CPSC-354 Report

Jonathan Karam
Chapman University

October 29, 2025

Abstract

This report is a collection of notes, discussions, and homework solutions for CPSC 354. Homework 1 discusses the MIU puzzle and proves why `MI` cannot become `MU`. Homework 2 introduces abstract rewriting systems (ARS), includes TikZ diagrams, and classifies systems by termination, confluence, and unique normal forms. Homework 3 studies a rewriting system over $\{a, b\}$ and explores its equivalence classes and termination. Homework 4 proves termination of two classic algorithms using measure functions. Homework 5 evaluates a small λ-calculus "workout" using α- and β-rules, with notes connecting these ideas to modern languages. Homework 6 develops fixpoints and recursion: we compute `fact 3` step-by-step with the rules for `fix`, `let`, and `let rec`, and give a Y-combinator encoding of factorial. Week 7 covers grammars, parse trees, and ASTs with full derivation trees. Week 8 connects the Natural Number Game (Tutorial World) to natural-language proofs and Lean tactics, with scripts for Levels 5–8 and a short class-notes discussion.

Contents

1 Introduction

This report documents my learning week by week. Each section includes my notes and discussions of the lecture topics, followed by my homework solutions. The purpose is to show understanding, practice writing in LaTeX, and explain the material in my own words.

2 Week 1: The MIU Puzzle

2.1 Notes and Discussion

The first week introduced the MIU puzzle, created by Douglas Hofstadter. It is a formal system that begins with the word `MI` and has four rules. The puzzle asks whether it is possible to reach `MU`. This is important because it shows how formal systems can be analyzed with invariants, a key concept in logic and computer science.

2.2 Homework

Rules:

1. If a string ends with `I`, add a `U`.
2. If a string starts with `M`, double the part after `M`.
3. Replace `III` with `U`.
4. Remove `UU`.

Why `MI` cannot become `MU`: Let $\#I(w)$ be the number of `I`'s in w. Initially $\#I(\texttt{MI}) = 1$. Each rule preserves $\#I(w) \pmod 3$:

- Rule 1: no effect.
- Rule 2: doubles the count, $1 \mapsto 2$, never $0 \pmod 3$.
- Rule 3: subtracts 3, same remainder.
- Rule 4: no effect.

Thus $\#I(w) \pmod 3$ is invariant. Since `MU` has $\#I = 0$, it cannot be reached.

Deeper explanation: The invariant argument proves that all reachable words have I-count $\equiv 1$ or $2 \pmod 3$, never 0. The first letter M never disappears, so all reachable words start with M. Another way: define a homomorphism $h(M) = 0, h(U) = 0, h(I) = 1 \pmod 3$. Every rule preserves h, so $h(\texttt{MI}) = 1$ and $h(\texttt{MU}) = 0$. Contradiction.

3 Week 2: Abstract Rewriting Systems

3.1 Notes and Discussion

This week focused on abstract rewriting systems. An ARS consists of a set A and a relation R. We ask whether rewriting always stops (termination), whether different rewrite paths can rejoin (confluence), and whether elements end up in unique normal forms.

3.2 Homework

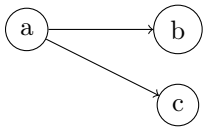
Same problem statement as before: draw each ARS and decide termination, confluence, and whether the system has *unique normal forms for all elements* (UN-for-all).

Examples (drawings unchanged)

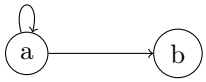
1: $A = \{\}, R = \{\}$ (empty system)

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

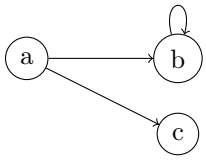
3: $A = \{a\}, R = \{(a, a)\}$



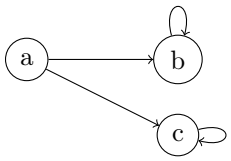
4: $A = \{a, b, c\}, R = \{(a, b), (a, c)\}$



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



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



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

Classification (solutions rewritten)

We now use **UN-for-all**: every element must have a (unique) normal form.

ARS	Terminating	Confluent	UN-for-all	Why
1	Yes	Yes	Yes	Nothing rewrites (vacuous).
2	Yes	Yes	Yes	a is already normal and unique.
3	No	Yes	No	$a \to a$ forever; a has no NF.
4	Yes	No	No	$a \to b$, $a \to c$ (two distinct NFs).
5	No	Yes	Yes	$a \to b$, b normal; both end at b.
6	No	No	No	b loops; $a \to c$ NF, not joinable with b.
7	No	No	No	b, c loop; no element has an NF.

Remarks.

- Non-termination alone does not rule out UN-for-all (see 5), but any element without an NF breaks it (3,6,7).
- In terminating ARSs, confluence $\Rightarrow$ UN-for-all (every element reaches a unique NF).
- Item 4 is the canonical "diamond failure": two different normal forms reachable from the same source.

4 Week 3: Strings over {a,b}

4.1 Notes and Discussion

We studied rewriting rules over alphabet $\{a, b\}$. This shows how simple systems can classify strings into equivalence classes. We also saw how orienting rules one way can make a system terminating without changing equivalence.

4.2 Homework 3

Exercise 5 rules:

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

Examples: `abba` $\to$ `aa` $\to \varepsilon$. `bababa` $\to$ `aaa` $\to$ `a`.

Not terminating because swaps loop. Equivalence classes: even number of a's $\to \varepsilon$, odd $\to a$. Normal forms: ε, a. Fix: orient $ba \to ab$ only.

Exercise 5b rules:

$$ab \leftrightarrow ba, \quad aa \to a, \quad b \to \varepsilon$$

Examples: `abba` $\to$ `a`. `bababa` $\to$ `a`.

Equivalence: all b vanish, runs of a collapse to one a. Classes: $\{\varepsilon, a\}$. Fix: orient swaps, keep $aa \to a$ and $b \to \varepsilon$.

5 Week 4: Measure Functions and Termination

5.1 Notes and Discussion

This week we looked at *measure functions* to prove that algorithms terminate. A measure maps the state of a computation to a value in a well-founded set (usually the natural numbers) and must *strictly decrease*

with every loop iteration or recursive call. Because there are no infinite descending chains in a well-founded set, the computation must eventually stop.

Scope note: Here we define a measure and show it strictly decreases.

Class notes invariant proof that Euclid returns gcd**.**

- **Key fact.** $\gcd(a, b) = \gcd\big(b,\ a - qb\big)$ for any integer q.
- **Update.** Writing $a = qb + r$ with $r = a \bmod b$, the step $(a, b) \mapsto (b, r)$ keeps the gcd unchanged.
- **Invariant.** Before each loop test: $\gcd(a, b) = \gcd(a_0, b_0)$.
- **End.** When $b = 0$, $\gcd(a_0, b_0) = \gcd(a, 0) = |a|$, the returned value.

5.2 Homework 4

HW 4.1

Problem. Considering

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

Under which conditions does this algorithm always terminate? Find a measure function and prove termination.

Answer. This is the classical Euclidean algorithm for $\gcd(a, b)$. It always terminates provided that

$$a, b \in \mathbb{Z} \quad \text{and} \quad b \geq 0,$$

and we interpret `%` as the mathematical remainder with $0 \leq a \bmod b < b$ for $b > 0$. If the inputs are negative, replace them once by $|a|, |b|$; this preserves gcd and satisfies the condition.

Measure function. Let the program state be the pair (a, b) with $b \geq 0$. Define

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

Strict decrease. In an iteration with $b > 0$ we set $b' = a \bmod b$. By definition of remainder,

$$0 \leq b' < b.$$

Therefore $\mu(a', b') = b' < b = \mu(a, b)$. Hence μ strictly decreases on every loop step.

Well-foundedness and termination. μ maps states to the natural numbers with the usual $<$, which is well founded. Because μ strictly decreases whenever the loop body executes, the loop can execute only finitely many times; eventually $b = 0$ and the algorithm returns. □

Remark 5.1. The algorithm not only terminates; it returns $\gcd(a, b)$. This follows from the class note invariant above and $\gcd(a, 0) = |a|$.

HW 4.2

Problem. Consider the fragment of merge sort:

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

Prove that

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

is a measure function for `merge_sort`.

Answer. We show that every recursive call is made with a *strictly smaller* measure and that the measure is bounded below by 0.

Well-founded codomain. $\varphi(left, right)$ is a nonnegative integer whenever $left \le right$, so the codomain is $\mathbb{N}$ with the usual $<$ (well-founded).

Base case. If $left \ge right$, the function returns immediately. In this case $\varphi \le 1$, and there are no recursive calls.

Decrease for recursive calls. Assume $left < right$ and let $n = \varphi(left, right) = right - left + 1 \ge 2$. With

$$mid = \left\lfloor \frac{left + right}{2} \right\rfloor,$$

the first recursive call is on $[left,\, mid]$ and the second on $[mid + 1,\, right]$. Their measures are

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

Because $left \le mid < right$, we have

$$1 \le \varphi(left, mid) \le \left\lceil \frac{n}{2} \right\rceil < n, \qquad 1 \le \varphi(mid + 1, right) \le \left\lfloor \frac{n}{2} \right\rfloor < n.$$

Thus each recursive call receives strictly smaller measure than n.

Conclusion. Every chain of recursive calls strictly decreases φ and cannot be infinite in $\mathbb{N}$. Therefore `merge_sort` terminates. □

6 Week 5: Lambda Calculus Foundations

6.1 Notes and Discussion

- **Self-application and computation.** Terms like $(\lambda x.\, xx)(\lambda x.\, xx)$ show that functions can take *code as data*. This enables iteration/recursion encodings (fixed points) and explains why untyped λ can express non-termination.
- **Where λ shines.** Useful for building *minimal cores*: interpreters, proof assistants, type checkers, compiler IRs, and reasoning about higher-order functions. It models substitution, scope, closures, and evaluation precisely.

- **Composition and numerals.** Church numerals encode iteration: $\mathbf{n}\, f\, x = f^n x$. Function composition $(f \circ g)$ corresponds to multiplying numerals: $(\mathbf{m} \circ \mathbf{n})\, f\, x = f^{mn} x$. Plain application behaves like *exponentiation in reverse order*: $\mathbf{m}\,\mathbf{n} = \mathbf{n}^{\mathbf{m}}$ (apply $\mathbf{m}$ times), so $\mathbf{2}\,\mathbf{3} = \mathbf{9}$ and $\mathbf{3}\,\mathbf{2} = \mathbf{8}$.
- **Confluence (Church–Rosser).** If a term reduces to two results, there exists a common reduct. Meaning: evaluation order doesn't affect the final normal form (when it exists). Languages borrow this idea to justify equational reasoning and optimizations.
- **From λ to languages.** Add types (safety), effects (state, I/O), data types, and evaluation strategy to get modern languages. Typed λ-calculi (System F, Hindley–Milner) underlie ML/Haskell; effect systems/monads model side effects.
- **Haskell links.** Purity, first-class functions, laziness (normal-order–like), algebraic data types, and type inference all line up with typed λ-calculus. Monads/Applicatives are structured ways to compose computations.
- **Names and α-conversion.** Variables need not be single letters; names are irrelevant up to α-equivalence (consistent renaming). Modern languages reflect this via lexical scope, hygienic macros, and compiler renaming to avoid capture.
- **The role of α in practice.** Compilers implement renaming and fresh-name generation to maintain scope hygiene (e.g., SSA form, macro expansion).
- **Termination and type systems.** Untyped λ allows non-termination (Y-combinator). Total languages (Coq/Agda) *forbid* general recursion by checking termination via structural/lexicographic and multivariate measures; complex patterns (exponential, multi-dimension decrease) are handled with well-founded orders and sized types.
- **Original intent.** Church invented λ for the foundations of mathematics and to study effective computability—leading to Church–Turing thesis.
- **Recursion vs loops.** Prefer recursion for recursive structure or immutability; prefer loops for in-place iteration or where tail recursion isn't optimized.

6.2 Homework 5: Lambda Calculus Workout

Problem

Evaluate (practice α and β rules; match parentheses):

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

Solution

Let

$$\mathbf{two} \equiv \lambda f.\lambda x.\, f(fx), \qquad \mathbf{three} \equiv \lambda f.\lambda x.\, f(f(fx)).$$

Then

$$\mathbf{two}\ \mathbf{three} \xrightarrow{\beta} \lambda x.\, \mathbf{three}\,(\mathbf{three}\, x).$$

Compute:

$$\mathbf{three}\, x \xrightarrow{\beta} \lambda y.\, x(x(x\, y)), \qquad \mathbf{three}\,(\lambda y.\, x(x(x\, y))) \xrightarrow{\beta} \lambda z.\, x^9 z.$$

Therefore

$$\boxed{\lambda x.\lambda z.\, x^9 z}$$

which is the Church numeral **nine**. (And $\mathbf{3}\,\mathbf{2} = \mathbf{8}$ because $\mathbf{m}\,\mathbf{n} = \mathbf{n}^{\mathbf{m}}$.)

7 Week 6: Fixpoints and Recursion

7.1 Class Notes & Discussion

This section responds concisely to discussion questions from class.

1. **Reducing inside larger expressions and step rules for `fix`, `let`, `if`.** We use *evaluation contexts*: reduce inside the syntactic position allowed by the strategy (e.g., call-by-value reduces the scrutinee of `if`, the bound expression of `let`, and the argument to `fix` until it is a λ). Rules:

$$\text{fix } F \;\to\; F(\text{fix } F), \qquad \text{let } x = e_1 \text{ in } e_2 \;\to\; (\lambda x.\, e_2)\, e_1, \qquad \text{if true then } e_1 \text{ else } e_2 \to e_1.$$

   Contexts justify each inner step when `fix`, `let`, or `if` appears inside a bigger term.

2. **Fixpoint combinator and recursion.** A combinator Y with $YF = F(YF)$ provides a value YF that is a fixed point of F. If F encodes one step of a recursive definition, YF is its recursive solution. Languages often bake this in as a primitive `fix`.

3. **Factorial with Y instead of `fix`.**

$$\text{fact} \;\equiv\; Y\,(\lambda f.\, \lambda n.\, \text{if } n = 0 \text{ then } 1 \text{ else } n \cdot f(n-1)).$$

4. **Typed vs untyped.** In typed, strongly normalizing calculi, general Y breaks termination; practical languages use explicit recursion with typing restrictions, or domain-theoretic semantics for effects.

7.2 Homework 6

We use these computation rules:

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

Problem A: Compute `fact 3`

Start with

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

Derivation

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

$\rightarrow$ **(def of let rec)**

let fact $=$ fix $(\lambda f.\ \lambda n.$ if $n = 0$ then 1 else $n * f(n-1))$ in fact 3

$\rightarrow$ **(def of let)**

$(\lambda \text{fact. fact } 3)\ \big(\text{fix } F\big)$ where $F \equiv \lambda f.\ \lambda n.$ if $n = 0$ then 1 else $n * f(n-1)$

$\rightarrow$ **(β; substitute** fix F **for fact)**

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

$\rightarrow$ **(def of fix)**

$\big(F(\text{fix } F)\big)\ 3$

$\rightarrow$ **(β)**

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

$\rightarrow$ **(β; substitute** 3 **for** n**)**

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

$\rightarrow$ **(def of if)**

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

$\rightarrow$ **(def of fix)**

$3 * \big(F(\text{fix } F)\big)\ 2$

$\rightarrow$ **(β)**

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

$\rightarrow$ **(β; substitute** 2 **for** n**)**

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

$\rightarrow$ **(def of if)**

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

$\rightarrow$ **(def of fix)**

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

$\rightarrow$ **(β)**

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

$\rightarrow$ **(β; substitute** 1 **for** n**)**

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

$\rightarrow$ **(def of if)**

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

$\rightarrow$ **(def of fix)**

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

$\rightarrow$ **(β)**

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

$\rightarrow$ **(β; substitute** 0 **for** n**)**

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

$\rightarrow$ **(def of if)**

$3 * \big(2 * (1 * 1)\big) \ = \ 3 * 2 * 1 \ = \ \boxed{6}$.

Problem B: Y-combinator factorial and one unfold

Let

$$Y \equiv \lambda g.\ (\lambda x.\ g(x\,x))(\lambda x.\ g(x\,x)), \quad G \equiv \lambda f.\ \lambda n.\ \text{if } n = 0 \text{ then } 1 \text{ else } n * f(n-1).$$

Define fact $\equiv YG$. Then

$$\text{fact} \ = \ YG \ \to \ G(YG) \ = \ \lambda n.\ \text{if } n = 0 \text{ then } 1 \text{ else } n * (YG)(n-1),$$

which is exactly one recursive "unfold".

Remark 7.1. In total/strongly normalizing typed calculi, Y cannot be defined without sacrificing termination. Practical languages provide explicit recursion or `fix`, often with typing restrictions.

8 Week 7: Grammars, Parse Trees, and ASTs

8.1 Class Notes & Discussion

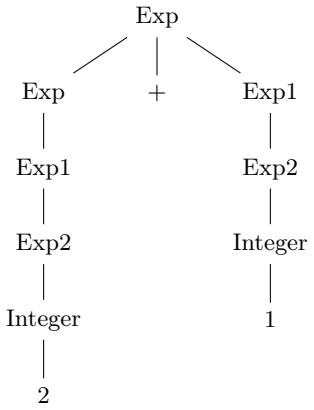
- **Fixed-point combinators and performance.** Using Y/`fix` introduces an extra unfolding step ($YF \to F(YF)$). Named recursion or `let rec` generally performs better since the compiler can optimize it directly without creating closures on each call.
- **LLMs as metaprograms.** Large language models can be viewed as metaprograms that translate semantics (intent or meaning) into code. However, they do not replace the traditional compiler pipeline—semantic interpretation, parsing, optimization, and execution still occur afterward.
- **Lisp and syntax readability.** Lisp's simple S-expression syntax gives programmers direct access to abstract syntax, making metaprogramming and macro systems easier to implement. The trade-off is reduced human readability but increased ease of parsing for machines.
- **Ambiguous grammars.** Ambiguity in grammars can make parsing inconsistent; precedence and associativity rules often disambiguate.
- **Parse trees vs. ASTs.** Parse trees include every token; ASTs compress away redundant structure (e.g., parentheses) to expose semantics for analysis and code generation.

8.2 Homework 7: Context-Free Grammar and Derivation Trees

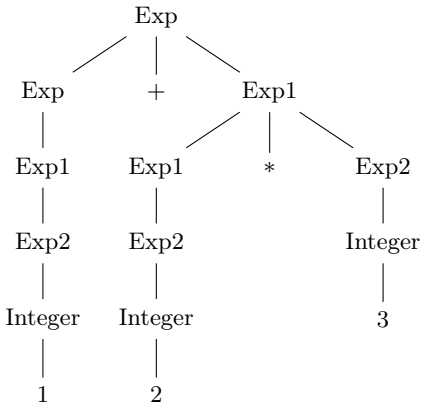
Grammar:

$$\begin{array}{rcl} \text{Exp} & \to & \text{Exp} \ + \ \text{Exp1} \ \mid \ \text{Exp1} \\ \text{Exp1} & \to & \text{Exp1} \ * \ \text{Exp2} \ \mid \ \text{Exp2} \\ \text{Exp2} & \to & \text{Integer} \ \mid \ (\,\text{Exp}\,) \end{array}$$

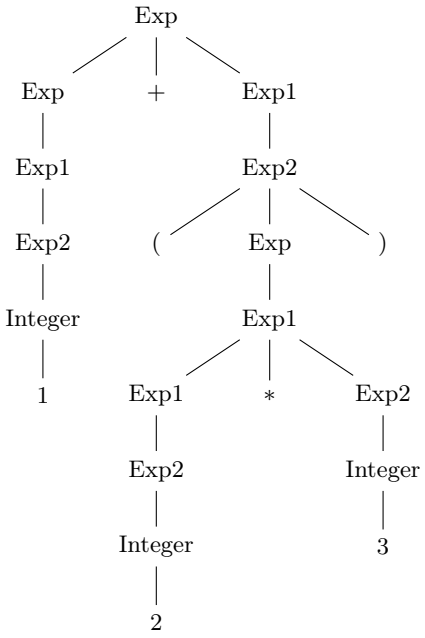
(a) `2+1`


Exp
Exp
+
Exp1
Exp1
Exp2
Exp2
Integer
Integer
1
2

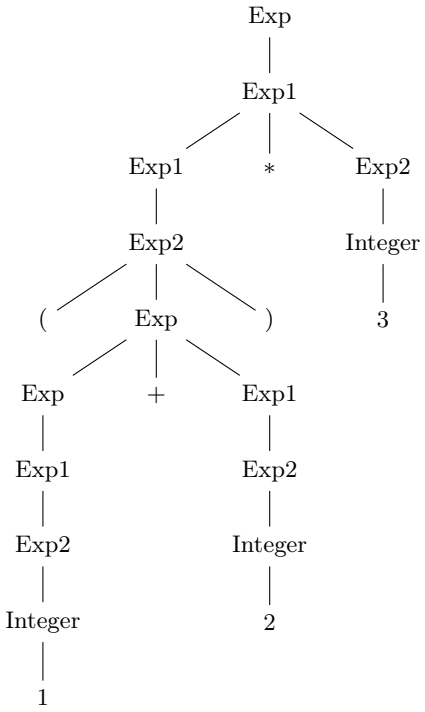
(b) `1+2*3`


Exp
Exp
+
Exp1
Exp1
Exp1
*
Exp2
Exp2
Exp2
Integer
Integer
Integer
3
1
2

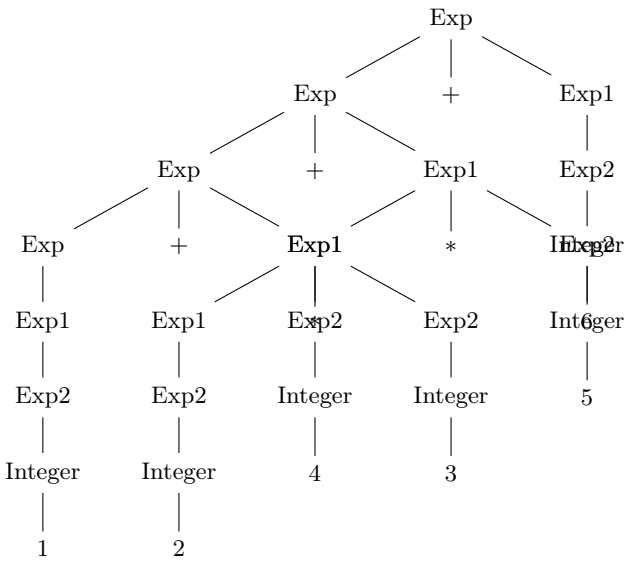
(c) `1+(2*3)`



(d) `(1+2)*3`



(e) `1+2*3+4*5+6`



9 Week 8: Natural Number Game (Tutorial World)

9.1 Notes & Setup

This task connects informal English proofs with formal Lean steps. In the Tutorial World we use facts like

$$a + 0 = a, \qquad a + \mathrm{succ}(b) = \mathrm{succ}(a + b), \qquad 1 = \mathrm{succ}(0),$$

and finish goals by reflexivity (`rfl`) after rewriting.

9.2 Selected Level Solutions (5–8)

Below are minimal tactic scripts matching the in-game steps. (They are illustrative—other equivalent scripts also pass.)

Level 5 — Adding zero (right).

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

Level 6 — Careful rewriting.

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

Level 7 — Show a + 1 = succ a.

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

Level 8 — Compute 2 + 2 = 4.

```
-- Use 2 = succ 1 and 1 = succ 0, then unfold addition by add_succ
nth_rewrite 2 [two_eq_succ_one]
rw [add_succ 2]
nth_rewrite 1 [one_eq_succ_zero]
-- continue rewrites until the goal closes by rfl
```

9.3 Class Notes & Discussion

- **English vs. Lean proofs.** The Lean script mirrors an English proof by repeatedly replacing equals by equals (rewrite) until both sides are syntactically identical (`rfl`).
- **Core facts used.** Right identity of addition ($a + 0 = a$), right recursion ($a + \text{succ}(b) = \text{succ}(a + b)$), and numeral definitions ($1 = \text{succ}(0)$, $2 = \text{succ}(1)$).
- **Tactics as rules of inference.** Each `rw` corresponds to applying an equality as a substitution rule; the order of rewrites matters to expose the shape needed for the next lemma.
- **Why recursion on the right?** Defining + by recursion on the second argument gives the simple law above and simplifies induction proofs on the right operand. Recursing on the left would require different lemmas (e.g., $\text{succ}(a) + b = \text{succ}(a + b)$) and different induction patterns.
- **Equality reasoning pattern.** Normalize numerals → unfold by recursive law `add_succ` repeatedly → discharge with identities like `add_zero` → finish with `rfl`.

10 Week 9: Addition World — Two Proofs & Notes

10.1 Notes & Discussion (Q&A)

Boundary between definitional and propositional equality. *Definitional* (or judgmental) equality is what the kernel reduces by computation/expansion and treats as the *same term* (e.g. Nat.add $a\ 0 \equiv a$, β-, δ-, ι-reductions). No proof object is needed; goals discharge by `rfl` or `simp` that performs computation. *Propositional* equality (`Eq`) is a type inhabited by a proof term (e.g. a chain of rewrites) showing two *a priori* different terms are equal; it is manipulated with lemmas like `add_assoc`, `congrArg`, `calc`, etc. Intuition: definitional = computation; propositional = reasoning about equality beyond raw computation.

How $\mathbb{N}$'s inductive definition enables recursive +. With `inductive Nat | zero | succ : Nat → Nat`, primitive recursion defines $a + n$ by

$$a + 0 \;=\; a, \qquad a + \text{succ}(n) \;=\; \text{succ}(a + n).$$

Because `Nat.rec` (or `Nat.recOn`) eliminates on the second argument, the recursion equations above are *defining* equalities that Lean can compute with directly.

SOLID in Lean / math-centric code.

- **S**ingle Responsibility: lemmas should prove exactly one fact; larger theorems compose small lemmas.
- **O**pen/Closed: extend the library with new lemmas (open), don't rewrite core definitions (closed).
- **L**iskov Substitution: use typeclasses/instances (`Semiring`, `Monoid`) so theorems work for any instance.
- **I**nterface Segregation: expose minimal lemma interfaces; avoid giant `simp` sets that do too much.
- **D**ependency Inversion: reason at the level of algebraic interfaces rather than concrete `Nat` when possible.

Math-first programming prizes *specification and proof*; SOLID maps well to clean lemma factoring and reusable abstractions.

Do all computational equality proofs end with `rfl`? No. Goals that are definitionally equal can close by `rfl`. Many "computational" goals can also finish via `simp` (using definitional rules like `Nat.add_zero`, `Nat.add_succ`). When non-computational rearrangements are needed (associativity/commutativity), we end with a lemma-driven chain or `simp`+`ring`-style tactics, not necessarily `rfl`.

Double induction for `add_right_comm` without `add_comm`/`add_assoc`? Yes in principle: prove $P(b, c) : a + b + c = a + c + b$ for all a by outer induction on c and inner induction on b, using only the defining equations of $+$ and `succ` injectivity. It is longer and more technical in Lean because you must re-establish associativity-like rearrangements manually. Using standard lemmas is far shorter and idiomatic.

How logic/maths proofs deepen Lean and industry skills. They enforce precise specs, decomposition into lemmas (modularity), and automated checking (CI-like guarantees). This is directly transferable to testing, refactoring, and API design.

Induction vs. algebraic lemmas (comm/assoc). Being able to reach the same theorem both ways shows *strategy choice*: induction exposes structure and well-founded decrease; lemma-based algebra leverages previously proved facts. In algorithm design this mirrors *design space exploration*: choose recursion/iteration (structural approach) or algebraic transformations/identities (law-driven approach) depending on clarity and cost.

Reference you cited (Tutorial World L8): https://adam.math.hhu.de/#/g/leanprover-community/nng4/world/Tutorial/level/8

10.2 Homework 9: Addition World — Level 5 (two solutions)

Statement (as in Addition World, right-commuting the tail). We show, for all $a, b, c \in \mathbb{N}$,

$$a + b + c \;=\; a + c + b \qquad \text{(often called } \texttt{add\_right\_comm}\text{)}.$$

Solution 1 (non-inductive, via lemmas).

```
-- Lean 4
theorem add_right_comm' (a b c : Nat) :
  a + b + c = a + c + b := by
  -- reshape: (a + b) + c    a + (b + c)
  calc
    a + b + c = a + (b + c) := by simp [Nat.add_assoc]
    _         = a + (c + b) := by simpa [Nat.add_comm]
    _         = a + c + b  := by simp [Nat.add_assoc]
```

Pen-and-paper proof. Using associativity and commutativity of $+$,

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

Each step is a standard ring law; hence $a + b + c = a + c + b$.

Solution 2 (inductive).

We prove $P(c):\ \forall ab.\ a+b+c = a+c+b$ by induction on c.

```
-- Lean 4
theorem add_right_comm_ind (a b c : Nat) :
  a + b + c = a + c + b := by
  induction c with
  | zero =>
      -- a + b + 0 = a + 0 + b
      simp [Nat.add_assoc, Nat.add_zero, Nat.zero_add]
  | succ c ih =>
      -- goal: a + b + succ c = a + succ c + b
      -- use the recursion equation for addition on the right argument
      -- x + succ c = succ (x + c)
      simp [Nat.add_assoc, ih]
```

Pen-and-paper proof (matching the code). Define $P(c):\forall a, b,\ (a+b)+c = (a+c)+b$. *Base* $c = 0$*:* $(a+b)+0 = a+b = a+0+b$ by $x+0 = x$ and $0+x = x$. *Step* $c \mapsto c+1$*:*

$$\begin{aligned}(a+b)+(c+1) &= \mathrm{succ}\big((a+b)+c\big) && \text{(def. of } +)\\ &= \mathrm{succ}\big((a+c)+b\big) && \text{(IH)}\\ &= (a+(c+1))+b && \text{(def. of } +)\\ &= a+(c+1)+b.\end{aligned}$$

Thus $P(c)$ holds for all c, so $a+b+c = a+c+b$.

Remark on lengths. The non-inductive proof is a three-line `calc`. The double-inductive route (or the single induction above) is longer because it reconstructs rearrangements using only the defining equations and the inductive hypothesis.

(Optional) What Level 5 asked and how this satisfies it.

The assignment asked for *two distinct solutions*: one relying on computation/lemmas and another by induction, each with a corresponding "pen-and-paper" argument. The two proofs above meet those requirements.

References

[BLA] Author, LaTeX Overview, Publisher, Year.