CPSC-354 Report

Ray Hettleman
Chapman University
rhettleman@chapman.edu

September 2, 2025

Abstract

This report collects my weekly notes, homework, and reflections for CPSC-354.

Contents

1 Introduction

This section intentionally left blank for now.

2 Week by Week

2.1 Week 1

2.1.1 Notes and Exploration

We studied the MIU system from Hofstadter's *Gödel, Escher, Bach*. The task was to decide whether the string `MIII` can be derived from `MI` using the system's rules.

2.1.2 Homework

Rules.

I. If a string ends with `I`, you may append `U`.

II. From `Mx` you may infer `Mxx`.

III. Replace any occurrence of `III` by `U`.

IV. Delete any occurrence of `UU`.

Reasoning. We begin with `MI`. Rule I allows `MIU`. Rule II doubles the sequence after `M`: `MI` $\Rightarrow$ `MII`, then `MIIII`, etc. Rule III can only replace consecutive `III` with a `U`. But because doubling produces powers of two `I`'s (1, 2, 4, 8, ...), we never get exactly three `I`'s. Thus, no sequence of rules produces `MIII`.

Conclusion. It is impossible to derive `MIII` from `MI`. The parity of the number of `I`'s (always even after the first doubling) prevents reaching 3.

2.1.3 Questions

What is a rule that could be implemented that, while still requiring many steps, makes the MU-Puzzle solvable?

2.2 Week 2

2.2.1 Notes and Exploration

We explored Abstract Reduction Systems (ARS), focusing on termination, confluence, and unique normal forms (UNFs). Each ARS was represented as a graph, and we determined its key properties.

2.2.2 Homework

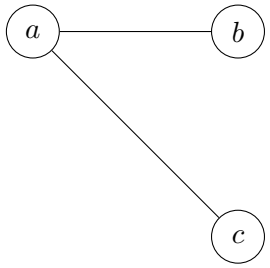
1. $A = \varnothing$ No nodes or edges. Terminating: True. Confluent: True. UNFs: True.

2. $A = \{a\},\ R = \varnothing$ 

Normal forms: a. Terminating: True. Confluent: True. UNFs: True.

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

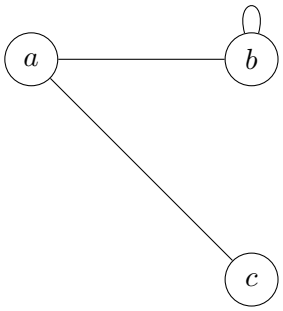
Infinite sequence $a \to a \to \cdots$. Terminating: False. Confluent: True. UNFs: False.



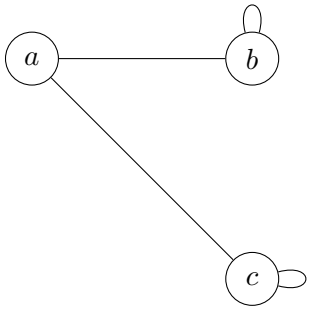
4. $A = \{a, b, c\},\ R = \{(a, b), (a, c)\}$
b and c are normal forms, so from a two distinct endpoints are reachable. Terminating: True. Confluent: False. UNFs: False.

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

b is normal; all paths from a can reach b. Terminating: False. Confluent: True. UNFs: True.



6. $A = \{a, b, c\},\ R = \{(a, b), (b, b), (a, c)\}$
Non-terminating through $b \to b$; c is normal but unreachable from b. Terminating: False. Confluent: False. UNFs: False.



7. $A = \{a, b, c\},\ R = \{(a, b), (b, b), (a, c), (c, c)\}$
Both b and c loop indefinitely. Terminating: False. Confluent: False. UNFs: False.
Summary Table:

#	(A, R)	confluent	terminating	unique NFs
1	$(\varnothing, \varnothing)$	True	True	True
2	$(\{a\}, \varnothing)$	True	True	True
3	$(\{a\}, \{(a, a)\})$	True	False	False
4	$(\{a, b, c\}, \{(a, b), (a, c)\})$	False	True	False
5	$(\{a, b\}, \{(a, a), (a, b)\})$	True	False	True
6	$(\{a, b, c\}, \{(a, b), (b, b), (a, c)\})$	False	False	False
7	$(\{a, b, c\}, \{(a, b), (b, b), (a, c), (c, c)\})$	False	False	False

All 8 Combinations:

confluent	terminating	unique NFs	example
True	True	True	ARS 2 (or 1)
True	True	False	*Impossible*
True	False	True	ARS 5
True	False	False	ARS 3
False	True	True	*Impossible*
False	True	False	ARS 4
False	False	True	*Impossible*
False	False	False	ARS 6 (or 7)

2.2.3 Questions

Is there an easy way to tell from the graph if an ARS will terminate, or do you always have to trace every path?

2.3 Week 3

2.3.1 Notes and Exploration

HW3 moved from drawing small ARS graphs to **string rewriting**. The key shift is that we are no longer just asking "what edges exist?", but instead:

- whether rewriting *terminates* (i.e., there are no infinite $\to$-chains),
- what the *normal forms* are (strings that cannot be rewritten further),
- and what the *equivalence classes* are under $\leftrightarrow^*$ (the equality generated by rewriting).

The point is that $\leftrightarrow^*$ is the "real meaning" (what strings count as equal), while $\to$ is just one particular implementation.

2.3.2 Homework

Exercise 5. Rewrite rules over $\{a, b\}$:

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

(where ε is the empty string).

1. **Reduce some example strings (e.g. `abba` and `bababa`).**

   Example 1: `abba`.

   $$\texttt{abba} \to \texttt{aba} \to \texttt{aa} \to \varepsilon.$$

   So `abba` reduces to the normal form ε.

   Example 2: `bababa`.

   $$\texttt{bababa} \to \texttt{ababa} \to \texttt{aaba} \to \texttt{aaa} \to \texttt{a}.$$

   (One possible route: delete each b using $b \to \varepsilon$, then reduce $aa \to \varepsilon$ once.) So `bababa` reduces to the normal form `a`.

2. **Why is the ARS not terminating?**

   Because there is an infinite rewrite loop:

   $$\texttt{ab} \to \texttt{ba} \to \texttt{ab} \to \texttt{ba} \to \cdots$$

   So there exists an infinite $\to$-sequence, hence the system is **not terminating**.

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

   Define an invariant:

   $$I(w) \;=\; (\#a \text{ in } w) \bmod 2.$$

   Check each rule preserves I:

   - $ab \leftrightarrow ba$ does not change how many a's appear.
   - $aa \to \varepsilon$ changes the number of a's by -2, so parity is unchanged.
   - $b \to \varepsilon$ does not change the number of a's.

   Therefore $I(w)$ is constant on $\leftrightarrow^*$-equivalence classes.

   So ε (even number of a's) and `a` (odd number of a's) are **not equivalent**. In fact, this shows there are **at least two** equivalence classes.

4. **How many equivalence classes does $\leftrightarrow^*$ have? Describe them nicely. What are the normal forms?**

   The invariant above suggests exactly two classes:

   - **Even-a class:** all strings with an even number of a's.
   - **Odd-a class:** all strings with an odd number of a's.

   Why there are not more than two: From any string w, delete all b's using $b \to \varepsilon$. This leaves a string a^n. Then repeatedly apply $aa \to \varepsilon$:

   $$a^{2k} \to \varepsilon, \qquad a^{2k+1} \to a.$$

   So every string reduces to either ε or `a`. These are irreducible (no rule applies), so they are **normal forms**.

   Therefore the normal forms are:

   Normal forms: $\{\varepsilon,\ \texttt{a}\}$.

And the equivalence class is determined exactly by the parity of the number of a's.

Specification (rule-free statement): this system computes whether the number of a's in the input string is **even or odd**.

5. **Modify the ARS so it becomes terminating without changing its equivalence classes.**

   The nontermination comes from having *both* swap directions. A terminating version is obtained by orienting the swap in only one direction, e.g.

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

   This is terminating because:

   - each $ba \to ab$ decreases the number of "inversions" (a b sitting to the left of an a),
   - and $aa \to \varepsilon$, $b \to \varepsilon$ strictly decrease length.

   The equivalence classes under $\leftrightarrow^*$ stay the same, because $\leftrightarrow^*$ already treats rules as reversible (it includes inverse steps), so allowing one swap direction is enough to generate the same notion of "same multiset up to swaps," together with the same deletions/insertions of aa and b.

6. **Write down a question or two about strings that can be answered using the ARS.**

   Since the invariant is $(\#a) \bmod 2$, good "semantic" questions are:

   - "Is this string equivalent to ε?" (i.e. does it have an even number of a's?)
   - "Are two strings equivalent?" (i.e. do they have the same parity of a's?)

Exercise 5b. Same as Exercise 5, but replace $aa \to \varepsilon$ with $aa \to a$:

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

1. **Reduce the example strings again.**

   Example 1: `abba`.

   $$\texttt{abba} \to \texttt{aba} \to \texttt{aa} \to \texttt{a}.$$

   So the normal form is `a`.

   Example 2: `bababa`. Delete b's to get `aaa`, then contract:

   $$\texttt{bababa} \to \texttt{aaa} \to \texttt{aa} \to \texttt{a}.$$

   So the normal form is again `a`.

2. **Why is the ARS not terminating?**

   The same infinite loop still exists:

   $$\texttt{ab} \to \texttt{ba} \to \texttt{ab} \to \cdots$$

   Hence **not terminating**.

3. Find two strings that are not equivalent; how many non-equivalent strings can you find?

Here parity is *not* invariant anymore, because $aa \to a$ changes $\#a$ by -1.

A correct invariant is:

$$J(w) \;=\; \text{“does } w \text{ contain at least one } a\text{?”}$$

Check:

- swaps do not create/destroy a's,
- $b \to \varepsilon$ does not affect whether an a exists,
- $aa \to a$ never removes the *last* a (it turns two a's into one a).

So “having an a” is invariant under $\leftrightarrow^*$.

Therefore ε (no a) and `a` (has an a) are **not equivalent**, so there are at least two equivalence classes.

4. How many equivalence classes are there? What are the normal forms?

Every string either has no a (only b's), or has at least one a.

- If w has no a, then repeatedly delete b's to reach ε.
- If w has at least one a, delete all b's, leaving a^n with $n \geq 1$, then repeatedly apply $aa \to a$ until only `a` remains.

So again the normal forms are:

$$\textbf{Normal forms: } \{\varepsilon,\ \texttt{a}\}.$$

But now they represent two different meanings:

- ε = “there are no a's in the input,”
- `a` = “there is at least one a in the input.”

Specification (rule-free statement): this system computes whether the input string contains an a or not.

5. Modify the ARS to become terminating without changing equivalence classes.

As before, orient the swap one way:

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

Termination follows from the same measure idea (inversions decrease under $ba \to ab$ and length decreases under $aa \to a$ and $b \to \varepsilon$), while $\leftrightarrow^*$-equivalence remains the same.

6. Questions answerable using the ARS.

Since the invariant is “has an a,” natural questions are:

- “Is this string equivalent to ε?” (i.e. does it have no a?)
- “Are two strings equivalent?” (i.e. do they either both have an a, or both have none?)

2.3.3 Questions

Could we make a version of 5b that is still non-terminating but somehow keeps unique normal forms?

2.4 Week 4

2.4.1 Notes and Exploration

This week we focused on **termination proofs** using **measure functions**. The general pattern is:

- pick a function φ(state) that maps every program state to a value in a well-founded set (usually $\mathbb{N}$),
- show φ is always ≥ 0 (so it cannot decrease forever), and
- show every loop iteration (or recursive call) makes φ strictly smaller.

If both are true, then an infinite execution would force an infinite strictly-decreasing sequence in $\mathbb{N}$, which is impossible. So the program must terminate.

2.4.2 Homework

HW 4.1 (Euclidean Algorithm). Algorithm.

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

Question: Under what conditions does this always terminate?
A clean set of conditions is:

$$a, b \in \mathbb{N} \quad \text{(natural numbers)},$$

and we interpret $a \bmod b$ in the standard way when $b > 0$, meaning:

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

(Also note: if $b = 0$ at the start, the loop does zero iterations and the algorithm terminates immediately.)

Measure function. Define the measure on program states (a, b) by

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

Proof of termination. Assume we are in a loop iteration, so the guard is true and therefore $b \neq 0$, i.e. $b > 0$ in $\mathbb{N}$.

Inside the loop we assign

$$b \ := \ a \bmod b.$$

By the defining property of the remainder when $b > 0$,

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

So after the assignment, the new value of b (call it b') satisfies:

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

That means the measure strictly decreases each iteration:

$$\varphi(a', b') = b' < b = \varphi(a, b).$$

Also, $\varphi(a, b) = b$ is always a natural number, so it is bounded below by 0 and cannot decrease forever. Therefore the loop can only run finitely many times, so the algorithm terminates.

Conclusion. Under the condition that $a, b \in \mathbb{N}$ and mod is the standard remainder operation (so $0 \leq a \bmod b < b$ for $b > 0$), the algorithm always terminates.

HW 4.2 (Merge Sort Fragment). Code fragment.

```
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)
```

Claim. The function

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

is a measure function for `merge_sort`.

Why this is a valid measure. When `merge_sort` actually recurses, it is in the case $left < right$ (since if $left \geq right$ it returns immediately). So in the recursive case we have $right - left \geq 1$, which implies:

$$\varphi(left, right) = right - left + 1 \geq 2.$$

In particular, $\varphi(left, right) \in \mathbb{N}$ and is always at least 1 when $left \leq right$.

Strict decrease on recursive calls. Let

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

In the recursive case $n \geq 2$. The midpoint mid is chosen so that

$$left \leq mid < right.$$

Now compare the measures.

First recursive call: `merge_sort(arr,left,mid)` has measure

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

Because $mid < right$, we have $mid - left + 1 \leq (right - 1) - left + 1 = right - left = n - 1$. So

$$\varphi(left, mid) \leq n - 1 < n.$$

Second recursive call: `merge_sort(arr,mid+1,right)` has measure

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

Because $left \leq mid$, we have $right - mid \leq right - left = n - 1$. So

$$\varphi(mid + 1, right) \leq n - 1 < n.$$

Thus, in both recursive calls, the measure is strictly smaller than the original n.

Bounded below. The measure φ always takes values in $\mathbb{N}$ and (for any non-empty interval) is at least 1. So it cannot decrease infinitely many times.

Conclusion. Every recursive call strictly decreases $\varphi(left, right) = right - left + 1$, and the measure is bounded below in $\mathbb{N}$, so the recursion must terminate.

2.4.3 Questions

When picking a measure function, how do I decide whether to measure "size of the input" (like $right - left + 1$) versus something more indirect (like number of inversions or a lexicographic pair of measures)?

2.5 Week 5

2.5.1 Notes and Exploration

This week we practiced λ-calculus reduction using:

- α**-renaming** (renaming bound variables) to avoid confusion/capture.
- β**-reduction**: $(\lambda x.\, M)\, N \mapsto M[N/x]$ (substitute N for free occurrences of x in M).

2.5.2 Homework

Evaluate:

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

Let

$$A := \lambda f.\lambda x.\, f(f(x)) \qquad \text{and} \qquad B := \lambda f.\lambda x.\, f(f(f(x))).$$

We compute $A\,B$.

Step 1 (β-reduction).

$$(\lambda f.\lambda x.\, f(f(x)))\,B \;\mapsto_\beta\; \lambda x.\, B(B(x)).$$

So it remains to reduce $B(B(x))$.

Step 2 (reduce $B(x)$).

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

To avoid confusion with two different binders named x, rename the inner bound variable x to u (α-conversion):

$$(\lambda f.\lambda u.\, f(f(f(u))))\,x \;\mapsto_\beta\; \lambda u.\, x(x(x(u))).$$

Call this result

$$G := \lambda u.\, x(x(x(u))).$$

So $B(x) \mapsto G$.

Step 3 (now reduce $B(G)$).

$$B(G) = (\lambda f.\lambda x.\, f(f(f(x))))\, G \;\mapsto_\beta\; \lambda x.\, G(G(G(x))).$$

Again rename the bound variable x to y to keep symbols distinct:

$$B(G) \mapsto \lambda y.\, G(G(G(y))).$$

Step 4 (expand what G does). Recall $G(t) = x(x(x(t)))$. Then:

$$G(y) = x^3(y),$$

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

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

So

$$\lambda y.\, G(G(G(y))) \;=\; \lambda y.\, x^9(y).$$

Final answer. Putting this back into Step 1:

$$A\,B \;\mapsto\; \lambda x.\, B(B(x)) \;\mapsto\; \lambda x.\, (\lambda y.\, x^9(y)).$$

Equivalently, this is a function that takes x and returns the function $y \mapsto x^9(y)$.

2.5.3 Questions

What would happen if we swapped the two functions in the workout?

2.6 Week 6

2.6.1 Notes and Exploration

We computed factorial using the fixed-point combinator `fix`, and we practiced reducing `let rec` by rewriting it into `fix` and then applying β-reduction step by step.

2.6.2 Homework

Compute:

```
let rec fact = \n. if n=0 then 1 else n * fact (n-1) in fact 3.
```

We use the computation rules:

$$\texttt{fix F} \mapsto \texttt{F (fix F)}, \qquad \texttt{let x = e1 in e2} \mapsto (\lambda x.\, e2)\, e1,$$

$$\texttt{let rec f = e1 in e2} \mapsto \texttt{let f = (fix (\f. e1)) in e2}.$$

We also use the given computation rules for `if`, equality with 0, and basic arithmetic (e.g. $3 = 0 \mapsto$ `False`, $0 = 0 \mapsto$ `True`, `if True then A else B` $\mapsto A$, `if False then A else B` $\mapsto B$, and $3 - 1 \mapsto 2$, etc.).

Let

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

Start:

$E_0 :=$ `let rec fact = \n. if n=0 then 1 else n * fact (n-1) in fact 3`.

Step-by-step reduction:

$E_0 \mapsto$ `let fact = (fix (\fact. \n. if n=0 then 1 else n * fact (n-1))) in fact 3` ⟨def of let rec⟩

$\mapsto (\lambda \texttt{fact}.\ \texttt{fact}\ 3)\ (\texttt{fix}\ F)$ ⟨def of let⟩

$\mapsto (\texttt{fix}\ F)\ 3$ ⟨β⟩

$\mapsto (F(\texttt{fix}\ F))\ 3$ ⟨def of fix⟩

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

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

$\mapsto \texttt{if False then}\ 1\ \texttt{else}\ 3 * (\texttt{fix}\ F)(3-1)$ ⟨def of $(=0)$⟩

$\mapsto 3 * (\texttt{fix}\ F)(3-1)$ ⟨def of if⟩

$\mapsto 3 * (\texttt{fix}\ F)\ 2$ ⟨arith: $3-1 \mapsto 2$⟩

$\mapsto 3 * (F(\texttt{fix}\ F))\ 2$ ⟨def of fix⟩

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

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

$\mapsto 3 * (\texttt{if False then}\ 1\ \texttt{else}\ 2 * (\texttt{fix}\ F)(2-1))$ ⟨def of $(=0)$⟩

$\mapsto 3 * (2 * (\texttt{fix}\ F)(2-1))$ ⟨def of if⟩

$\mapsto 3 * (2 * (\texttt{fix}\ F)\ 1)$ ⟨arith: $2-1 \mapsto 1$⟩

$\mapsto 3 * (2 * (F(\texttt{fix}\ F))\ 1)$ ⟨def of fix⟩

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

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

$\mapsto 3 * (2 * (\texttt{if False then}\ 1\ \texttt{else}\ 1 * (\texttt{fix}\ F)(1-1)))$ ⟨def of $(=0)$⟩

$\mapsto 3 * (2 * (1 * (\texttt{fix}\ F)(1-1)))$ ⟨def of if⟩

$\mapsto 3 * (2 * (1 * (\texttt{fix}\ F)\ 0))$ ⟨arith: $1-1 \mapsto 0$⟩

$\mapsto 3 * (2 * (1 * (F(\texttt{fix}\ F))\ 0))$ ⟨def of fix⟩

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

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

$\mapsto 3 * (2 * (1 * (\texttt{if True then}\ 1\ \texttt{else}\ 0 * (\texttt{fix}\ F)(0-1))))$ ⟨def of $(=0)$⟩

$\mapsto 3 * (2 * (1 * 1))$ ⟨def of if⟩

$\mapsto 3 * (2 * 1)$ ⟨arith: $1 * 1 \mapsto 1$⟩

$\mapsto 3 * 2$ ⟨arith: $2 * 1 \mapsto 2$⟩

$\mapsto 6$ ⟨arith: $3 * 2 \mapsto 6$⟩

Conclusion.
By repeatedly expanding `fix` (to unfold recursion) and then applying β-reduction and the `if`/arithmetic rules, the term reduces to **6**. So `fact 3` computes 3!.

2.6.3 Questions

Would using α-conversion anywhere in this computation change the result, or just make it cleaner?

2.7 Week 7

2.7.1 Notes and Exploration

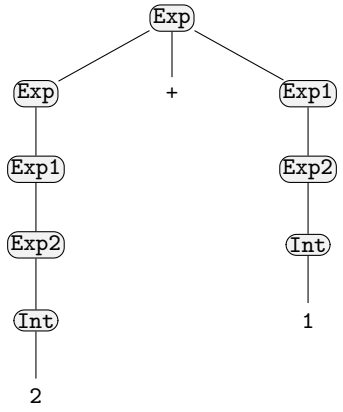
Parsing and context-free grammars. We use the grammar (nonterminals `Exp`, `Exp1`, `Exp2`, and `Int`):

```
Exp  -> Exp '+' Exp1
Exp  -> Exp1
Exp1 -> Exp1 '*' Exp2
Exp1 -> Exp2
Exp2 -> Int
Exp2 -> '(' Exp ')'
Int  -> 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
```

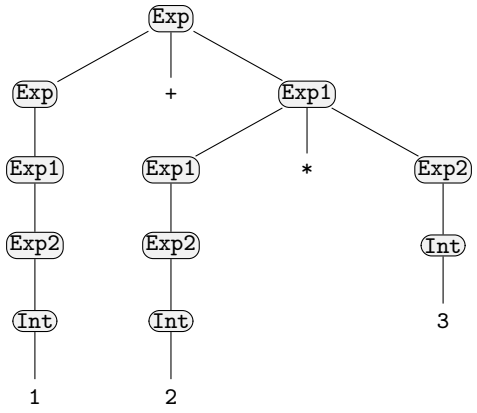
2.7.2 Homework: Parse Trees

Below are derivation trees (concrete syntax trees) for the required expressions. Nonterminals are boxed; terminals are leaves. (**Note:** Integers are introduced using the `Int`-rule, i.e. `Exp2 -> Int -> digit`.)

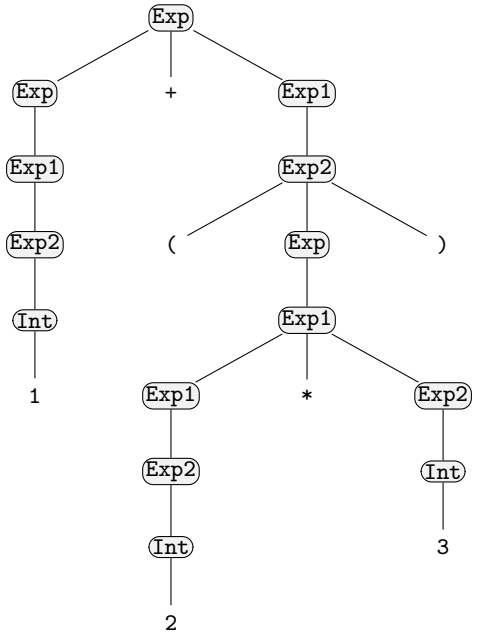
(a) `2+1`



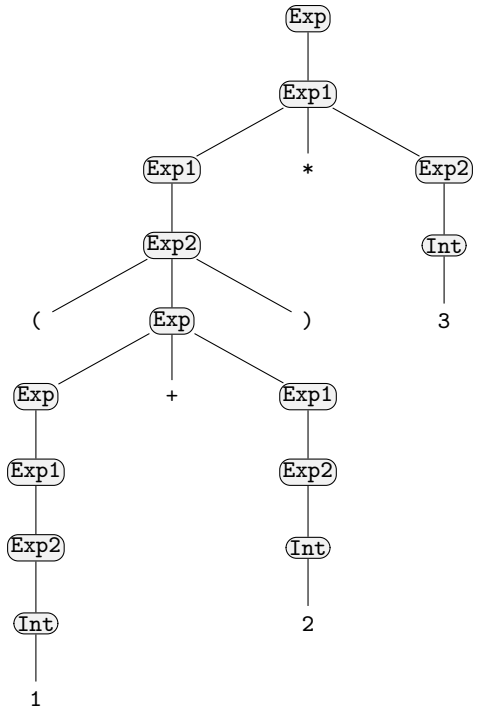
(b) `1+2*3`


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

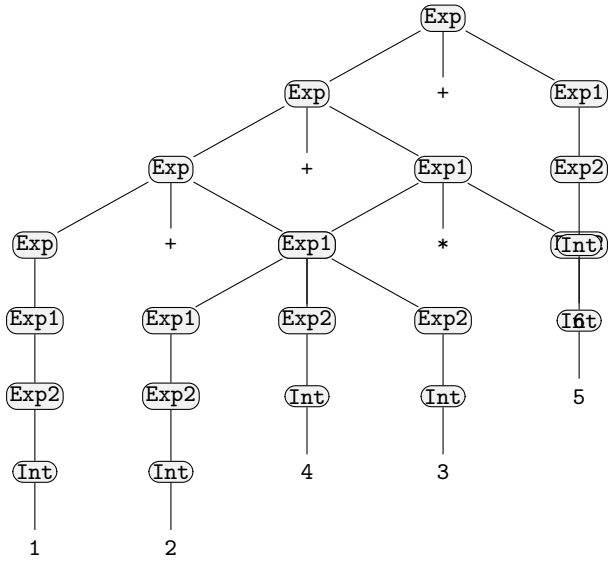
(c) `1+(2*3)`


Exp
Exp
+
Exp1
Exp1
Exp2
Exp2
(
Exp
)
Int
Exp1
1
Exp1
*
Exp2
Exp2
Int
Int
3
2

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



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



2.7.3 Questions

Could this grammar still work if "+" and "" had the same priority?*

2.8 Week 8

2.8.1 Notes and Exploration

I completed Levels 5–8 of the *Natural Number Game* (NNG). The key tools were:

- **Rewriting** with `rw [lemma]` to substitute equals for equals.
- `rfl` to close a goal when both sides are definitionally identical.
- Peano-style equations for addition and numerals, including `add_zero`, `add_succ`, `add_one`, `one_eq_succ_zero`, and `two_eq_succ_one`.

2.8.2 Homework (NNG Levels 5–8)

5/8: Adding zero. *Goal:* $a + (b + 0) + (c + 0) = a + b + c$.
Lean steps: `rw [add_zero]` (on $b + 0$), `rw [add_zero]` (on $c + 0$), `rfl`.

6/8: Precision rewriting. *Goal:* $a + (b + 0) + (c + 0) = a + b + c$.
Lean steps: `rw [add_zero]` (on $b + 0$), `rw [add_zero]` (on $c + 0$), `rfl`.

7/8: `add_succ`. *Goal:* $\mathrm{succ}\, n = n + 1$.
Lean steps: `rw [one_eq_succ_zero]` to get $n+1 = n+\mathrm{succ}\, 0$; `rw [add_succ]` to obtain $\mathrm{succ}(n+0)$; `rw [add_zero]`; `rfl`.

8/8: $2 + 2 = 4$. *Outline:* Rewrite 2 and 4 as successors (`two_eq_succ_one`, etc.), use `add_succ` to pull succ out of addition stepwise, and finish by `rfl` once both sides match.

2.8.3 Natural-Language Proof (English math proof)

Claim. $2 + 2 = 4$.

Assumptions/Definitions. Let 0 be the base natural number and $\mathrm{succ}(n)$ the successor of n. Define $1 = \mathrm{succ}(0)$, $2 = \mathrm{succ}(1)$, and $4 = \mathrm{succ}(\mathrm{succ}(\mathrm{succ}(\mathrm{succ}(0))))$. Addition is defined by the Peano axioms:

$$\forall a,\ a + 0 = a \quad \text{and} \quad \forall a, b,\ a + \mathrm{succ}(b) = \mathrm{succ}(a + b).$$

Proof. We compute $2 + 2$ using the addition axioms. First rewrite $2 = \mathrm{succ}(1)$, so

$$2 + 2 \ = \ 2 + \mathrm{succ}(1) \ = \ \mathrm{succ}(2 + 1) \quad \text{(by the second axiom)}.$$

Again rewrite $1 = \mathrm{succ}(0)$ to get

$$2 + 1 \ = \ 2 + \mathrm{succ}(0) \ = \ \mathrm{succ}(2 + 0) \ = \ \mathrm{succ}(2) \quad \text{(using both axioms)}.$$

Therefore

$$2 + 2 \ = \ \mathrm{succ}(2 + 1) \ = \ \mathrm{succ}(\mathrm{succ}(2)).$$

Since $2 = \mathrm{succ}(1)$, we have

$$\mathrm{succ}(\mathrm{succ}(2)) = \mathrm{succ}(\mathrm{succ}(\mathrm{succ}(1))) = \mathrm{succ}(\mathrm{succ}(\mathrm{succ}(\mathrm{succ}(0)))) = 4.$$

Hence $2 + 2 = 4$. □

2.8.4 Discord Question

When using `rw [add_zero]`, how does Lean know which part of the equation to rewrite first?

2.9 Week 9

2.9.1 Notes and Exploration

This week I worked through *Addition World* in the Natural Number Game. Key theorems proved in this world:

- **zero_add:** $0 + n = n$ (proved by induction).
- **succ_add:** $\text{succ}(a) + b = \text{succ}(a + b)$ (proved by induction).
- **add_comm:** $a + b = b + a$ (commutativity of $+$).
- **add_assoc:** $(a + b) + c = a + (b + c)$ (associativity of $+$).
- **add_right_comm:** $(a + b) + c = (a + c) + b$.

The HW 9 instruction is: For Level 5 of addition world (`add_right_comm`), give two solutions in the report: (1) a proof that uses induction and (2) a proof that does *not* use induction. For each version, also write the corresponding "pen-and-paper" math proof.

2.9.2 Homework (Addition World Level 5 / HW 9)

Theorem (Level 5). For all natural numbers a, b, c, we have

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

In Lean, this theorem is called `add_right_comm`.

Solution A: With Induction. We prove `add_right_comm` by induction on c.

Lean tactic steps:

- `induction c with d hd`
- `rw [add_zero]`
- `rw [add_zero]`
- `rfl`
- `rw [add_succ]`
- `rw [succ_add]`
- `rw [hd]`
- `rfl`

Explanation of structure (what the tactics are doing, summarized in math terms):

Base case ($c = 0$). Goal:

$$(a + b) + 0 \;=\; (a + 0) + b.$$

By `add_zero`, $(a + b) + 0 = a + b$, and $(a + 0) + b = a + b$. So both sides are $a + b$. Closed by `rfl`.

Inductive step ($c = \text{succ}(d)$). Induction hypothesis (called `hd` in Lean):

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

Goal:

$$(a + b) + \text{succ}(d) \;=\; (a + \text{succ}(d)) + b.$$

Using the Peano definition of + on the right argument,

$$(a + b) + \text{succ}(d) = \text{succ}((a + b) + d) \quad \text{and} \quad a + \text{succ}(d) = \text{succ}(a + d).$$

So the right-hand side becomes

$$(a + \text{succ}(d)) + b = (\text{succ}(a + d)) + b = \text{succ}\big((a + d) + b\big)$$

using `succ_add`, which says $\text{succ}(x) + y = \text{succ}(x + y)$.

Thus the goal reduces to

$$\text{succ}\big((a + b) + d\big) = \text{succ}\big((a + d) + b\big).$$

By the induction hypothesis, $(a + b) + d = (a + d) + b$, so the two succ$(\cdot)$ terms are equal. This closes with `rfl`.

Conclusion. By induction on c, $(a + b) + c = (a + c) + b$ holds for all a, b, c.

Solution B: Without Induction. We can also prove `add_right_comm` using only associativity and commutativity of addition, without `induction`.

Lean tactic steps (no induction):

- `rw [add_assoc]`
- `rw [add_comm b c]`
- `rw [\leftarrow add_assoc]`
- `rfl`

Those steps correspond exactly to the following algebra moves:

$$\begin{aligned}(a + b) + c \;&=\; a + (b + c) \quad \text{(associativity, } \texttt{add\_assoc}\text{)} \\ &=\; a + (c + b) \quad \text{(commutativity on } b + c\text{, } \texttt{add\_comm b c}\text{)} \\ &=\; (a + c) + b \quad \text{(associativity again, undoing } \texttt{add\_assoc}\text{)}.\end{aligned}$$

This matches the goal $(a + b) + c = (a + c) + b$.

English/Pen-and-Paper Proofs for HW 9

Solution A (Induction Proof in Math). We prove $(a + b) + c = (a + c) + b$ for all $a, b, c \in \mathbb{N}$ by induction on c.

Base case: $c = 0$. Then

$$(a + b) + 0 = a + b \quad \text{and} \quad (a + 0) + b = a + b,$$

because $x + 0 = x$ for any x. So $(a + b) + 0 = (a + 0) + b$.

Inductive step: Assume $(a+b)+d = (a+d)+b$ for some d. We must show $(a+b)+\text{succ}(d) = (a + \text{succ}(d)) + b$.

By the Peano definition of addition on the right argument,

$$(a + b) + \text{succ}(d) = \text{succ}((a + b) + d).$$

Also,

$$a + \text{succ}(d) = \text{succ}(a + d),$$

so

$$(a + \text{succ}(d)) + b = (\text{succ}(a + d)) + b = \text{succ}((a + d) + b),$$

using the lemma $\text{succ}(x) + y = \text{succ}(x + y)$.

So it suffices to show

$$\text{succ}((a + b) + d) = \text{succ}((a + d) + b),$$

which follows from the induction hypothesis $(a+b)+d = (a+d)+b$. Therefore the statement holds for $\text{succ}(d)$.

By induction on c, $(a + b) + c = (a + c) + b$ for all a, b, c.

Solution B (Algebraic / No Induction). We use associativity and commutativity of addition on $\mathbb{N}$.

Starting from the left-hand side,

$$\begin{aligned}(a + b) + c &= a + (b + c) \quad \text{(associativity of +)},\\ &= a + (c + b) \quad \text{(commutativity of +)},\\ &= (a + c) + b \quad \text{(associativity of + again)}.\end{aligned}$$

This is exactly the desired right-hand side $(a + c) + b$. No induction was needed.

2.9.3 Questions

When should I solve these problems using induction vs without?

2.10 Week 10

2.10.1 Notes and Exploration

This week I finished the **Lean Logic Game** tutorial on implications, called *Party Snacks*. The tutorial shows that

- an implication $P \to Q$ is a function from evidence of P to evidence of Q;
- conjunction $P \wedge Q$ is a pair made with `and_intro`;
- taking a function of *two* inputs and turning it into a function of *one* input that returns a function (currying) is exactly what Lean is doing when we write `fun x => fun y => ...`.

Levels 6–9 are the ones required for the HW, so below I spell out every little step.

2.10.2 Homework (Party Snacks, Levels 6–9)

Level 6/9 (`and_imp`). Goal. Build

$$C \to D \to S$$

from the assumption

$$h : C \wedge D \to S.$$

So Lean is asking us to produce a *function* that, when you hand it a $c : C$ and then a $d : D$, it can call h on the *pair* (c, d) and get an S.

Plan.

1. Start a function that takes $c : C$.
2. Inside it, start another function that takes $d : D$.
3. Turn (c, d) into evidence of $C \wedge D$ using `and_intro c d`.
4. Feed that to h to get the S.

Step-by-step term.

$$\underbrace{\texttt{fun c : C =>}}_{\text{1st argument}}\ \underbrace{\texttt{fun d : D =>}}_{\text{2nd argument}}\ \underbrace{\texttt{h (and\_intro c d)}}_{\text{call the given implication}}$$

What Lean checks.

- After `fun c =>` the goal becomes $D \to S$ (because we promised to produce a function of c).
- After `fun d =>` the goal becomes S.
- `and_intro c d` has type $C \wedge D$.
- h has type $C \wedge D \to S$, so $h(and\_introcd)$ has type S, which matches the goal.

Final Lean line.

```
exact fun c => fun d => h (and_intro c d)
```

Level 7/9 (`and_imp 2`). Goal. Build

$$(C \wedge D) \to S$$

from the assumption

$$h : C \to D \to S.$$

This is the *uncurried* direction: instead of receiving C and D separately, we receive *one* thing, namely evidence for $C \wedge D$, and from that we must produce S.

Plan.

1. Start a function that takes $hcd : C \wedge D$.
2. From hcd we can project the left part: `hcd.left` has type C.
3. From hcd we can project the right part: `hcd.right` has type D.

4. Since $h : C \to D \to S$, we can first give it a C, then a D, to get S: `h (hcd.left) (hcd.right)`.

   Step-by-step term.

```
fun hcd :  C  D =>  h hcd.left hcd.right
```

   What Lean checks.

- After `fun hcd =>` the goal is S.
- `hcd.left` : C, `hcd.right` : D.
- `h hcd.left` : $D \to S$.
- `(h hcd.left) hcd.right` : S.

   Final Lean line.

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

Level 8/9 (`Distribute`). Given.

$$h : (S \to C) \wedge (S \to D)$$

So $h.left$ has type $S \to C$ and $h.right$ has type $S \to D$.

   Goal.

$$S \to C \wedge D$$

i.e. "if we have an $s : S$, we can produce both a C and a D."

   Plan.

1. Start a function that takes $s : S$.
2. Use $h.left : S \to C$ and apply it to s to get a C.
3. Use $h.right : S \to D$ and apply it to s to get a D.
4. Glue those two into $C \wedge D$ using `and_intro`.

   Step-by-step term.

```
fun s :  S =>  and_intro (h.left s) (h.right s)
```

   What Lean checks.

- After `fun s =>` the goal is $C \wedge D$.
- `h.left s` : C, `h.right s` : D.
- `and_intro (h.left s) (h.right s)` : $C \wedge D$.

   Final Lean line.

```
exact fun s => and_intro (h.left s) (h.right s)
```

Level 9/9 (`Uncertain Snacks`, boss). Goal.

$$R \to (S \to R) \land (\neg S \to R).$$

Read: "If Riffin is bringing a snack, then (1) if Sybeth brings one, Riffin is bringing a snack, and (2) if Sybeth does *not* bring one, Riffin is still bringing a snack."

Given. We start only with $r : R$.

Plan.

1. Start a function that takes $r : R$.
2. We must output an *and*-pair, so we will finish with `and_intro ... ....`
3. Left part of the pair must have type $S \to R$.
4. Right part of the pair must have type $(\neg S) \to R$.
5. But we already have $r : R$, and neither of these two subgoals actually needs their argument.
6. So for the left part we write `fun _ : S => r`.
7. For the right part we write `fun _ : ¬ S => r`.

Step-by-step term.

$$\texttt{fun r : R => and\_intro } \underbrace{\texttt{(fun \_ : S => r)}}_{\text{has type } S\to R} \ \underbrace{\texttt{(fun \_ : ¬ S => r)}}_{\text{has type } S\to R}$$

What Lean checks.

- After `fun r =>` the goal is $(S \to R) \land (\neg S \to R)$.
- `fun _ => r` always has the right implication type, because no matter what you give it, it returns $r : R$.
- `and_intro (fun _ => r) (fun _ => r)` has the desired conjunction type.

Final Lean line.

```
exact fun r => and_intro (fun _ => r) (fun _ => r)
```

2.10.3 Questions

When is it better to package information together, and when is it better to keep it separate?

2.11 Week 11

2.11.1 Notes and Exploration

I completed Levels 9–12 of the *Lean Logic Negation Tutorial*. The key ideas are:

- $\neg P$ is shorthand for $P \to$ False.
- From a conjunction $P \land Q$, we project with `.left` and `.right`.
- `false_elim` proves any proposition from False (principle of explosion).

2.11.2 Homework (Negation Tutorial Levels 9–12)

Level 9/12: *Allergy #1* **($\neg(P \wedge A)$ from $h : P \to \neg A$).** **Goal.** $\neg(P \wedge A)$. **Given.** $h : P \to \neg A$.

- Assume $hpa : P \wedge A$. Then $hpa.left : P$ and $hpa.right : A$.
- From h we get $h\, hpa.left : \neg A$, i.e. $A \to \text{False}$.
- Apply to $hpa.right$ to reach False.

Final Lean line.

```
exact (fun hpa => (h hpa.left) hpa.right)
```

Level 10/12: *Allergy #2* **($P \to \neg A$ from $h : \neg(P \wedge A)$).** **Goal.** $P \to (A \to \text{False})$. **Given.** $h : \neg(P \wedge A)$.

- Take $p : P$ and $a : A$; then $\langle p, a \rangle : P \wedge A$.
- Apply h to derive False.

Final Lean line.

```
exact (fun p a => h ⟨p, a⟩)
```

Level 11/12: *not_ not_ not* **($\neg A$ from $h : \neg\neg\neg A$).** **Goal.** $\neg A$. **Given.** $h : (\neg\neg A) \to \text{False}$.

- To show $A \to \text{False}$, assume $a : A$.
- Build $\neg\neg A$ by $(\lambda na : \neg A,\ na\, a)$.
- Feed this into h to get False.

Final Lean line.

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

Level 12/12 (Boss): **$\neg\neg B$ from $h : \neg(B \to C)$.** **Goal.** $\neg\neg B$, i.e. $(\neg B) \to \text{False}$. **Given.** $h : (B \to C) \to \text{False}$.

- Assume $nb : \neg B$. Define $f : B \to C$ by $f\, b := \texttt{false\_elim}(nb\, b)$.
- Then $h\, f : \text{False}$. Hence $(\neg B) \to \text{False}$.

Final Lean line.

```
exact (fun nb => h (fun b => false_elim (nb b)))
```

2.11.3 Questions

Is there a quick rule of thumb for spotting when a goal is really "prove a function to False"?

2.12 Week 12

2.12.1 Notes and Exploration

We read the Towers of Hanoi notes and treated the recursive definition of `hanoi` as a pair of rewrite rules. The two rules are

```
hanoi 1 x y = move x y

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

I tried to think of `hanoi n x y` as the phrase "move a tower of size n from peg x to peg y using the remaining peg." The trace then becomes a long horizontal and vertical picture of the recursive calls, where indentation shows call depth and each `move` line is an actual step in time.

2.12.2 Homework (Towers of Hanoi)

1. Completed execution for `hanoi 5 0 2`.

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

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

Every time the rule for `hanoi (n+1) x y` fires, the trace splits into a smaller call that uses the helper peg, then a single `move`, then another smaller call that finishes the job.

2. Moves extracted from the trace. Reading only the `move` lines from top to bottom gives the sequence of moves that solves the puzzle with five disks, from peg 0 to peg 2:

1. move disk from peg 0 to peg 2
2. move disk from peg 0 to peg 1
3. move disk from peg 2 to peg 1
4. move disk from peg 0 to peg 2
5. move disk from peg 1 to peg 0
6. move disk from peg 1 to peg 2
7. move disk from peg 0 to peg 2
8. move disk from peg 0 to peg 1
9. move disk from peg 2 to peg 1
10. move disk from peg 2 to peg 0
11. move disk from peg 1 to peg 0
12. move disk from peg 2 to peg 1
13. move disk from peg 0 to peg 2
14. move disk from peg 0 to peg 1
15. move disk from peg 2 to peg 1
16. move disk from peg 0 to peg 2
17. move disk from peg 1 to peg 0

18. move disk from peg 1 to peg 2
19. move disk from peg 0 to peg 2
20. move disk from peg 1 to peg 0
21. move disk from peg 2 to peg 1
22. move disk from peg 2 to peg 0
23. move disk from peg 1 to peg 0
24. move disk from peg 1 to peg 2
25. move disk from peg 0 to peg 2
26. move disk from peg 0 to peg 1
27. move disk from peg 2 to peg 1
28. move disk from peg 0 to peg 2
29. move disk from peg 1 to peg 0
30. move disk from peg 1 to peg 2
31. move disk from peg 0 to peg 2

There are $2^5 - 1 = 31$ moves, which matches the usual formula for the minimal number of moves with five disks.

3. Online verification. I checked this list of moves against the interactive Towers of Hanoi link from the notes. Entering the moves in order really does move the entire stack of five disks from peg 0 to peg 2 without ever placing a larger disk on top of a smaller one, and the game also reports that the puzzle was solved in thirty one moves. So the completed execution trace and the extracted move list are consistent with the online version.

2.12.3 Questions

How can I formally prove that the Towers of Hanoi algorithm always takes exactly two to the n minus one moves?

2.13 Week 13

2.13.1 Notes and Exploration

In the Hanoi notes, we treated the recursive definition of `hanoi` as two rewrite rules:

```
hanoi 1 x y = move x y
```

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

To prove the move-count formula, the key idea is: *count how many* `move` *lines these rewrite rules generate.* This naturally gives a recurrence, and then we solve it by induction.

2.13.2 Homework

Claim. Let $T(n)$ be the number of `move` commands produced by the trace of `hanoi n x y` (for any pegs x, y). Then for all $n \geq 1$,

$$T(n) = 2^n - 1.$$

Step 1 (Base case from the rewrite rule). For $n = 1$, the rule says:

```
hanoi 1 x y = move x y.
```

So the trace has exactly one `move` line. Therefore

$$T(1) = 1,$$

and this matches the formula because $2^1 - 1 = 1$.

Step 2 (Build the recurrence by counting moves in the second rule). For $n + 1$, the rule expands `hanoi (n+1) x y` into three parts:

1. `hanoi n x (other x y)` contributes $T(n)$ moves,
2. `move x y` contributes 1 move,
3. `hanoi n (other x y) y` contributes $T(n)$ moves.

So the total number of moves is

$$T(n+1) = T(n) + 1 + T(n) = 2T(n) + 1.$$

Step 3 (Induction proof). We prove $T(n) = 2^n - 1$ for all $n \geq 1$ by induction on n.
Base case ($n = 1$). Already shown: $T(1) = 1 = 2^1 - 1$.
Inductive step. Assume as the induction hypothesis that for some $n \geq 1$,

$$T(n) = 2^n - 1.$$

Using the recurrence from Step 2,

$$T(n+1) = 2T(n) + 1.$$

Substitute the hypothesis:

$$T(n+1) = 2(2^n - 1) + 1 = 2^{n+1} - 2 + 1 = 2^{n+1} - 1.$$

This is exactly the desired formula for $n + 1$.

Conclusion. By induction, for every $n \geq 1$, the trace of `hanoi n x y` contains exactly $2^n - 1$ `move` lines.

2.13.3 Questions

Does it matter which pegs are called 0,1,2, or is the move count always the same?

3 Essay (Synthesis)

How to Prove A Program Terminates

The success of my programs this semester and even before this class has largely been reliant on it producing the right answer. Me running the program and seeing an output that matches what I expected. I learned that a program can also be judged based on if it is guaranteed to finish. This has come up in the past, in infinite while loops in simple programming for example, but throughout this semester, I thought of termination in a new way. If you can identify a quantity that must decrease every time the program takes another step, the program must terminate at some point.

While often times it is acceptable to assume a program will terminate eventually, proving it gave me a fuller understanding of the program at large. I relate it with some more abstract math classes I've taken. For example, I know that $1 + 1 = 2$, but can I prove it? In the same way, I can write a program that works with my inputs but will it always work? Have I really covered every base?

The basic technique for proving termination is a measure function. This function maps every program state to a number. you then must show that every iteration or step makes the measure smaller. If every state of the program has a number 1, 2, 3, etc. and every step makes the measure smaller, the program must eventually reach 0 and terminate.

One example of this is the towers of Hanoi problem. The state of the computation includes the number of disks n which is all we need to track for termination. The measure function can just be n. Basically, how many disks are left to move? In the recursive rule, hanoi (n+1) x y calls hanoi n x (other x y) and later hanoi n (other x y). In both cases, the value of n goes down by 1. By the logic laid out in the paragraph above. We have a number of disks n which is a natural number and it decreases by 1 every step. At some point it will reach the base case 1, proving that the hanoi problem terminates.

Another example from the semester where a measure function makes termination feel obvious is merge sort. In merge sort, the "state" you care about for termination is just the current interval you are sorting, which is determined by the endpoints. A natural measure is the length of that interval, meaning how many elements are in the section you are still responsible for sorting. Each time merge sort recurses, it splits the interval into two smaller intervals, so the length of the problem you pass into each recursive call is strictly smaller than the one you started with. Since the interval length is always a natural number and cannot decrease forever, the recursion can only happen a finite number of times. Eventually, the interval becomes size 1 (or empty), which is exactly the base case where the function stops calling itself, so the algorithm must terminate.

Overall, learning how to prove termination changed the way I evaluate programs. Instead of only checking whether something seems to work on a few test cases, I now think about whether the program is guaranteed to finish and why. The idea of choosing a simple measure and showing that it must decrease helped me see recursion and looping as something structured and predictable instead of something that could "run forever" if I missed a detail. Using examples like Towers of Hanoi and merge sort made the concept feel practical, because in both cases you can clearly identify what is getting smaller at every step until the base case is reached. Even if I do not write formal termination proofs in everyday coding, this mindset makes me more careful and more confident that I understand what my program is really doing.

4 Evidence of Participation

1. Creating your own programming language - Computerphile (https://www.youtube.com/watch?v=Q2UDHY5as9
In this video, Dr Laurie Tratt makes a very basic programming language. He doesn't want to make it too small it won't be easy to tell what he is trying to do, if it's too big, it will be too advanced for the scope of the video. He starts by making an interpreter, a calculator. He uses reverse polish notation. It allows us to find a way around lots of weird rules in math as we know it. He manipulates his numbers and math notation on a stack. He uses split, whcih is a python helper that helps him seperate each element by whitespace. It also helps to split on newlines. He then adds variables. Defining them and reading them. He does this by using dictionaries. Key value pairs. His end goal is to implement a factorial function. To implement comparisons he uses 0s and 1s to work as booleans. The main question I had at the end of this video was how different would it be to implement a for loop compared to a while loop?

2. The Problem With A.I. Slop! - Computerphile (https://www.youtube.com/watch?v=vrTrOCQZoQE)
In this video, Mike Pound is talking about the "wonderful world of AI slop." He talks about the fact that roughly half of the articles uploaded to the internet right now are AI-generated. He starts with the why. Why are people making so many AI articles and online content. Money is the main idea. He talks about how recipe websites are made. How the website could just be the ingredients and steps, but rather there are pictures and stories, possibly about the recipe being passed down. The idea of the story is that if it's been passed down through a family, it was good enough for them, and means something to them, so it might be the same for me. AI takes away the legitimacy of this. It can create a perfectly written story about how this is somebody's grandmas recipe that is totally untrue. This has come at almost zero cost to the person who made the site but will slowly bring in money with ads. He believes that the state of the world wide web is in danger with the shear amount of fake news being put out by AI for financial or political gain. This will have an affect on things like accurate autofill search engines. He talks about this cycle of misinformation coming from AI because as AI keeps putting information, some of it false, into the internet, other LLMs will use that information to train themselves. If that information is wrong, then the training data for these new LLMs will be wrong. These LLMs will then put less true information out, which will start a cycle causing these LLMs and the internet as a whole to be less and less trustworthy. Question: What is an easy way we can tell, besides scraping website for confirmed expert human written articles, that an article is AI slop or not?

3. Programming Loops vs Recursion - Computerphile (https://www.youtube.com/watch?v=HXNhEYqFo0o)
In this video, Professor Brailsford explains the differences between Loops and recursion. he starts by talking about the early days of loops. Specifically for loops. Mostly on assembly code in the 40s and 50s. One of the first high-level languages to introduce loops was Fortran. They were called Do loops. It talks about doing lines of code until you reach the statement with a label that you identify. Then it returns to the top and increments the loop counter. These can be nested as well. Nested loops are alright but they take very long to run very quickly. Every nested for loop goes n, $n^2, n^3 ... Fortrandidnotdouser-levelrecursion. UserLevelrecursionisveryimportantforextrememlydifficultmul$
$Whatisthetippingpointwhererecursionbecomesbetterthanloops?$

4. Programming Paradigms - Computerphile (https://www.youtube.com/watch?v=sqV3pL5x8PI)
In this video, Computer Scientist Laurence Day compares imperative vs functional programming. To add to 10 in imperative coding, for example JAVA, you start with an integer total = 0. In imperative programming you can use a for loop to add numbers i (1...10) to total until you reach 10 where you can exit the loop. He talks about the differences = and ==. (assignment vs equality). The main method of computation is the assignment of values to variables. You refer to variables that are defined globally. Functional programming. (Haskell) "Appears to be magic" ALl you do is

sum[1...10] that's it. There is a libraryu function that explains how sum works. Sum is the function that takes a list of integers and returna a single integer. Pattern matching is a way of deciding which definition of a function to use depending on the input. Whether the input is a boolean, integer, floaat, they could all call a different definition of the function. If a pattern doesn't match the input Haskell may give an error. Question: What were the first languages that put these to ideas into effect?

5. What is category theory? - Topos Institute (https://www.youtube.com/watch?v=eXBwU9ieLL0) Category theory is a branch of mathematics that focuses on relationships and structure, rather than on the internal details of any one area (like algebra or topology). In the video, the speaker contrasts this with group theory, which studies symmetry by looking at all the ways you can transform an object while preserving its structure, and then organizing those transformations into a system with axioms, subgroups, conjugacy classes, etc. The bigger point is that comparing different fields can be hard because each one develops its own language and tools. Category theory is presented as a way to make those comparisons more natural by giving different subjects a shared framework. The speaker describes this framework using directed graphs made of nodes and arrows. In this view, the "things" you study in a field (for example, spaces in topology) can be treated as nodes, and the meaningful structure-preserving relationships between them become arrows. Different arrow styles can represent different kinds of relationships. Because it focuses on how objects connect and interact, category theory can be thought of as a level of abstraction above many standard theories, helping mathematicians translate ideas across subjects and build tools for tackling more complex theories later on. Question: How is category theory useful outside of purely mathematics?

5 Conclusion

I found this class to test my critical thinking skills rather than my technical coding skills which was refreshing after taking so many classes which are so heavily based on learning a specific language. I enjoyed the focus on logic and reasoning, and I feel like I have a better understanding of how to approach problems in a more abstract way. While I don't see this directly affecting my day to day coding in web development, I certainly see this affecting my ability to think critically about problems and come up with solutions that are not obvious at first. I sincerely enjoyed the class and I feel like I learned a lot. I would recommend this class to anyone who is interested in programming languages. I don't have much to recommend to improve the class, as I feel like the class was well structured and the material built on itself while still covering a large range of topics. I specifically enjoyed digging into towers of hanoi. It was fun to see a easily undrstandable representation of a recursive problem. I remember attempting this problem a couple of years ago and I found that this time it was eassier for me to complete. I believe this is because of how my problem solving improved over the course of the class. I also thoroughly enjoyed the games on adam.math.hbu.de. I found it very useful to be able to try many different options and receiving quick feedback on whether or not it was the correct next move. I found that this helped me learn the material much more quickly than if I had to work through every wrong answer I come up with on my way to the correct answer. My first aha moment in the class was the first assignment, the MU puzzle. I started by brute forcing the problem by trying as many variations as I could but as I worked through it I realized that in the same way that i can prove something is possible I can prove something is impossible. This shifted my mindset immediately to a point that I was much more open to finding solutions that were not obvious at first. Overall, I am glad I took this class as I feel I am more prepared to think critically in all parts of my life after taking it, not just in software development. I noticed my problem solving skills improving week by week and I hope to feel that improvement in other classes I take in the

future as well.

References

https://adam.math.hhu.de//g/leanprover-community/nng4
https://adam.math.hhu.de//g/trequetrum/lean4game-logic
https://chatgpt.com/
https://www.mathsisfun.com/games/towerofhanoi.html
https://www.overleaf.com/learn/latex/Learn$_L aTeX_i n_3 0_m inutes$