CPSC-354 Report

Nathan Garcia
Chapman University

October 27, 2025

Abstract

Contents

1 Introduction

2 Week by Week

2.1 Week 1

2.1.1 HW 1 - MU Puzzle

The MU puzzle comes from the book *Gödel, Escher, Bach*. You start with the string **MI** and the goal is to turn it into **MU** by following four rules:

1. If a string ends with **I**, you can add a **U** at the end.
2. If a string starts with **M**, you can copy everything after the M.
3. If you see **III**, you can change it to **U**.
4. If you see **UU**, you can delete it.

The puzzle is about seeing if you can reach **MU** by only using these rules. It is not really about the letters themselves, but about how rules control what strings you can or cannot make.

2.1.2 HW1

The MU puzzle comes from the book *Gödel, Escher, Bach*. You start with the string **MI** and the goal is to turn it into **MU** by following four rules:

1. If a string ends with **I**, you can add a **U** at the end.
2. If a string starts with **M**, you can copy everything after the M.
3. If you see **III**, you can change it to **U**.
4. If you see **UU**, you can delete it.

At first, I tried small derivations. For example:

$$\texttt{MI} \Rightarrow \texttt{MIU} \Rightarrow \texttt{MIUIU}$$

or duplicating I's:

$$\texttt{MI} \Rightarrow \texttt{MII} \Rightarrow \texttt{MIIII}$$

From `MIIII`, I can replace `III` with `U`, giving `MUI`, but not `MU`. Every time, an extra `I` is left over, and there is no rule that deletes a single `I`.

Invariant Argument. Let $\#I(w)$ denote the number of `I`'s in string w. If we track $\#I(w) \pmod 3$, we find:

- Rule 1: $\#I$ unchanged.
- Rule 2: $\#I$ doubles. Over $\mathbb{Z}/3\mathbb{Z}$, $1 \mapsto 2$, $2 \mapsto 1$, never 0.
- Rule 3: Removes 3 I's, leaving the remainder mod 3 unchanged.
- Rule 4: Deletes U's only, so $\#I$ unchanged.

We start with `MI`, which has $\#I = 1$. This is congruent to $1 \pmod 3$. Because no rule ever makes $\#I \equiv 0 \pmod 3$, it is impossible to reach a string with $\#I = 0$.

Conclusion. The target `MU` has $\#I = 0$, which is divisible by 3. Since that is unreachable from `MI`, the puzzle is unsolvable. As a student, the cool part here is that the solution isn't about brute-force trying rules—it's about spotting a hidden invariant (the number of I's mod 3) that blocks the path completely.

2.2 Week 2

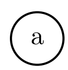
2.2.1 HW2 - Abstract Rewriting Systems (ARS) Properties

Problem. Consider the following list of Abstract Rewriting Systems (ARSs).

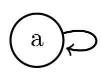
1. $A = \varnothing$.
2. $A = \{a\}$ and $R = \varnothing$.
3. $A = \{a\}$ and $R = \{(a, a)\}$.
4. $A = \{a, b, c\}$ and $R = \{(a, b), (a, c)\}$.
5. $A = \{a, b\}$ and $R = \{(a, a), (a, b)\}$.
6. $A = \{a, b, c\}$ and $R = \{(a, b), (b, b), (a, c)\}$.
7. $A = \{a, b, c\}$ and $R = \{(a, b), (b, b),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

Task. Draw a picture for each ARS above (nodes = elements of A, arrows = pairs in R). Then determine whether each ARS is *terminating*, *confluent*, and whether it has *unique normal forms*.

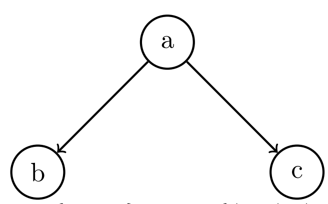
ARS 1: $A = \varnothing$ (no elements to draw)



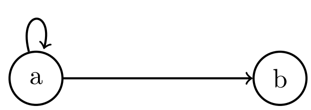
ARS 2: $A = \{a\},\ R = \varnothing$



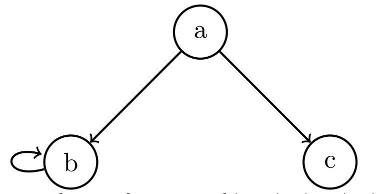
ARS 3: $A = \{a\},\ R = \{(a, a)\}$



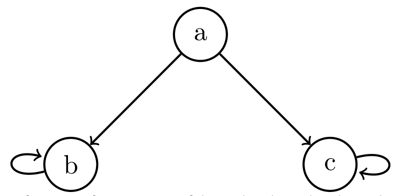
ARS 4: $A = \{a, b, c\},\ R = \{(a, b), (a, c)\}$



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



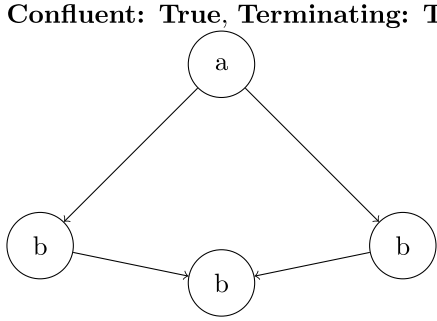
ARS 6: $A = \{a, b, c\}, \;R = \{(a, b), (b, b), (a, c)\}$



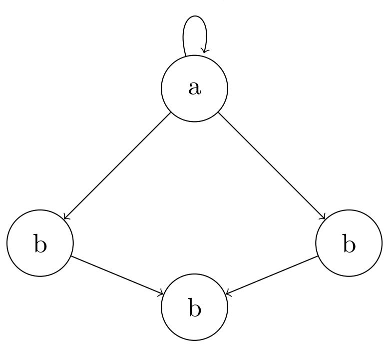
ARS 7: $A = \{a, b, c\}, \;R = \{(a, b), (b, b), (a, c), (c, c)\}$

ARS	**Terminating**	**Confluent**	**Has Unique Normal Forms**
1	X	X	X
2	X	X	X
3		X	
4	X		X
5		X	X
6			
7			

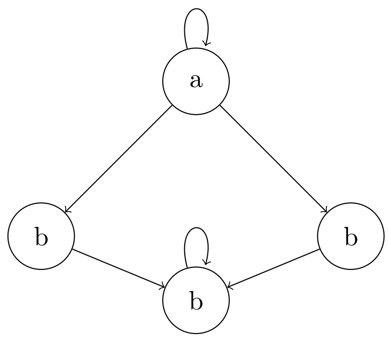
1. **Confluent: True**, **Terminating: True**, **Unique Normal Forms: False**



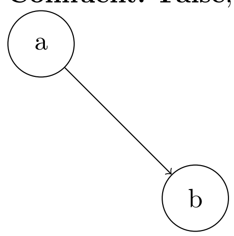
2. **Confluent: True**, **Terminating: False**, **Unique Normal Forms: True**



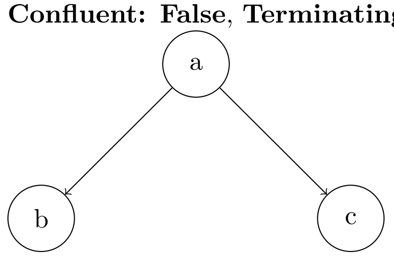
3. **Confluent: True**, **Terminating: False**, **Unique Normal Forms: False**



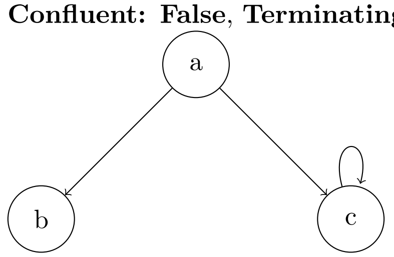
4. **Confluent: False**, **Terminating: True**, **Unique Normal Forms: True**



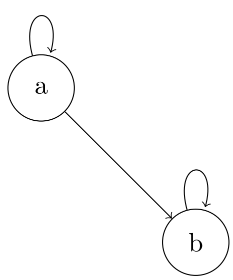
5. **Confluent: False**, **Terminating: True**, **Unique Normal Forms: False**



6. **Confluent: False**, **Terminating: False**, **Unique Normal Forms: True**



7. **Confluent: False**, **Terminating: False**, **Unique Normal Forms: False**



2.3 Week 3

2.3.1 HW 3 - Exercise 5: Reduction

Exercise 5.

Rules:

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

Sample reductions:

$$abba \;\to\; bbaa \;\to\; baa \;\to\; aa \;\to\; \epsilon$$

$$bababa \;\to\; aaabbb \;\to\; aabbb \;\to\; abbb \;\to\; a$$

Non-termination: The rules $ab \to ba$ and $ba \to ab$ form an infinite loop:

$$ab \to ba \to ab \to ba \to \cdots$$

Non-equivalent strings: a and ϵ are not equivalent, since a single a cannot be eliminated.

Equivalence classes: Order does not matter (due to swapping). b's vanish. $aa \to \epsilon$ ensures only the *parity* of the number of a's matters.

$$I(w) = \#a(w) \bmod 2 \in \{0, 1\}$$

Thus there are exactly two equivalence classes:

$$\{w \mid \#a(w) \equiv 0 \pmod 2\} \quad \longmapsto \quad \epsilon$$

$$\{w \mid \#a(w) \equiv 1 \pmod 2\} \quad \longmapsto \quad a$$

Modified terminating system:

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

Termination follows from length and inversion-count measures.

Specification: The algorithm computes the *parity of the number of a's*, ignoring b's.

Exercise 5b.

Rules:

$$ab \to ba, \quad ba \to ab, \quad aa \to a, \quad b \to \epsilon$$

Sample reductions:

$$abba \;\to\; bbaa \;\to\; baa \;\to\; aa \;\to\; a$$

$$bababa \;\to\; aaabbb \;\to\; aabbb \;\to\; abbb \;\to\; a$$

Non-termination: As before, infinite swapping is possible.

Non-equivalent strings: ϵ and a are not equivalent: all b's vanish, and any positive number of a's reduces to a.

Equivalence classes: Order does not matter. b's vanish. $aa \to a$ collapses any positive number of a's to a single a.

$$J(w) = \begin{cases} 0 & \text{if } \#a(w) = 0 \\ 1 & \text{if } \#a(w) \geq 1 \end{cases}$$

Thus there are exactly two equivalence classes:

$$\{w \mid \#a(w) = 0\} \quad \longmapsto \quad \epsilon$$

$$\{w \mid \#a(w) \geq 1\} \quad \longmapsto \quad a$$

Modified terminating system:

$$ab \to ba, \quad aa \to a, \quad b \to \epsilon$$

This terminates and yields unique normal forms.

Specification: The algorithm computes whether the input contains at least one a, ignoring all b's.

2.4 Week 4

2.4.1 HW4 - Termination

For the definition of a *measure function*, see our notes on rewriting and, in particular, on termination.

HW 4.1. Consider the following algorithm (Euclid's algorithm for the greatest common divisor):

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

Conditions. Assume inputs $a, b \in \mathbb{N}$ with $b \geq 0$ and the usual remainder operation, i.e. for $b > 0$ we have $0 \leq a \bmod b < b$. (If $b = 0$, the loop is skipped and the algorithm terminates immediately.)

Measure function. Define

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

Proof of termination. If the loop guard holds ($b \neq 0$), one iteration maps the state (a, b) to

$$(a', b') = (\, b, \; a \bmod b \,).$$

By the property of the remainder,

$$0 \leq b' = a \bmod b < b = \mu(a, b).$$

Thus μ strictly decreases on every loop iteration and is bounded below by 0. Since $(\mathbb{N}, <)$ is well-founded, no infinite descending chain

$$\mu(a_0, b_0) > \mu(a_1, b_1) > \mu(a_2, b_2) > \cdots$$

exists. Hence only finitely many iterations are possible; the loop terminates and the algorithm halts. $\square$

HW 4.2. Consider the following fragment of merge sort:

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

Define

$$\phi(left, right) := right - left + 1.$$

Claim. ϕ is a measure function for `merge_sort`.

Proof.

- *Well-defined, nonnegative.* For valid indices with $left \le right$, we have $\phi(left, right) \in \mathbb{N}$ and $\phi \ge 1$. If $left > right$ the function is not called (or $\phi \le 0$, and the base case applies immediately).
- *Base case.* When $left \ge right$, the function returns immediately; in this case $\phi(left, right) \le 1$, i.e. there is no further recursion.
- *Strict decrease on recursive calls.* Suppose $left < right$ and let $n := \phi(left, right) = right - left + 1 \ge 2$. With $mid = \lfloor (left + right)/2 \rfloor$:

  $$\phi(left, mid) = mid - left + 1 \le \left\lfloor \tfrac{n}{2} \right\rfloor < n,$$

  $$\phi(mid + 1, right) = right - (mid + 1) + 1 = right - mid \le \left\lceil \tfrac{n}{2} \right\rceil < n.$$

  Thus both recursive calls strictly reduce the measure.

Since ϕ maps each call to a natural number that strictly decreases along every recursive edge and is bounded below, there are no infinite descending chains. By well-founded induction on ϕ, all recursive calls terminate. $\square$

2.5 HW 5

2.5.1 Workout: Step-by-step α/β-reductions

Problem. Evaluate

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

Notation. We use $\rightsquigarrow_\beta$ for a single β-reduction step and "α" to indicate a capture-avoiding renaming of bound variables.

Intuition. The term $\lambda f.\, \lambda x.\, f(f\, x)$ applies a function twice (*iterate-2*). The term $\lambda f.\, \lambda x.\, f(f(f\, x))$ applies a function three times (*iterate-3*). Applying iterate-2 *to* iterate-3 yields iterate-9.

Derivation.

$$
\begin{aligned}
&(\lambda f.\, \lambda x.\, f(f\,x))\ (\lambda f.\, \lambda x.\, f(f(f\,x)))\\
&\rightsquigarrow_\beta\ \lambda x.\, \big[(\lambda f.\, \lambda x.\, f(f(f\,x)))\big((\lambda f.\, \lambda x.\, f(f(f\,x)))\,x\big)\big] \quad \text{(substitute } f := \lambda f.\lambda x.\, f(f(f\,x)) \text{ into } \lambda x.\, f(f\,x))\\
&\overset{\alpha}{\equiv}\ \lambda x.\, (\lambda f.\, \lambda y.\, f(f(f\,y)))\Big((\lambda f.\, \lambda u.\, f(f(f\,u)))\,x\Big) \quad \text{(rename bound } x\text{'s to } y, u \text{ to avoid shadowing)}\\
&\rightsquigarrow_\beta\ \lambda x.\, (\lambda f.\, \lambda y.\, f(f(f\,y)))\big(\lambda u.\, x(x(x\,u))\big) \quad \text{(apply } \beta \text{ to } (\lambda f.\lambda u.\, f(f(f\,u)))\,x)\\
&\rightsquigarrow_\beta\ \lambda x.\, \lambda y.\, F\big(F(F\,y)\big) \quad \text{with } F := \lambda u.\, x(x(x\,u)) \quad \text{(apply } (\lambda f.\lambda y.\, f(f(f\,y)))\,F)\\
&=\ \lambda x.\, \lambda y.\, F\Big(F\big(F\,y\big)\Big)\\
&=\ \lambda x.\, \lambda y.\, F\Big(F\big(x(x(x\,y))\big)\Big) \quad \text{(since } F\,y = x(x(x\,y)))\\
&=\ \lambda x.\, \lambda y.\, F\Big(x\big(x\big(x\big(x(x(x\,y))\big)\big)\big)\Big) \quad \text{(apply } F \text{ again; adds 3 more } x\text{'s: total 6)}\\
&=\ \lambda x.\, \lambda y.\, x\big(x\big(x\big(x\big(x\big(x\big(x(x(x\,y))\big)\big)\big)\big)\big)\big) \quad \text{(apply } F \text{ a third time; } +3 \text{ more: total 9).}
\end{aligned}
$$

Normal form.

$$\boxed{\lambda x.\, \lambda y.\ \underbrace{x(x(x(x(x(x(x(x(x\,y))))))))}_{\text{9 applications of } x}}$$

So the result is the *iterate-9* operator: given x and y, it applies x to y nine times.

2.6 HW 6

2.6.1 Computing $\mathrm{fact}\,3$ via a Fixed Point Combinator

We use the computation rules

$$
\begin{aligned}
&\texttt{fix}\ F\ \rightarrow\ (F\ (\texttt{fix}\ F)) && \text{(fix)}\\
&\texttt{let}\ x = e_1\ \texttt{in}\ e_2\ \rightarrow\ ((\lambda x.\, e_2)\ e_1) && \text{(let)}\\
&\texttt{let rec}\ f = e_1\ \texttt{in}\ e_2\ \rightarrow\ \texttt{let}\ f = (\texttt{fix}\ (\lambda f.\, e_1))\ \texttt{in}\ e_2 && \text{(let rec)}
\end{aligned}
$$

and the usual β-reduction $((\lambda x.\, e)\ v) \rightarrow e[x := v]$, plus base computation rules

$0 = 0 \rightarrow \texttt{True}$, $\quad n>0 \Rightarrow (n = 0) \rightarrow \texttt{False}$, $\quad \texttt{if True then}\ A\ \texttt{else}\ B \rightarrow A$, $\quad \texttt{if False then}\ A\ \texttt{else}\ B \rightarrow B$.

Abbreviation. Let

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

Then $\texttt{fact} \equiv \texttt{fix}\ F$.

Goal. Evaluate

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

$$
\begin{aligned}
& \texttt{let rec fact} = \lambda n.\ \ldots\ \texttt{in fact}\ 3 && \\
\to\ & \texttt{let fact} = (\texttt{fix}\ (\lambda f.\,\lambda n.\ \ldots))\ \texttt{in fact}\ 3 && \text{(let rec)} \\
\to\ & ((\lambda \texttt{fact}.\,\texttt{fact}\ 3)\ (\texttt{fix}\ F)) && \text{(let)} \\
\to\ & (\texttt{fix}\ F)\ 3 && (\beta) \\
\to\ & (F\ (\texttt{fix}\ F))\ 3 && \text{(fix)} \\
\to\ & ((\lambda f.\,\lambda n.\,\texttt{if}\ (n=0)\ \texttt{then}\ 1\ \texttt{else}\ n * f(n-1))\ (\texttt{fix}\ F))\ 3 && \text{(def. of } F\text{)} \\
\to\ & (\lambda n.\,\texttt{if}\ (n=0)\ \texttt{then}\ 1\ \texttt{else}\ n * (\texttt{fix}\ F)(n-1))\ 3 && (\beta) \\
\to\ & \texttt{if}\ (3=0)\ \texttt{then}\ 1\ \texttt{else}\ 3 * (\texttt{fix}\ F)(2) && (\beta) \\
\to\ & 3 * (\texttt{fix}\ F)(2) && \text{(arith. and if-False)}
\end{aligned}
$$

Now expand ($\texttt{fix}$ F) 2:

$$
\begin{aligned}
(\texttt{fix}\ F)2 &\to (F(\texttt{fix}\ F))2 && \text{(fix)} \\
&\to \texttt{if}\ (2=0)\ \texttt{then}\ 1\ \texttt{else}\ 2 * (\texttt{fix}\ F)(1) && \text{(def. } F,\ \beta) \\
&\to 2 * (\texttt{fix}\ F)(1) && \text{(if-False)}
\end{aligned}
$$

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

$$
\begin{aligned}
(\texttt{fix}\ F)1 &\to (F(\texttt{fix}\ F))1 && \text{(fix)} \\
&\to \texttt{if}\ (1=0)\ \texttt{then}\ 1\ \texttt{else}\ 1 * (\texttt{fix}\ F)(0) && \text{(def. } F,\ \beta) \\
&\to 1 * (\texttt{fix}\ F)(0) && \text{(if-False)}
\end{aligned}
$$

Expand ($\texttt{fix}$ F)0:

$$
\begin{aligned}
(\texttt{fix}\ F)0 &\to (F(\texttt{fix}\ F))0 && \text{(fix)} \\
&\to \texttt{if}\ (0=0)\ \texttt{then}\ 1\ \texttt{else}\ 0 * (\texttt{fix}\ F)(-1) && \text{(def. } F,\ \beta) \\
&\to 1 && \text{(if-True)}
\end{aligned}
$$

Unwinding:

$$
1 * (\texttt{fix}\ F)0 \to 1 * 1 \to 1, \qquad 2 * (\texttt{fix}\ F)1 \to 2 * 1 \to 2, \qquad 3 * (\texttt{fix}\ F)2 \to 3 * 2 \to 6.
$$

$$
\boxed{\texttt{fact}\ 3\ \to^*\ 6}
$$

2.7 Week 7

2.7.1 HW7 - Parse Trees for Arithmetic Expressions

Using the context-free grammar:

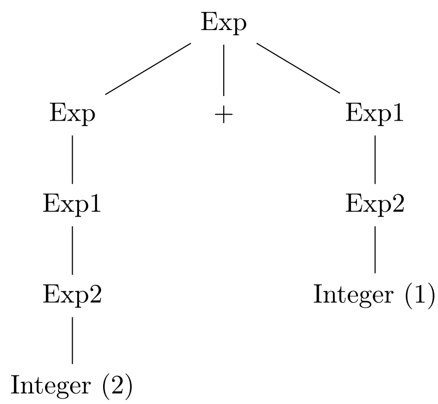
```
Exp  -> Exp '+' Exp1
Exp1 -> Exp1 '*' Exp2
Exp2 -> Integer
Exp2 -> '(' Exp ')'
Exp  -> Exp1
Exp1 -> Exp2
```

Write out the derivation trees (parse trees) for the following strings:

2+1, 1+2*3, 1+(2*3), (1+2)*3, 1+2*3+4*5+6

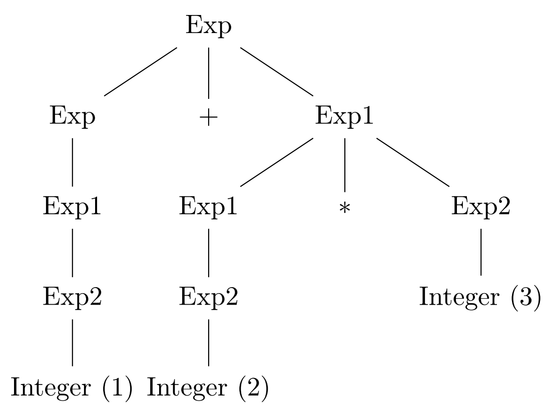
—

1. Parse tree for 2 + 1



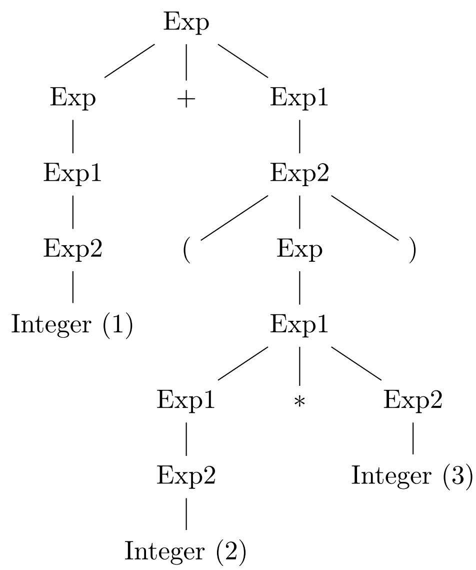
—

2. Parse tree for 1 + 2 * 3



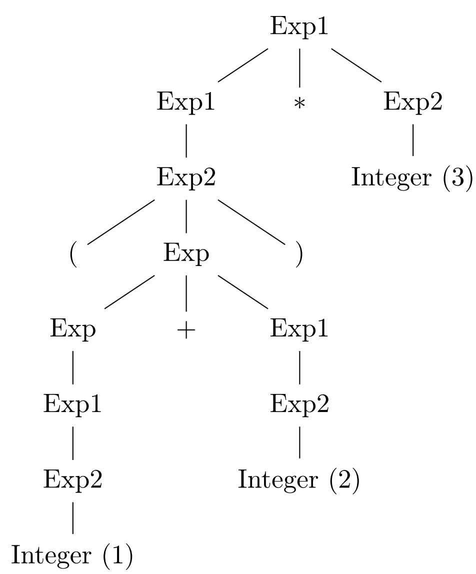
—

3. Parse tree for 1 + (2 * 3)



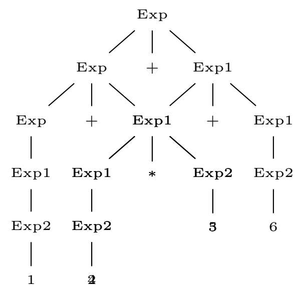
—

4. Parse tree for (1 + 2) * 3



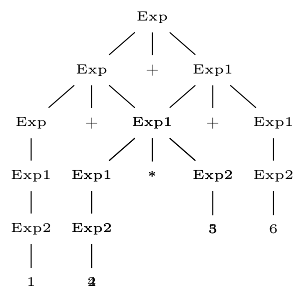
—

5. Parse tree for 1 + 2 * 3 + 4 * 5 + 6



—

5. Parse tree for 1 + 2 * 3 + 4 * 5 + 6



—

Notes:

- Each derivation strictly follows the grammar rules.
- Multiplication has higher precedence than addition, as enforced by the separation of `Exp` and `Exp1`.
- Parentheses in examples (3) and (4) override default precedence by forcing evaluation order.

2.8 Week 7

2.8.1 HW 8 Natural Numbers Game

Problem: 5

Prove that

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

for all natural numbers $a, b, c \in \mathbb{N}$.

Solution in Lean:

```
example (a b c : ℕ) : a + (b + 0) + (c + 0) = a + b + c := by
  rw [add_zero]
  rw [add_zero]
  rfl
```

Explanation:

- The lemma `add_zero` states that $x + 0 = x$ for any natural number x.
- The first `rw [add_zero]` simplifies $b + 0$ to b.

- The second `rw [add_zero]` simplifies $c + 0$ to c.
- Finally, `rfl` (reflexivity) completes the proof since both sides are identical.

Proof. On the natural numbers, addition is defined so that $x + 0 = x$ for every x (the identity law for 0), and the rest of addition is built recursively. Applying the identity law with $x = b$ gives $b + 0 = b$, and with $x = c$ gives $c + 0 = c$. Substituting these equalities into the left-hand side yields

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

Both sides are now the same expression, so the equality holds. □

Problem: 6

Prove that

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

for all natural numbers $a, b, c \in \mathbb{N}$, but this time tell Lean to simplify the $c + 0$ term first.

Solution

```
example (a b c : ℕ) : a + (b + 0) + (c + 0) = a + b + c := by
  rw [add_zero c]
  rw [add_zero b]
  rfl
```

Explanation:

- The lemma `add_zero x` proves that $x + 0 = x$.
- Writing `rw [add_zero c]` explicitly tells Lean to apply this lemma to the term $c + 0$ first.
- Then `rw [add_zero b]` simplifies $b + 0$ to b.
- Finally, `rfl` completes the proof since both sides are identical.

Result: The equality holds, and the proof demonstrates how to use precision rewriting in Lean.

Problem. 7 Prove that for all natural numbers n,

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

Lean solution:

```
theorem succ_eq_add_one (n : ℕ) : succ n = n + 1 := by
  rw [one_eq_succ_zero]
  rw [add_succ]
  rw [add_zero]
  rfl
```

Explanation:

- We begin by rewriting 1 as succ(0) using `one_eq_succ_zero`.
- Next, we apply `add_succ` to expand $n + \text{succ}(0)$ into $\text{succ}(n + 0)$.
- Then, the lemma `add_zero` simplifies $n + 0$ to n.
- Finally, `rfl` (reflexivity) confirms that both sides are equal, proving that $\text{succ}(n) = n + 1$.

Problem 8: Prove that
$$2 + 2 = 4.$$

Lean solution:

```
example : 2 + 2 = 4 := by
  rw [two_eq_succ_one]
  rw [add_succ]
  rw [add_one_eq_succ]
  rfl
```

Explanation:

- We first rewrite 2 as $\mathrm{succ}(1)$ using `two_eq_succ_one`.
- Then `add_succ` expands the addition: $2 + 2 = \mathrm{succ}(1 + 1)$.
- Next, `add_one_eq_succ` converts $1 + 1$ into $\mathrm{succ}(1)$.
- Finally, `rfl` (reflexivity) confirms both sides are equal, completing the proof that $2 + 2 = 4$.

2.9 Week9

2.9.1 hw9

Level 5: `add_right_comm`

Theorem. For all natural numbers a, b, c, we have
$$(a + b) + c = (a + c) + b.$$

This property is known as the *right commutativity of addition.*

Solution 1 (Using Induction). We can prove this by performing induction on one of the variables, for example c.

Base Case: Let $c = 0$. Then
$$(a + b) + 0 = a + b = (a + 0) + b,$$
where we use the identity property of addition ($x + 0 = x$ and $0 + x = x$).

Inductive Step: Assume the statement holds for some $c = k$, i.e.
$$(a + b) + k = (a + k) + b.$$

We must show it holds for $c = k + 1$. Then:
$$\begin{aligned}
(a + b) + (k + 1) &= ((a + b) + k) + 1 && \text{(by definition of addition)} \\
&= ((a + k) + b) + 1 && \text{(by inductive hypothesis)} \\
&= (a + k) + (b + 1) && \text{(by associativity)} \\
&= (a + (k + 1)) + b && \text{(by definition of addition).}
\end{aligned}$$

Thus, by induction, $(a + b) + c = (a + c) + b$ for all $c \in \mathbb{N}$.

Lean-style Inductive Proof:

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

—

Solution 2 (Without Induction). We can also prove $(a+b)+c = (a+c)+b$ *without induction*, by using the results we have already established: the **associativity** and **commutativity** of addition.

Proof:

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

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

Lean-style Non-Inductive Proof:

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

3 Essay

4 Evidence of Participation

5 Conclusion

References

[BLA] Author, Title, Publisher, Year.