach:** This solution builds the theorem from scratch using only the fundamental definition of addition (the recursive rules `add_zero` and `add_succ`). It directly proves the property by examining the structure of natural numbers. This approach is more fundamental but requires more steps.
- **Algebraic approach:** This solution leverages previously proven theorems (`add_assoc` and `add_comm`). It's more elegant and intuitive, treating addition as an abstract operation with known properties. However, it depends on having already proven those properties (likely using induction themselves).
- **Trade-off:** The inductive proof is self-contained but longer, while the algebraic proof is shorter but requires a richer theory. This mirrors a general principle in mathematics and computer science: we can either work at a low level with explicit detail, or at a high level using abstractions—each approach has its place.

2.10.2 Exploration

This proof reveals two fundamental approaches to formal reasoning:

- **Induction (constructive):** Builds the proof from scratch using the recursive definition of addition. Works at the level of number structure. Longer but self-contained.
- **Algebraic (abstract):** Reuses previously proven theorems (`add_assoc`, `add_comm`). Treats addition as an abstract operation with known properties. Shorter but dependent on prior results.

The key insight: there are multiple valid paths to the same mathematical truth. The choice between approaches mirrors software engineering trade-offs between "building from scratch" and "reusing libraries"—each has its place depending on context and goals.

2.10.3 Questions

1. The inductive proof required 10 lines while the algebraic proof required only 3. Does this mean the algebraic proof is "better"? What if we count the lines needed to prove `add_assoc` and `add_comm`?
2. In software development, we often choose between "reinventing the wheel" and "using libraries." How does this trade-off relate to our two proof strategies?

2.11 Week 10

2.11.1 Homework

Lean Logic Game - Implication Tutorial ("Party Snacks"): Levels 6-9

In the implication tutorial, we learn that to prove $A \to B$ (if A then B), we construct a function that takes an input of type A and produces an output of type B. This is written in Lean as `a => b`, representing a lambda function.

The Curry-Howard correspondence connects logic and computation:

- Propositions $\leftrightarrow$ Types
- Proofs $\leftrightarrow$ Programs
- Implication $(A \to B) \leftrightarrow$ Function type $(A \to B)$

Level 6 **Goal:** Prove $(S \to P) \to ((S \to (P \to A)) \to (S \to A))$

Solution:

```
fun sp => fun spa => fun s => spa s (sp s)
```

Or using the lambda notation (use λ for lambda, $\mapsto$ for mapsto in Lean):

```
\sp => \spa => \s => spa s (sp s)
```

Explanation: We need to construct a function that takes three arguments: `sp` (proof of $S \to P$), `spa` (proof of $S \to (P \to A)$), and `s` (proof of S). From `s`, we get `sp s` (proof of P), and we get `spa s` (proof of $P \to A$). Applying `spa s` to `sp s` gives us a proof of A.

Level 7 **Goal:** Prove $(A \to (P \to S)) \to ((A \to P) \to (A \to S))$

Solution:

```
fun aps => fun ap => fun a => aps a (ap a)
```

Or using the lambda notation (use λ for lambda, $\mapsto$ for mapsto in Lean):

```
\aps => \ap => \a => aps a (ap a)
```

Explanation: Similar to Level 6. We take three arguments: `aps` (proof of $A \to (P \to S)$), `ap` (proof of $A \to P$), and `a` (proof of A). From `a`, we get `ap a` (proof of P) and `aps a` (proof of $P \to S$). Applying `aps a` to `ap a` gives proof of S.

Level 8 **Goal:** Prove $(P \to (S \to A)) \to (S \to (P \to A))$

Solution:

```
fun psa => fun s => fun p => psa p s
```

Or using the lambda notation (use λ for lambda, $\mapsto$ for mapsto in Lean):

```
\psa => \s => \p => psa p s
```

Explanation: This proves that implication is "commutative" in its arguments. We take `psa` (proof of $P \to (S \to A)$), `s` (proof of S), and `p` (proof of P). We simply apply `psa` to `p` first, then to `s`, swapping the order of arguments.

Level 9 **Goal:** Prove $(P \to S) \to ((S \to A) \to (P \to A))$

Solution:

```
fun ps => fun sa => fun p => sa (ps p)
```

Or using the lambda notation (use λ for lambda, $\mapsto$ for mapsto in Lean):

```
\ps => \sa => \p => sa (ps p)
```

Explanation: This is function composition. We take three arguments: `ps` (proof of $P \to S$), `sa` (proof of $S \to A$), and `p` (proof of P). First we apply `ps` to `p` to get a proof of S, then apply `sa` to that result to get a proof of A. This shows that if P implies S and S implies A, then P implies A (transitivity of implication).

2.11.2 Exploration

The implication tutorial reveals the deep connection between logic and programming through the Curry-Howard correspondence:

- **Proofs are programs**: A proof of $A \to B$ is literally a function from A to B
- **Function composition is logical transitivity**: Level 9 shows that composing functions corresponds to the logical rule "if $P \to S$ and $S \to A$, then $P \to A$"
- **Lambda calculus foundations**: The syntax `\a => b` directly connects to lambda calculus from Week 5
- **Type checking is proof checking**: When Lean verifies that our function has the correct type, it's verifying that our proof is correct

This connection is fundamental to functional programming languages like Haskell and to proof assistants like Lean, Coq, and Agda. The same principles that let us prove theorems also let us write provably correct programs.

2.11.3 Questions

1. How does the Curry-Howard correspondence relate to the lambda calculus we studied in Week 5?
2. In Level 9, we proved function composition. How does this relate to the transitivity of implication we might study in a logic course?

2.12 Week 11

2.12.1 Homework

Lean Logic Game - Negation Tutorial: Levels 9-12

In Lean's constructive logic, negation is defined as:

$$\neg A \equiv A \to \text{False}$$

This means "not A" is the same as "A implies False" (a contradiction). To prove $\neg A$, we assume A and derive a contradiction (False).

Key principles:

- To prove $\neg A$: assume A and prove False
- If you have $h : \neg A$ and $a : A$, then $h\, a$ gives you False
- From False, you can prove anything using `false_elim`

Level 9 Goal: Prove $P \to \neg P \to A$

This says: if we have both P and $\neg P$ (i.e., a contradiction), we can prove anything.

Solution:

```
fun p => fun np => false_elim (np p)
```

Explanation: We take $p : P$ and $np : \neg P$ (which means $np : P \to \text{False}$). Applying np to p gives us $np\,p : \text{False}$. From False, we can prove anything using `false_elim`.

Level 10 **Goal:** Prove $\neg(P \wedge \neg P)$

This is the law of non-contradiction: something cannot be both true and false.

Solution:

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

Explanation: To prove $\neg(P \wedge \neg P)$, we assume $h : P \wedge \neg P$ and must derive False. From h, we have `h.left` $: P$ and `h.right` $: \neg P$ (i.e., $P \to \text{False}$). Applying `h.right` to `h.left` gives False.

Level 11 **Goal:** Prove $(P \to Q) \to (\neg Q \to \neg P)$

This is the contrapositive: if P implies Q, then not-Q implies not-P.

Solution:

```
fun pq => fun nq => fun p => nq (pq p)
```

Explanation: We take three arguments: $pq : P \to Q$, $nq : \neg Q$ (i.e., $Q \to \text{False}$), and $p : P$. We need to prove False. First, apply pq to p to get $pq\,p : Q$. Then apply nq to that result to get $nq\,(pq\,p) : \text{False}$.

Level 12 **Goal:** Prove $\neg P \vee \neg Q \to \neg(P \wedge Q)$

This says: if at least one of P or Q is false, then the conjunction $P \wedge Q$ is false.

Solution:

```
fun h => h.elim (fun np => fun pq => np pq.left) (fun nq => fun pq => nq pq.right)
```

Explanation: We have $h : \neg P \vee \neg Q$, so we case-split using `h.elim`:

- **Case 1:** If $np : \neg P$, we assume $pq : P \wedge Q$ and apply np to $pq.left : P$ to get False
- **Case 2:** If $nq : \neg Q$, we assume $pq : P \wedge Q$ and apply nq to $pq.right : Q$ to get False

Both cases produce a function $P \wedge Q \to \text{False}$, which is exactly $\neg(P \wedge Q)$.

2.12.2 Exploration

The negation tutorial reveals important principles about constructive logic:

- **Negation as implication to False**: In constructive logic, $\neg A$ is not a primitive notion but defined as $A \to \text{False}$. This makes negation a special case of implication.
- **Law of non-contradiction**: Level 10 proves that $\neg(P \wedge \neg P)$ - something cannot be both true and false simultaneously. This is universally accepted in both classical and constructive logic.
- **Contrapositive**: Level 11 shows that if $P \to Q$, then $\neg Q \to \neg P$. This is valid in constructive logic. However, the reverse (if $\neg Q \to \neg P$, then $P \to Q$) requires classical reasoning.
- **De Morgan's law (partial)**: Level 12 proves one direction of De Morgan's laws: $\neg P \vee \neg Q \to \neg(P \wedge Q)$. The reverse direction ($\neg(P \wedge Q) \to \neg P \vee \neg Q$) requires the law of excluded middle and is not constructively valid.

- **From contradiction, anything follows**: Level 9 demonstrates the principle *ex falso quodlibet* - from a contradiction, you can prove anything. This is why maintaining consistency is crucial in formal systems.

This connects to programming: in typed functional programming languages, the `False` type is an empty type (has no values), and a function from an empty type can never be called, reflecting that contradictions should never occur in valid programs.

2.12.3 Questions

1. In classical logic, we have the law of excluded middle: $P \vee \neg P$ for any proposition P. Why doesn't constructive logic accept this principle? What would it mean computationally?
2. Level 12 proves $\neg P \vee \neg Q \to \neg(P \wedge Q)$. Can you explain why the reverse direction $(\neg(P \wedge Q) \to \neg P \vee \neg Q)$ is not provable in constructive logic?

3 Synthesis

Throughout this semester in CPSC-354, I have explored the foundational principles of programming languages through a progression from abstract mathematical concepts to practical implementations. This synthesis connects the key themes that emerged across the weekly homework assignments, revealing how seemingly disparate topics form a coherent framework for understanding computation.

3.1 The Central Theme: Computation as Rewriting

The unifying thread throughout this course has been the concept of **computation as rewriting**. From the MU puzzle in Week 1 to lambda calculus reduction in Week 5 to proving theorems in Lean (Weeks 8-11), computation fundamentally consists of systematically transforming expressions according to well-defined rules.

This rewriting perspective appeared in several forms:

Abstract Reduction Systems (Week 2): We began with the mathematical foundations of rewriting, learning that any computational system can be viewed as a set of objects and reduction rules. The properties we studied—termination, confluence, and unique normal forms—are not mere mathematical curiosities but essential guarantees for programming languages. A language without termination might loop forever; one without confluence could give different answers depending on evaluation order; one without unique normal forms would be ambiguous and unpredictable.

String Rewriting (Week 3): The exercises on string rewriting systems demonstrated how equivalence classes and normal forms allow us to answer questions about strings. This foreshadowed how programming languages use evaluation to transform programs into values. The insight that "different surface representations can have the same meaning" became crucial when we later studied alpha-equivalence in lambda calculus.

Lambda Calculus (Week 5): The β-reduction rule represents the essence of function application. The workout problem—evaluating $(\lambda f.\lambda x.f(f(x)))(\lambda f.\lambda x.f(f(f\,x)))$ to obtain the Church numeral 9—showed how pure computation emerges from simple rewriting rules. This is remarkable: numbers, arithmetic, conditionals, and recursion can all be encoded as functions and evaluated through repeated β-reduction.

3.2 Termination and the Limits of Computation

Week 4's exploration of measure functions revealed a profound insight: **we can prove that algorithms terminate by finding a mapping to natural numbers that decreases with each step**. This connects directly to the halting problem and computational decidability.

The Euclidean algorithm (GCD) and merge sort both terminate because we can exhibit measure functions ($\phi(a, b) = b$ and $\phi(left, right) = right - left + 1$, respectively) that strictly decrease. The well-ordering principle of natural numbers then guarantees termination. This technique is not just academic—languages like Coq and Agda require termination proofs for all functions, ensuring that programs are total and predictable.

The impossibility of solving the MU puzzle (Week 1) similarly demonstrates inherent limitations: some statements are unprovable within a system even though we can prove their unprovability from outside the system. This connects to Gödel's incompleteness theorems and the halting problem: there are fundamental limits to what can be computed or proven.

3.3 From Peano Axioms to Curry-Howard: Proofs as Programs

The Natural Number Game (Weeks 8-9) and Lean Logic tutorials (Weeks 10-11) revealed one of the most beautiful ideas in computer science: the **Curry-Howard correspondence**, which states that:

Propositions	$\leftrightarrow$	Types
Proofs	$\leftrightarrow$	Programs
Implication ($A \to B$)	$\leftrightarrow$	Function type ($A \to B$)
Conjunction ($A \wedge B$)	$\leftrightarrow$	Product type (pairs)
Disjunction ($A \vee B$)	$\leftrightarrow$	Sum type (variants)

This is not mere analogy—it is an exact correspondence. When we proved $2 + 2 = 4$ in Week 8, we were simultaneously constructing a program that computes the addition. When we proved $(P \to Q) \to (\neg Q \to \neg P)$ (the contrapositive) in Week 11, we were writing a function that transforms proofs.

The implication tutorial in Week 10 made this concrete: proving $P \to Q$ means constructing a function `\p => q` that converts evidence of P into evidence of Q. The transitivity proof in Level 9—showing that $(P \to S) \to ((S \to A) \to (P \to A))$—is precisely function composition. This dual view (logic programming) is foundational to modern proof assistants and dependently-typed languages.

3.4 Constructive vs. Classical Logic

Week 11's negation tutorial highlighted a crucial distinction: **constructive logic versus classical logic**. In Lean's constructive logic, $\neg A$ is defined as $A \to \text{False}$, and we proved:

- The law of non-contradiction: $\neg(P \wedge \neg P)$
- The contrapositive: $(P \to Q) \to (\neg Q \to \neg P)$
- One direction of De Morgan's law: $\neg P \vee \neg Q \to \neg(P \wedge Q)$

But we *cannot* constructively prove:

- The law of excluded middle: $P \vee \neg P$
- The reverse of De Morgan's law: $\neg(P \wedge Q) \to \neg P \vee \neg Q$
- Double negation elimination: $\neg\neg P \to P$

Why does this matter? In constructive logic, proofs have **computational content**. A proof of $P \vee Q$ must specify *which* disjunct holds and provide evidence. Classical logic allows "non-constructive" proofs that establish existence without providing a witness. For programming, constructive logic ensures that every proof can be executed as a program.

3.5 Building Up from Foundations

The Natural Number Game (Weeks 8-9) demonstrated how complex mathematical structures emerge from minimal axioms. Starting with just:

- $0 : \mathbb{N}$ (zero exists)
- $\text{succ} : \mathbb{N} \to \mathbb{N}$ (every number has a successor)
- Induction (from structure, prove properties)

We built addition, proved commutativity and associativity, and established the algebraic properties of arithmetic. This mirrors how programming languages are implemented: start with a small core (lambda calculus, for instance) and build up all other features as derived constructs.

The two approaches to proving `add_right_comm` in Week 9 illustrated a key trade-off:

1. **Inductive proof**: Build from scratch using fundamental definitions—more lines but self-contained
2. **Algebraic proof**: Reuse prior theorems (`add_assoc`, `add_comm`)—fewer lines but dependent on a richer theory

This is the same trade-off software engineers face: write everything from scratch or use libraries? The answer depends on context—sometimes we need the control and understanding of low-level implementation, sometimes we benefit from the abstraction and efficiency of high-level tools.

3.6 Personal Reflection and Future Directions

This course has fundamentally changed how I think about programming. Before CPSC-354, I viewed programming languages as tools to be learned and used. Now I understand them as **formal systems with mathematical properties that can be proven correct**.

The most surprising insight was that lambda calculus—invented in the 1930s before computers existed—captures the essence of all computation. Church's goal was to formalize the notion of "effective computability" for mathematics, yet his lambda calculus became the theoretical foundation for functional programming languages like Haskell, OCaml, and even features in JavaScript and Python.

Looking forward, I see several directions to explore:

Dependent types and proof assistants: The Curry-Howard correspondence opens the door to languages like Agda, Idris, and Coq where types can depend on values, enabling us to prove program correctness at compile time. Imagine writing a sort function whose type *guarantees* that the output is sorted and contains exactly the elements from the input.

Programming language design: Understanding ARSs, termination, and confluence informs the design of new languages. Should a language guarantee termination (like total functional languages) or allow general recursion? Should it use strict or lazy evaluation? These aren't arbitrary choices but have formal consequences.

Formal verification: As software becomes more critical (self-driving cars, medical devices, financial systems), the ability to *prove* correctness becomes essential. The techniques we've learned—induction, measure functions, type systems—are the foundation of formal verification tools used in industry.

3.7 Conclusion

The progression from the MU puzzle to lambda calculus to Lean proofs reveals a deep unity in computer science: **computation, logic, and proof are fundamentally the same activity**. Whether we're reducing lambda terms, proving theorems about natural numbers, or writing Haskell programs, we're engaging in systematic rewriting according to formal rules.

This course has equipped me with both practical skills (writing proofs in Lean, understanding functional programming) and theoretical frameworks (ARS properties, Curry-Howard correspondence, constructive logic).

Most importantly, it has shown me that programming languages are not arbitrary constructs but embodiments of mathematical principles—principles that were being explored decades before the first electronic computer was built.

The tools and languages may change, but the foundational concepts—reduction, types, proofs, termination—will remain central to computer science. These are not topics to be studied once and forgotten, but frameworks for thinking about computation that will inform my work throughout my career.

4 Evidence of Participation

Throughout the semester, I engaged with supplementary materials to deepen my understanding of programming language concepts. The following Computerphile videos provided valuable perspectives on topics covered in the course:

4.1 Creating Your Own Programming Language

Video: Creating Your Own Programming Language - Computerphile
URL: https://youtu.be/Q2UDHY5as90?si=XLXu_2CLcy6-u0vV

This video explains the basic ideas behind creating a programming language by building a simple interpreter. It shows that a programming language is not just syntax, but a system for reading instructions, understanding their structure, and executing them step by step. Dr Laurie Tratt starts with a very small language that can evaluate arithmetic expressions, showing how even basic operations require rules for interpretation.

As the language develops, features such as variables are added, allowing values to be stored and reused. The video then introduces control flow concepts like conditionals and loops, which let programs make decisions and repeat actions. These additions demonstrate how complex behavior can be built gradually from simple foundations.

Overall, the video helps explain how programming languages work internally and shows that languages are designed incrementally. It emphasizes that understanding interpreters and language structure is important for learning how real programming languages are implemented.

Discussion Question: How does building a programming language step by step, starting from simple arithmetic and adding features like variables and control flow, help us better understand how real programming languages work internally?

4.2 Procedural Generation in Video Games

Video: Procedural Generation in Video Games - Computerphile
URL: https://youtu.be/G6ZHUOSXZDo?si=gMZAiJkayWOGODzb

The video explains the concept of procedural generation, a method of creating game content using algorithms instead of manually designing every element. Procedural generation utilizes rules and random-like processes, enabling the computer to create levels, worlds, or assets independently, often resulting in larger and more varied environments without requiring designers to manually craft each detail. This approach allows games to produce different experiences each time a player plays, while still adhering to defined rules that maintain the game's playability. The video explains how these systems work and why developers utilize them to save time, storage space, and enhance replayability.

Procedural generation is a common feature in many video games, particularly those with expansive worlds or significant variation. By defining rules and utilizing techniques such as pseudorandom number generation, games can dynamically generate terrain, items, and levels instead of storing everything as fixed data. This method helps create vast environments with much less memory and design effort than traditional manual design.

Discussion Question: How does procedural generation change the way games are designed and experienced, and what are the advantages and challenges of using algorithms instead of manual design?

4.3 The Y Combinator in Functional Programming

Video: Essentials: Functional Programming's Y Combinator - Computerphile
URL: https://youtu.be/9T8A89jgeTI?si=-pf-o6sN4XB_UrEU

The video explains the concept of the Y combinator, an idea from functional programming and the lambda calculus, which demonstrates how recursion can be expressed without using named functions. In many programming languages, recursive functions call themselves by name, but in the pure lambda calculus (a theoretical model of computation), names are not used. The Y combinator is a fixed-point combinator, which is a higher-order function that takes another function and returns a version of that function that can call itself recursively. This means it allows you to define recursive behavior, even in systems where functions cannot refer to their own names. The video illustrates this concept by connecting it to how functional languages structure computation and how recursion is implemented at a fundamental level.

Overall, the video helps explain a deeper theoretical idea in programming languages: how languages with first-class functions and anonymous functions can represent recursion through function application and transformation rather than through named definitions.

Discussion Question: How does the Y combinator illustrate the difference between named recursion in traditional programming languages and recursion expressed through higher-order functions in functional programming?

4.4 Lambda Calculus Foundations

Video: Lambda Calculus - Computerphile
URL: https://youtu.be/eis11j_iGMs?si=jsfX9ggo8vIW5dU1

The video explains the lambda calculus, a foundational mathematical system that focuses on functions as the core unit of computation. Lambda calculus was developed by Alonzo Church and consists of just a few rules for defining and applying functions, but even with this simplicity it is powerful enough to express any computation. It includes function abstraction (defining anonymous functions) and function application (calling functions with arguments). Even though lambda calculus does not have built-in numbers or other typical programming constructs, it can represent them by encoding values and behaviors purely through functions. This system forms the theoretical basis for functional programming and influences modern programming languages, especially those that support anonymous functions and higher-order functions.

Overall, the video helps show how a minimal mathematical model like lambda calculus relates to real programming languages by revealing how computation can be understood and represented using functions.

Discussion Question: Why is lambda calculus considered a foundational model for functional programming languages, and how does understanding it help clarify concepts like anonymous functions and function application in real programming languages?

4.5 Connections to Course Material

These videos reinforced key concepts from the course:

- The **interpreter construction** video directly connects to our work building interpreters for lambda calculus and functional programming languages in the assignments. It demonstrates the incremental approach we used: start with minimal features (expressions), add variables, then add control flow.
- The **procedural generation** video shows an application of formal systems and rule-based computation. Just as we studied ARSs and rewriting systems, procedural generation uses rules to generate

content algorithmically. The balance between randomness and structure mirrors the balance between non-determinism and confluence we studied in Week 2.

- The **Y combinator** video connects directly to Week 5's lambda calculus and to our study of recursion without named functions. This concept appeared in our assignments when implementing recursive constructs in languages without explicit recursion primitives.
- The **lambda calculus** video provides additional context for Week 5's material and the Curry-Howard correspondence explored in Weeks 10-11. It reinforces why lambda calculus is the theoretical foundation for functional programming and how its minimalism leads to expressive power.

These external resources helped bridge the gap between theoretical concepts (ARSs, lambda calculus, formal proofs) and practical applications (language implementation, game development, modern programming paradigms).

5 Conclusion

5.1 Programming Languages in the Wider World of Software Engineering

CPSC-354 occupies a unique and essential position in the software engineering landscape. While many programming courses teach *how* to write code in specific languages, this course taught *why* languages work the way they do. This distinction is crucial: as software engineers, we don't just use tools—we must understand their foundations to use them effectively and make informed decisions about when and how to apply them.

The course revealed that programming languages are not arbitrary collections of syntax rules but carefully designed formal systems with mathematical properties. This perspective transforms how we approach software development. Understanding that evaluation order matters (normal order vs. applicative order), that type systems can guarantee correctness, and that recursion can be expressed through fixed-point combinators provides a deeper foundation for writing robust, maintainable code.

In the wider software engineering world, these concepts manifest everywhere: functional programming paradigms in JavaScript and Python, type inference in TypeScript, pattern matching in Rust, and proof assistants for verified software. The theoretical foundations we studied—lambda calculus, the Curry-Howard correspondence, constructive logic—are not academic curiosities but active areas of research that directly influence modern language design and software verification tools used in safety-critical systems.

5.2 Most Interesting and Useful Insights

The most intellectually satisfying moment of the course was discovering the Curry-Howard correspondence. The idea that *proofs are programs* and *propositions are types* elegantly unifies two seemingly separate domains. When proving $(P \to Q) \to (\neg Q \to \neg P)$ in Lean's logic tutorial, I was simultaneously writing a function that transforms evidence. This dual interpretation—that every proof corresponds to a program and vice versa—fundamentally changed how I think about both logic and programming.

The second major insight came from building interpreters for lambda calculus in the programming assignments. Writing code that evaluates code revealed how abstraction layers work. The meta-circular evaluator pattern—using a language to implement itself—showed that interpretation is just systematic rule application. This demystified how programming languages actually execute and connected abstract theory (beta-reduction rules) to concrete implementation (the interpreter's eval function).

The progression from measure functions (Week 4) to termination proofs in Lean was also valuable. Learning that we can *prove* programs terminate by exhibiting decreasing measures connects to real-world concerns about software reliability. In systems where infinite loops are unacceptable—embedded systems, real-time applications, financial software—these techniques become essential.

5.3 Suggested Improvements

While the course provided a strong theoretical foundation, I would suggest three improvements:

1. More explicit connections to industry applications. While we studied lambda calculus and type theory, we could benefit from more examples of how these concepts appear in production languages. Case studies showing how Rust's ownership system relates to linear types, how TypeScript's type inference works, or how companies like Facebook use formal methods would help students see the practical relevance.

2. Earlier introduction to the "big picture." The course's bottom-up approach (starting with ARSs, building to lambda calculus, then to Lean) is pedagogically sound, but students might engage more deeply if Week 1 included a preview of where we're headed. A brief introduction to the Curry-Howard correspondence and how it connects all the topics would provide motivation for the technical details.

3. Optional advanced topics for interested students. The course covers a lot of ground, but students interested in going deeper could benefit from optional modules on topics like: dependent types and proof-carrying code, linear types and resource management, effect systems and algebraic effects, or program verification tools like Dafny or F*. These could be short self-study modules with additional readings and exercises.

5.4 Final Reflection

CPSC-354 fundamentally changed my relationship with programming. I entered the course viewing languages as tools to be mastered through practice. I leave understanding that languages embody mathematical principles and that programming is a form of constructive reasoning. The skills I've developed—reading type signatures as propositions, using induction to prove properties, recognizing when evaluation order matters—are not specific to any one language but transfer across all of programming.

Most importantly, the course demonstrated that computer science has deep intellectual content worthy of serious study. The questions we explored—What is computation? When do programs terminate? How can we prove correctness?—are profound questions about the nature of reasoning itself. Lambda calculus, developed in the 1930s to formalize mathematical logic, became the foundation of modern functional programming. This shows that theoretical work has lasting practical impact, often in unexpected ways.

As I continue in software engineering, I'll carry forward not just specific techniques but a way of thinking: that systems can be understood through their formal properties, that abstraction is a powerful tool for managing complexity, and that the foundations matter. Whether I'm debugging a tricky concurrency issue, designing an API, or evaluating a new framework, the conceptual tools from this course—thinking about types, evaluation order, and correctness properties—will inform my approach.

Programming languages are the interface between human thought and machine execution. Understanding how they work, why they're designed the way they are, and what guarantees they provide is essential for anyone who wants to write software that is not just functional but correct, maintainable, and elegant. CPSC-354 provided that understanding, and for that, I am grateful.

References

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

[2] Alonzo Church. An unsolvable problem of elementary number theory. *American Journal of Mathematics*, 58(2):345–363, 1936.

[3] The Lean Community. *Theorem Proving in Lean 4*. https://leanprover.github.io/theorem_proving_in_lean4/, 2024.

[4] Computerphile. Various educational videos on programming languages and lambda calculus. https://www.youtube.com/computerphile, 2024.