

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 | 3 |
| 2 | Week by Week | 3 |
| 2.1 | Week 1 | 3 |
| 2.1.1 | Notes and Exploration | 3 |
| 2.1.2 | Homework | 3 |
| 2.1.3 | Questions | 3 |
| 2.2 | Week 2 | 3 |
| 2.2.1 | Notes and Exploration | 3 |
| 2.2.2 | Homework | 3 |
| 2.2.3 | Questions | 5 |
| 2.3 | Week 3 | 5 |
| 2.3.1 | Notes and Exploration | 5 |
| 2.3.2 | Homework | 5 |
| 2.3.3 | Questions | 9 |
| 2.4 | Week 4 | 9 |
| 2.4.1 | Notes and Exploration | 9 |
| 2.4.2 | Homework | 9 |
| 2.4.3 | Questions | 11 |
| 2.5 | Week 5 | 11 |
| 2.5.1 | Notes and Exploration | 11 |
| 2.5.2 | Homework | 11 |
| 2.5.3 | Questions | 12 |
| 2.6 | Week 6 | 12 |
| 2.6.1 | Notes and Exploration | 12 |
| 2.6.2 | Homework | 12 |
| 2.6.3 | Questions | 15 |
| 2.7 | Week 7 | 15 |
| 2.7.1 | Notes and Exploration | 15 |
| 2.7.2 | Homework: Parse Trees | 15 |

| | | |
|----------|---|-----------|
| 2.7.3 | Questions | 17 |
| 2.8 | Week 8 | 17 |
| 2.8.1 | Notes and Exploration | 17 |
| 2.8.2 | Homework (NNG Levels 5–8) | 18 |
| 2.8.3 | Natural-Language Proof (English math proof) | 18 |
| 2.8.4 | Discord Question | 18 |
| 2.9 | Week 9 | 19 |
| 2.9.1 | Notes and Exploration | 19 |
| 2.9.2 | Homework (Addition World Level 5 / HW 9) | 19 |
| 2.9.3 | Questions | 21 |
| 2.10 | Week 10 | 21 |
| 2.10.1 | Notes and Exploration | 21 |
| 2.10.2 | Homework (Party Snacks, Levels 6–9) | 22 |
| 2.10.3 | Questions | 24 |
| 2.11 | Week 11 | 24 |
| 2.11.1 | Notes and Exploration | 24 |
| 2.11.2 | Homework (Negation Tutorial Levels 9–12) | 25 |
| 2.11.3 | Questions | 25 |
| 2.12 | Week 12 | 26 |
| 2.12.1 | Notes and Exploration | 26 |
| 2.12.2 | Homework (Towers of Hanoi) | 26 |
| 2.12.3 | Questions | 28 |
| 2.13 | Week 13 | 28 |
| 2.13.1 | Notes and Exploration | 28 |
| 2.13.2 | Homework | 29 |
| 2.13.3 | Questions | 29 |
| 3 | Essay (Synthesis) | 30 |
| 4 | Evidence of Participation | 30 |
| 5 | Conclusion | 30 |

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.

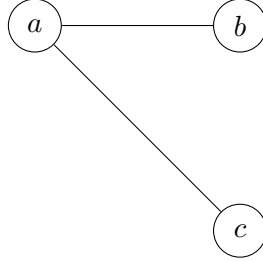
2.2.2 Homework

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

2. $A = \{a\}$, $R = \emptyset$ 
Normal forms: a . Terminating: True. Confluent: True. UNFs: True.


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

Infinite sequence $a \rightarrow a \rightarrow \dots$. Terminating: False. Confluent: True. UNFs: False.

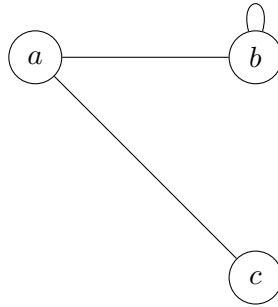


4. $A = \{a, b, c\}$, $R = \{(a, b), (a, c)\}$

b and c are normal forms, so from a two distinct endpoints are reachable. Terminating: True. Confluent: False. UNFs: False.

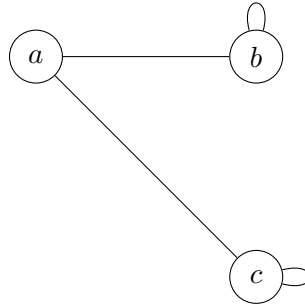
5. $A = \{a, b\}$, $R = \{(a, a), (a, b)\}$ 

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 $b \rightarrow b$; 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 | (\emptyset, \emptyset) | True | True | True |
| 2 | $(\{a\}, \emptyset)$ | 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 | <i>Impossible</i> |
| True | False | True | ARS 5 |
| True | False | False | ARS 3 |
| False | True | True | <i>Impossible</i> |
| False | True | False | ARS 4 |
| False | False | True | <i>Impossible</i> |
| 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 \rightarrow -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 \rightarrow is just one particular implementation.

2.3.2 Homework

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

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

(where ε is the empty string).

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

Example 1: abba.

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

So abba reduces to the normal form ε .

Example 2: bababa.

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

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

2. Why is the ARS not terminating?

Because there is an infinite rewrite loop:

$$\text{ab} \rightarrow \text{ba} \rightarrow \text{ab} \rightarrow \text{ba} \rightarrow \dots$$

So there exists an infinite \rightarrow -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.
- $aa \rightarrow \varepsilon$ changes the number of a 's by -2 , so parity is unchanged.
- $b \rightarrow \varepsilon$ 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 $b \rightarrow \varepsilon$. This leaves a string a^n . Then repeatedly apply $aa \rightarrow \varepsilon$:

$$a^{2k} \rightarrow \varepsilon, \quad a^{2k+1} \rightarrow a.$$

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:

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

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.

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

This is terminating because:

- each $ba \rightarrow ab$ decreases the number of “inversions” (a b sitting to the left of an a),
- and $aa \rightarrow \varepsilon$, $b \rightarrow \varepsilon$ 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 $aa \rightarrow \varepsilon$ with $aa \rightarrow a$:

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

1. Reduce the example strings again.

Example 1: $abba$.

$$abba \rightarrow aba \rightarrow aa \rightarrow a.$$

So the normal form is a .

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

$$bababa \rightarrow aaa \rightarrow aa \rightarrow a.$$

So the normal form is again a .

2. Why is the ARS not terminating?

The same infinite loop still exists:

$$ab \rightarrow ba \rightarrow ab \rightarrow \dots$$

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 $aa \rightarrow a$ 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,
- $b \rightarrow \varepsilon$ does not affect whether an a exists,
- $aa \rightarrow a$ 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 $aa \rightarrow a$ until only **a** remains.

So again the normal forms are:

$$\text{Normal forms: } \{\varepsilon, \mathbf{a}\}.$$

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:

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

Termination follows from the same measure idea (inversions decrease under $ba \rightarrow ab$ and length decreases under $aa \rightarrow a$ and $b \rightarrow \varepsilon$), 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 $\varphi(\text{state})$ 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

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

is a measure function for `merge_sort`.

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

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

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

Strict decrease on recursive calls. Let

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

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

$$\text{left} \leq \text{mid} < \text{right}.$$

Now compare the measures.

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

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

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

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

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

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

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

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

We compute AB .

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:

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

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

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

Let

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

Start:

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

Step-by-step reduction:

| | |
|--|--|
| $E_0 \mapsto \text{let fact} = (\text{fix } (\backslash \text{fact}. \backslash n. \text{if } n=0 \text{ then } 1 \text{ else } n * \text{fact } (n-1))) \text{ in fact } 3$ | $\langle \text{def of let rec} \rangle$ |
| $\mapsto (\lambda \text{fact}. \text{fact } 3) (\text{fix } F)$ | $\langle \text{def of let} \rangle$ |
| $\mapsto (\text{fix } F) 3$ | $\langle \beta \rangle$ |
| $\mapsto (F(\text{fix } F)) 3$ | $\langle \text{def of fix} \rangle$ |
| $\mapsto (\lambda n. \text{if } n = 0 \text{ then } 1 \text{ else } n * (\text{fix } F)(n - 1)) 3$ | $\langle \beta \rangle$ |
| $\mapsto \text{if } 3 = 0 \text{ then } 1 \text{ else } 3 * (\text{fix } F)(3 - 1)$ | $\langle \beta \rangle$ |
| $\mapsto \text{if False then } 1 \text{ else } 3 * (\text{fix } F)(3 - 1)$ | $\langle \text{def of } (= 0) \rangle$ |
| $\mapsto 3 * (\text{fix } F)(3 - 1)$ | $\langle \text{def of if} \rangle$ |
| $\mapsto 3 * (\text{fix } F) 2$ | $\langle \text{arith: } 3 - 1 \mapsto 2 \rangle$ |
| $\mapsto 3 * (F(\text{fix } F)) 2$ | $\langle \text{def of fix} \rangle$ |
| $\mapsto 3 * (\lambda n. \text{if } n = 0 \text{ then } 1 \text{ else } n * (\text{fix } F)(n - 1)) 2$ | $\langle \beta \rangle$ |
| $\mapsto 3 * (\text{if } 2 = 0 \text{ then } 1 \text{ else } 2 * (\text{fix } F)(2 - 1))$ | $\langle \beta \rangle$ |
| $\mapsto 3 * (\text{if False then } 1 \text{ else } 2 * (\text{fix } F)(2 - 1))$ | $\langle \text{def of } (= 0) \rangle$ |
| $\mapsto 3 * (2 * (\text{fix } F)(2 - 1))$ | $\langle \text{def of if} \rangle$ |
| $\mapsto 3 * (2 * (\text{fix } F) 1)$ | $\langle \text{arith: } 2 - 1 \mapsto 1 \rangle$ |
| $\mapsto 3 * (2 * (F(\text{fix } F)) 1)$ | $\langle \text{def of fix} \rangle$ |
| $\mapsto 3 * (2 * (\lambda n. \text{if } n = 0 \text{ then } 1 \text{ else } n * (\text{fix } F)(n - 1)) 1)$ | $\langle \beta \rangle$ |
| $\mapsto 3 * (2 * (\text{if } 1 = 0 \text{ then } 1 \text{ else } 1 * (\text{fix } F)(1 - 1)))$ | $\langle \beta \rangle$ |
| $\mapsto 3 * (2 * (\text{if False then } 1 \text{ else } 1 * (\text{fix } F)(1 - 1)))$ | $\langle \text{def of } (= 0) \rangle$ |
| $\mapsto 3 * (2 * (1 * (\text{fix } F)(1 - 1)))$ | $\langle \text{def of if} \rangle$ |
| $\mapsto 3 * (2 * (1 * (\text{fix } F) 0))$ | $\langle \text{arith: } 1 - 1 \mapsto 0 \rangle$ |
| $\mapsto 3 * (2 * (1 * (F(\text{fix } F)) 0))$ | $\langle \text{def of fix} \rangle$ |
| $\mapsto 3 * (2 * (1 * (\lambda n. \text{if } n = 0 \text{ then } 1 \text{ else } n * (\text{fix } F)(n - 1)) 0))$ | $\langle \beta \rangle$ |
| $\mapsto 3 * (2 * (1 * (\text{if } 0 = 0 \text{ then } 1 \text{ else } 0 * (\text{fix } F)(0 - 1))))$ | $\langle \beta \rangle$ |
| $\mapsto 3 * (2 * (1 * (\text{if True then } 1 \text{ else } 0 * (\text{fix } F)(0 - 1))))$ | $\langle \text{def of } (= 0) \rangle$ |
| $\mapsto 3 * (2 * (1 * 1))$ | $\langle \text{def of if} \rangle$ |
| $\mapsto 3 * (2 * 1)$ | $\langle \text{arith: } 1 * 1 \mapsto 1 \rangle$ |
| $\mapsto 3 * 2$ | $\langle \text{arith: } 2 * 1 \mapsto 2 \rangle$ |
| $\mapsto 6$ | $\langle \text{arith: } 3 * 2 \mapsto 6 \rangle$ |

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

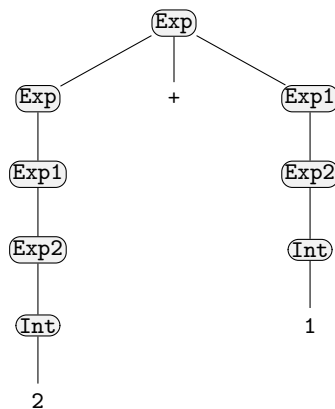
Parsing and context-free grammars. We use the grammar (nonterminals **Exp**, **Exp1**, **Exp2**, and **Int**):

```
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
```

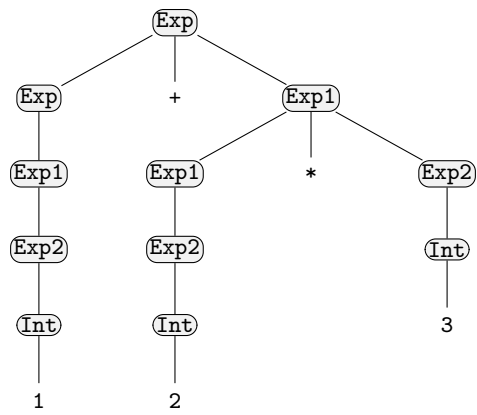
2.7.2 Homework: Parse Trees

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** \rightarrow **Int** \rightarrow digit.)

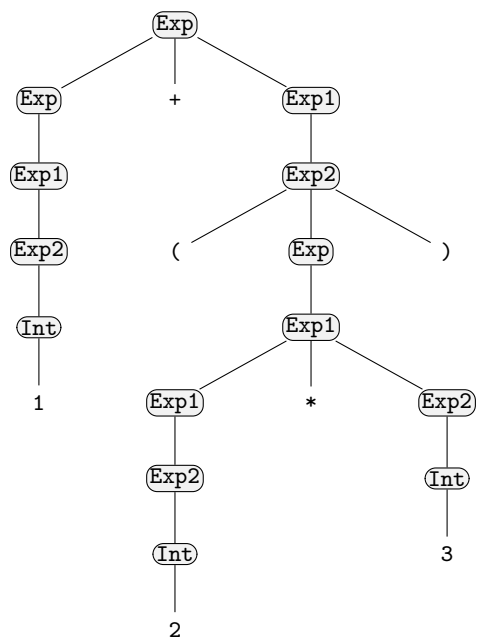
(a) 2+1



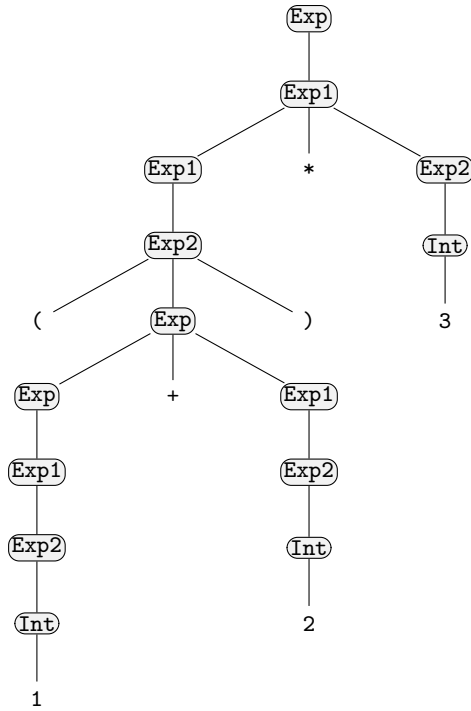
(b) 1+2*3



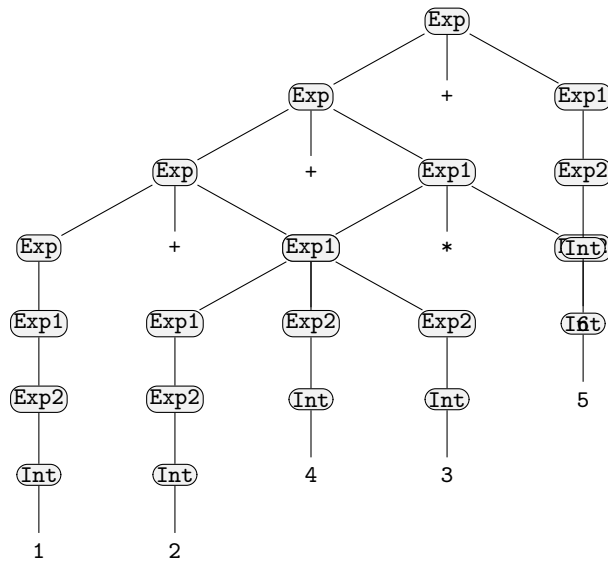
(c) $1+(2*3)$



(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:* $\text{succ } n = n + 1$.

Lean steps: `rw [one_eq_succ_zero]` to get $n + 1 = n + \text{succ } 0$; `rw [add_succ]` to obtain $\text{succ}(n + 0)$; `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 $\text{succ}(n)$ the successor of n . Define $1 = \text{succ}(0)$, $2 = \text{succ}(1)$, and $4 = \text{succ}(\text{succ}(\text{succ}(\text{succ}(0))))$. Addition is defined by the Peano axioms:

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

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

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

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

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

Therefore

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

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

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

Hence $2 + 2 = 4$. \square

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

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

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

Thus the goal reduces to

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

By the induction hypothesis, $(a + b) + d = (a + d) + b$, so the two $\text{succ}(\cdot)$ 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:

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

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,

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

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 $P \rightarrow Q$ 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

$$C \rightarrow D \rightarrow S$$

from the assumption

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

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.

$$\underbrace{\text{fun } c : C \Rightarrow}_{\text{1st argument}} \underbrace{\text{fun } d : D \Rightarrow}_{\text{2nd argument}} \underbrace{h (\text{and_intro } c \ d)}_{\text{call the given implication}}$$

What Lean checks.

- After `fun c =>` the goal becomes $D \rightarrow S$ (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 $C \wedge D \rightarrow S$, so $h(\text{and_intro } c \ d)$ 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

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

from the assumption

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

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 $h : C \rightarrow D \rightarrow S$, 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 → S`.
- `(h hcd.left) hcd.right : S`.

Final Lean line.

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

Level 8/9 (Distribute). Given.

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

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

Goal.

$$S \rightarrow C \wedge D$$

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* : $S \rightarrow C$ and apply it to s to get a C .
3. Use *h.right* : $S \rightarrow D$ 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 ∧ D`.

Final Lean line.

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

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

$$R \rightarrow (S \rightarrow R) \wedge (\neg S \rightarrow R).$$

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 $S \rightarrow R$.
4. Right part of the pair must have type $(\neg S) \rightarrow R$.
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.

$$\text{fun } r : R \Rightarrow \text{and_intro } \underbrace{(\text{fun } _ : S \Rightarrow r)}_{\text{has type } S \rightarrow R} \underbrace{(\text{fun } _ : \neg S \Rightarrow r)}_{\text{has type } S \rightarrow R}$$

What Lean checks.

- After `fun r =>` the goal is $(S \rightarrow R) \wedge (\neg S \rightarrow R)$.
- `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 $P \rightarrow \text{False}$.
- From a conjunction $P \wedge Q$, 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 $h : P \rightarrow \neg A$). **Goal.** $\neg(P \wedge A)$. **Given.** $h : P \rightarrow \neg A$.

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

Final Lean line.

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

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

- To show $A \rightarrow \text{False}$, 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 $h : \neg(B \rightarrow C)$. **Goal.** $\neg\neg B$, i.e. $(\neg B) \rightarrow \text{False}$. **Given.** $h : (B \rightarrow C) \rightarrow \text{False}$.

- Assume $nb : \neg B$. Define $f : B \rightarrow C$ by $f\ b := \text{false_elim}(nb\ b)$.
- Then $h\ f : \text{False}$. Hence $(\neg B) \rightarrow \text{False}$.

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

$$\text{hanoi } 1 \ x \ y = \text{move } x \ y$$
$$\text{hanoi } (n+1) \ x \ y = \text{hanoi } n \ x \ (\text{other } x \ y); \text{ move } x \ y; \text{ hanoi } n \ (\text{other } x \ y) \ y.$$

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:

$$\text{hanoi } 1 \ x \ y = \text{move } x \ y$$

$$\text{hanoi } (n+1) \ x \ y = \text{hanoi } n \ x \ (\text{other } x \ y); \text{ move } x \ y; \text{ hanoi } n \ (\text{other } x \ y) \ y.$$

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:

$$\text{hanoi } 1 \text{ } x \text{ } y = \text{move } x \text{ } y.$$

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)

(approx 1 page, plus references) The homework section gives you an opportunity to practice "skill drill" and to explore the material in more depth. The purpose of this section is to synthesize the knowledge you gained. Since Programming Languages is a wide field, it may be appropriate to focus on a particular topic of your choice. We suggest the following timeline. Week 5-8: Decide on a topic and discuss it with your instructor in the office hours. Write a summary email to your instructor after office hours, including the feedback you got with further reflection and planning. Week 9-12: Write a draft of your synthesis and discuss it with your instructor during office hours. Again, write a summary email to your instructor after office hours, including the feedback you got with further reflection and planning. The final version of the Synthesis is due with the rest of the final report.

4 Evidence of Participation

List your contributions to the course for your participation points

5 Conclusion

(approx 400 words) A critical reflection on the content of the course. Step back from the technical details. How does the course fit into the wider world of software engineering? What did you find most interesting or useful? What improvements would you suggest?

References