CPSC-354 Report

Junho Yi
Chapman University

October 26, 2025

Abstract

Contents

1 Introduction

2 Week by Week

2.1 Week 1

2.1.1 Notes and Exploration

Place holder for notes

In abstract rewriting, an object is in normal form if it cannot be rewritten any further, i.e. it is irreducible

Confluence system: in a system the result eventually converges into the same answer.

Termination: Means the system stops at some point.

Decidability:

church turing thesis:

Abstract rewriting system(ARS): mathematically the same as a directed graph A is a set of "strings" (can be anything) R is the relation

so in the MIU puzzle, A is M I U, the strings we use then R is the rules we are given. ie: (Mx,Mxx)|x e A U...

2.1.2 Homework 1

This week's HW is regarding the MU puzzle, and its relevance and application to formal systems. Here we use the MU puzzle to practice and familiarize ourselves with staying within the confines of a formal system. We are given 4 rules/restrictions, which is referred to as the "Requirement of Formality."

Our formal system consists of these 4 rules:

1. **RULE I**: If you possess a string whose last letter is I, you can add on a U at the end.
2. **RULE II**: Suppose you have Mx. Then you may add Mxx to your collection.
3. **RULE III**: If III occurs in one of the strings in your collection, you may make a new string with U in place of III.
4. **RULE IV**: If UU occurs inside one of your strings, you can drop it.

With these four rules in mind, we have one objective: stay within the rules and produce MU from MI.

As I worked through the rules, I logically deduced these points in this order:

1. When applying RULE II, if I exists somewhere in the string, the parity of I becomes even until RULE III is applied again.
2. When applying RULE III, the I's (which are even, if RULE III applies) swap parity, i.e. even $\mapsto$ odd.
3. The lowest continuous string of I's where RULE III can be applied is $IIII$ (four I's).
4. Because RULE III is the only way to reduce the number of I's, and it is only possible to apply RULE III if there is a minimum of four continuous I's (due to RULE II) and an even parity of I's, using RULE III to reduce the amount of I's will always result in a remainder (a leftover I).
5. Therefore, you can never get rid of I's fully with RULE III, or any other RULE usable by us without modifications of the rules.

From the above observations, we can see that there is no way to completely reduce the number of I's into zero with the given rules. This is my personal analysis of the MU puzzle; below is the "correct" analysis of the MU puzzle.

Proof (invariant mod 3). Let n be the number of I's in the current string. Then:

$$\text{Rule I: } n \mapsto n, \quad \text{Rule II: } n \mapsto 2n, \quad \text{Rule III: } n \mapsto n-3, \quad \text{Rule IV: } n \mapsto n.$$

Hence n mod 3 is preserved by Rules I, III, IV, and toggles between 1 and 2 under Rule II. Initially, MI has $n = 1 \equiv 1 \pmod 3$. No sequence of the above operations can yield $n \equiv 0 \pmod 3$. But MU has 0 I's, i.e. $n = 0 \equiv 0 \pmod 3$. Therefore MU is not derivable from MI. □

2.2 Week 2, Rewriting theory

2.2.1 Notes and Exploration

placeholder for notes

2.2.2 HW 2

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

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

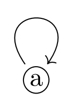
1.

Terminating ✓ | Confluent ✓ | Unique normal forms ✓

2.

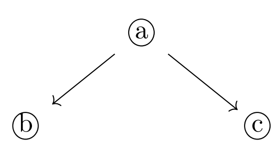
Terminating ✓ | Confluent ✓ | Unique normal forms ✓

3.



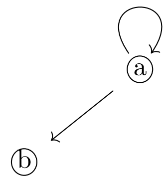
Terminating × | Confluent ✓ | Unique normal forms ×

4.



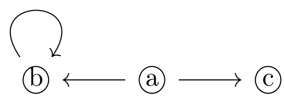
Terminating ✓ | Confluent × | Unique normal forms ×

5.



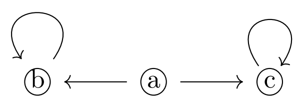
Terminating $\times$ | Confluent ✓ | Unique normal forms ✓

6.



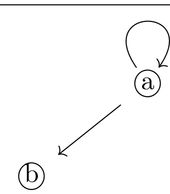
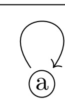
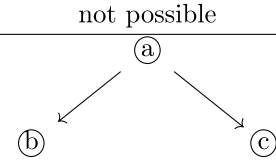
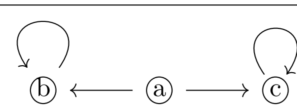
Terminating $\times$ | Confluent $\times$ | Unique normal forms $\times$

7.



Terminating $\times$ | Confluent $\times$ | Unique normal forms $\times$

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

confluent	**terminating**	**has unique normal forms**	**example**
True	True	True	◯ , ⓐ
True	True	False	not possible
True	False	True	
True	False	False	
False	True	True	not possible
False	True	False	
False	False	True	not possible
False	False	False	

2.3 Week 3, String rewriting

2.3.1 HW 3

For this week's homework, we are tasked with answering the questions on Exercises 5 and 5b.

Exercise 5:

- Reduce some example strings such as `abba` and `bababa`.

Reduce `abba`	**Reduce** `bababa`
`abba`	`bababa`
`aba`	`ababa`
`aa`	`aaba`
ε	`aaa`
	`a`

- **Why is the ARS not terminating?**

  The ARS does not terminate because the system is not strongly normalizing. Rules 1 and 2 are bi-directional swap rules, thus there exists an infinite reduction.

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

  `a` and ε. If the question is asking for "how many strings not equivalent to ε," then we can find an infinite number of non-equivalent strings (i.e., any string with odd number of `a`'s vs even number of `a`'s).

- **How many equivalence classes does $\overset{*}{\longleftrightarrow}$ have? Can you describe them in a nice way? What are the normal forms?**

  The normal forms are `a` and ε, the two equivalence classes are strings with even number of `a`'s and odd number of `a`'s.

- **Can you modify the ARS so that it becomes terminating without changing its equivalence classes?**

  We have to get rid of either rule 1 or 2, to make the ARS strongly normalizing.

- **Write down a question or two about strings that can be answered using the ARS. Think about whether this amounts to giving a semantics to the ARS.**

  [Hint: The best answers are likely to involve a complete invariant.]

  Remark: A characterization of the equivalence classes that mentions the reduction relation is not interesting.

  Question: "Does this string have an even or odd number of `a`'s? Which equivalence class does even/odd `a`'s belong to?"

Excercise 5b:

- Reduce some example strings such as `abba` and `bababa`.

Reduce `abba`	**Reduce** `bababa`
`abba`	`bababa`
`aba`	`ababa`
`aa`	`aaba`
`a`	`aaa`
	`aa`
	`a`

- **Why is the ARS not terminating?**

  The ARS does not terminate because the system is not strongly normalizing. Rules 1 and 2 are bi-directional swap rules, thus there exists an infinite reduction.

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

  `a` and ε. If the question is asking for "how many strings not equivalent to ε," then there are infinitely many (e.g., any string containing at least one `a`).

- **How many equivalence classes does $\stackrel{*}{\longleftrightarrow}$ have? Can you describe them in a nice way? What are the normal forms?**

  The normal forms are `a` and ε. The two equivalence classes are: strings with *no* `a`'s (equivalent to ε), and strings with *at least one* `a` (equivalent to `a`).

- **Can you modify the ARS so that it becomes terminating without changing its equivalence classes?**

  We have to get rid of either rule 1 or 2, to make the ARS strongly normalizing.

- **Write down a question or two about strings that can be answered using the ARS. Think about whether this amounts to giving a semantics to the ARS.**

  [Hint: The best answers are likely to involve a complete invariant.]

  Remark: A characterization of the equivalence classes that mentions the reduction relation is not interesting.

  Question: "Does this string have an `a` or not? Which equivalence class do 'no `a`' and 'some `a`' belong to?"

2.4 Week 4, Termination

HW 4 (Termination) [kurz-hw4]

Below I use the simple "measure gets smaller" idea from the notes on termination [kurz-termination]: pick something that you can track after each step. If that thing always gets smaller and can never go down forever, the process must stop.

4.1 (Euclid's algorithm for greatest common divisor). Code we are talking about

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

Why it always stops

- Think of `b` as our "measure." We just watch the value of `b`.
- Each time through the loop we set `b` to the remainder when `a` is divided by `b`.
- A remainder is always a smaller nonnegative number than the thing you divide by. So new `b` is always smaller than the old `b`, and it never becomes negative.
- Because `b` keeps shrinking and cannot shrink forever, it will eventually reach zero.
- When `b` becomes zero, the loop condition `b != 0` is false and the program stops.

Keep track of `b`; it strictly goes down each step; it must hit zero and stop.

4.2 (Merge sort). Code shape we are talking about

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

Why it always stops

- Our measure is the length of the current subarray, from `left` to `right`.
- If the length is zero or one, we return immediately. No more calls.
- Otherwise, we split the range into two halves. Each half is strictly smaller than the original range.
- The two recursive calls work on these smaller halves. Since the lengths keep getting smaller, we cannot keep splitting forever.
- Eventually every piece is length zero or one, so the base case triggers and all calls finish.
- The `merge` step does not call `merge_sort` again, so it does not affect whether the algorithm stops.

Each recursive step makes "current length" smaller. You cannot reduce the length forever, so the process reaches the base case and stops.

2.5 Week 5, Lambda Calculus, Syntax and Semantics

HW 5 (Lambda Calculus, Syntax and Semantics) [kurz-hw5]

Task. Reduce the following by *normal order* (leftmost–outermost), keeping the outer abstractions intact and using α-renaming only to avoid clashes:

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

My steps.

1.
$$(\lambda f.\,\lambda x.\,f(f\,x))\;(\lambda f.\,\lambda x.\,f(f(f\,x)))\quad \text{(renaming note: set } G \equiv (\lambda f.\,\lambda x.\,f(f(f\,x))); \alpha\text{-rename inner } x \to y \text{ if needed)}$$

2.
$$(\lambda x.\,(\lambda f.\,\lambda y.\,f(f(f\,y)))\;((\lambda f.\,\lambda y.\,f(f(f\,y)))\;x))\quad (\lambda x. \text{ is the function; inside is the body with two applications } A \text{ and } B)$$
$$\text{here } A \;:=\; (\lambda f.\,\lambda y.\,f(f(f\,y))), \qquad B \;:=\; ((\lambda f.\,\lambda y.\,f(f(f\,y)))\;x), \quad \text{apply } \beta \text{ only to applications.}$$

3. — *rename inner y into u:*
$$(\lambda x.\,(\lambda f.\,\lambda u.\,f(f(f\,u)))\;((\lambda f.\,\lambda y.\,f(f(f\,y)))\;x))$$

4.
$$(\lambda x.\,(\lambda u.\,((\lambda f.\,\lambda y.\,f(f(f\,y)))\;x)\;(((\lambda f.\,\lambda y.\,f(f(f\,y)))\;x)\;(((\lambda f.\,\lambda y.\,f(f(f\,y)))\;x)\;u))))$$

5. $$(\lambda x.\,(\lambda u.\,((\,(\lambda y.\,(x\,(x\,(x\,y)))\,)\,((\lambda y.\,(x\,(x\,(x\,y))))\,((\lambda y.\,(x\,(x\,(x\,y))))\,u)))))$$

6. *get rid of unneeded parentheses:*

$$\lambda x.\,\lambda u.\,x\big(x\big(x\,\big((\lambda y.\,x(x(x\,y)))\,((\lambda y.\,x(x(x\,y)))\,u)\big)\big)\big)$$

7. $$\lambda x.\,\lambda u.\,x\big(x\big(x\big(x\big(x\big(x\,((\lambda y.\,x(x(x\,y)))\,u)\big)\big)\big)\big)\big)$$

8. $$\lambda x.\,\lambda u.\,x\big(x\big(x\big(x\big(x\big(x\big(x\big(x\big(x\,u\big)\big)\big)\big)\big)\big)\big)\big)$$

9. $$\lambda x.\,\lambda u.\,x\big(x\big(x\big(x\big(x\big(x\big(x\big(x\big(x\,u\big)\big)\big)\big)\big)\big)\big)\big)$$

Result. The final line is in normal form (no β-redexes remain): every application is headed by the variable x, not by a λ.

2.6 Week 6, Lambda Calculus, Syntax and Semantics

HW 6 (Fixed Point Combinator / `fix`)

start:

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

rules:

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

1:

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

$$e_1 = \lambda n.\ \text{if } n = 0 \text{ then } 1 \text{ else } n * \mathsf{fact}(n-1) \quad ||| \quad e_2 = \mathsf{fact}\ 3$$

let $\mathsf{fact} = \Big(\texttt{fix}\ (\lambda f.\,\lambda n.\ \text{if } n = 0 \text{ then } 1 \text{ else } n{*}\mathsf{fact}(n{-}1))\Big)$ in $\mathsf{fact}\ 3$ $\leftarrow$ *should have alpha renamed up here aka* $\big($*let* $\mathsf{fact} = (f$

$$e_1 = \texttt{fix}\ (\lambda f.\,\lambda n.\ \text{if } n = 0 \text{ then } 1 \text{ else } n * \mathsf{fact}(n-1)) \quad |||| \quad e_2 = \mathsf{fact}\ 3$$

2: let $x = e_1$ **in** $e_2 \to (\lambda x.\,e_2)\,e_1$

$$x = \mathsf{fact} \quad \Big| \quad e_1 = \texttt{fix}\ (\lambda f.\,\lambda n.\ \text{if } n = 0 \text{ then } 1 \text{ else } n * \mathsf{fact}(n-1)) \quad \Big| \quad e_2 = \mathsf{fact}\ 3$$

$$(\lambda\mathsf{fact}.\ \mathsf{fact}\ 3)\ \Big(\texttt{fix}\ (\lambda f.\,\lambda n.\ \text{if } n = 0 \text{ then } 1 \text{ else } n * \mathsf{fact}(n-1))\Big) \ \leftarrow \textit{end of step 2}$$

with alpha rename: $(\lambda\mathsf{fact}.\ \mathsf{fact}\ 3)\ \big(\texttt{fix}\ (\lambda f.\,\lambda n.\ \text{if } n = 0 \text{ then } 1 \text{ else } n * f(n-1))\big)$

3:

$\big(\texttt{fix}\ (\lambda f.\,\lambda n.\ \text{if } n = 0 \text{ then } 1 \text{ else } n{*}\mathsf{fact}(n{-}1))\big)\ 3$ *with alpha rename:* $\big(\texttt{fix}\ (\lambda f.\,\lambda n.\ \text{if } n = 0 \text{ then } 1 \text{ else } n{*}f(n{-}1))\big)\ 3$ *and*

4<def of fix>:

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

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

5<beta rule: substitute $\texttt{fix}\ F$>:

$(\lambda n.\ \text{if } n = 0 \text{ then } 1 \text{ else } n{*}(\texttt{fix}\ (\lambda \mathsf{fact}.\, \lambda n.\ \text{if } n = 0 \text{ then } 1 \text{ else } n{*}\mathsf{fact}(n{-}1)))\ (n{-}1))\ 3$ $\leftarrow$ *should have alpha renamed the fix*

6<beta ruls: substitute 3>:

$\text{if } 3 = 0 \text{ then } 1 \text{ else } 3{*}(\texttt{fix}\ (\lambda \mathsf{fact}.\, \lambda n.\ \text{if } n = 0 \text{ then } 1 \text{ else } n{*}\mathsf{fact}(n{-}1)))\ (3{-}1)$ $\leftarrow$ *from here on out dont sub n inside fix funct*

7<def of if>:

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

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

8<def of fix>:

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

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

9<beta rule: substitute $\texttt{fix}\ F$>:

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

10<beta ruls: substitute 2>:

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

11<def of if>:

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

12<def of fix>:

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

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

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

13<beta rule: substitute $\texttt{fix}\ F$>:

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

14<beta ruls: substitute 1>:

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

14<def of if>:

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

15<def of fix>:

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

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

16<beta rule: substitute `fix` F>:

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

17<beta ruls: substitute 0>:

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

18<def of if>:

$$(3 * (2 * (1 * 1)))$$

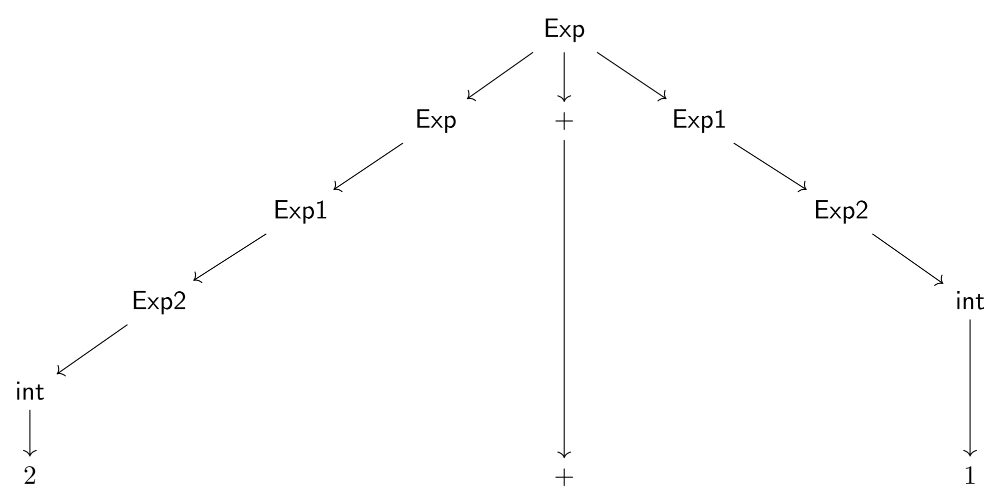
19

$$(3 * 2 * 1 * 1)$$

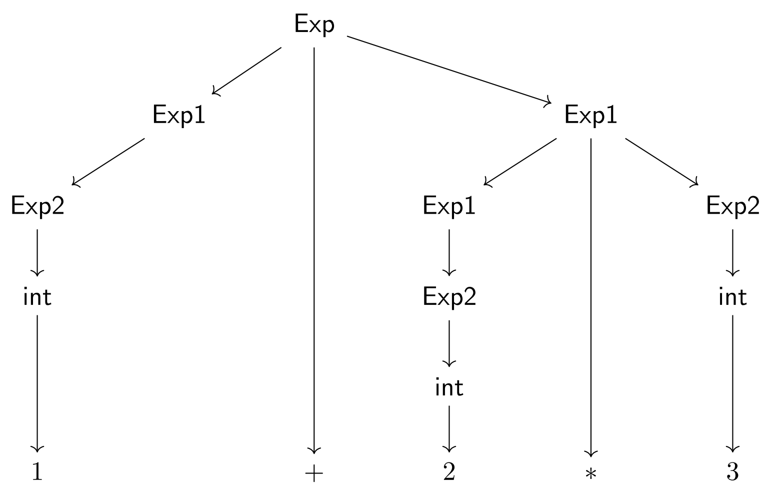
2.7 Week 7, context-free grammar

HW 7 (derivation trees)

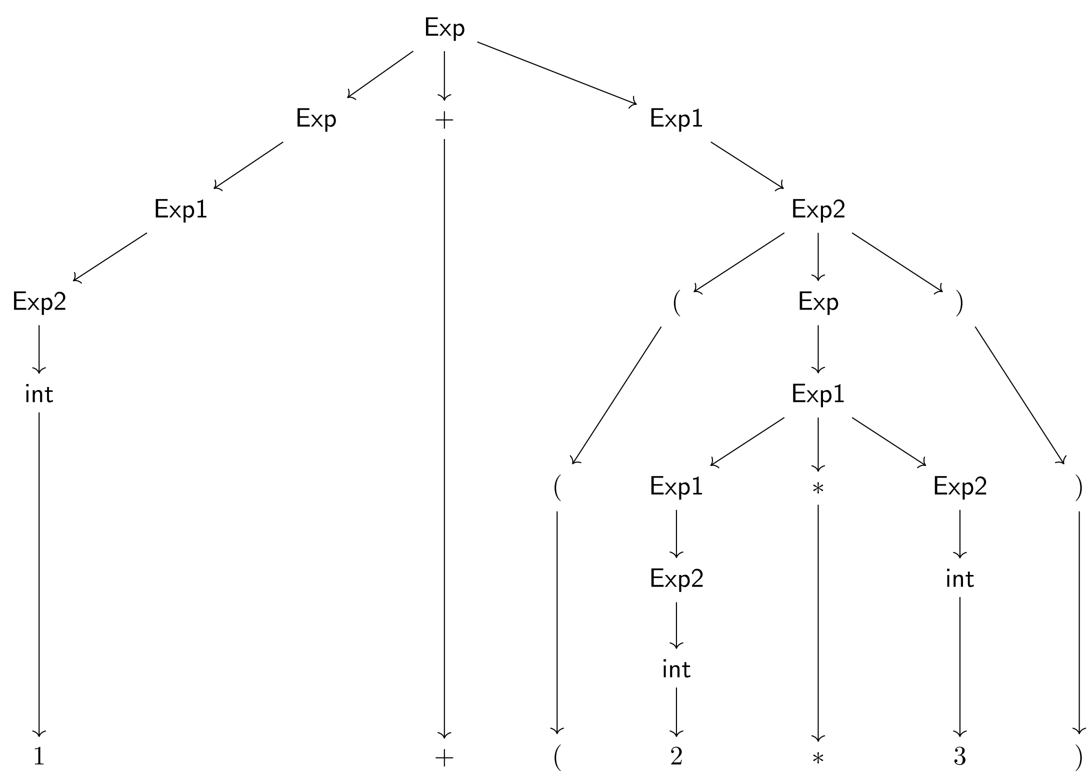
Tree #1



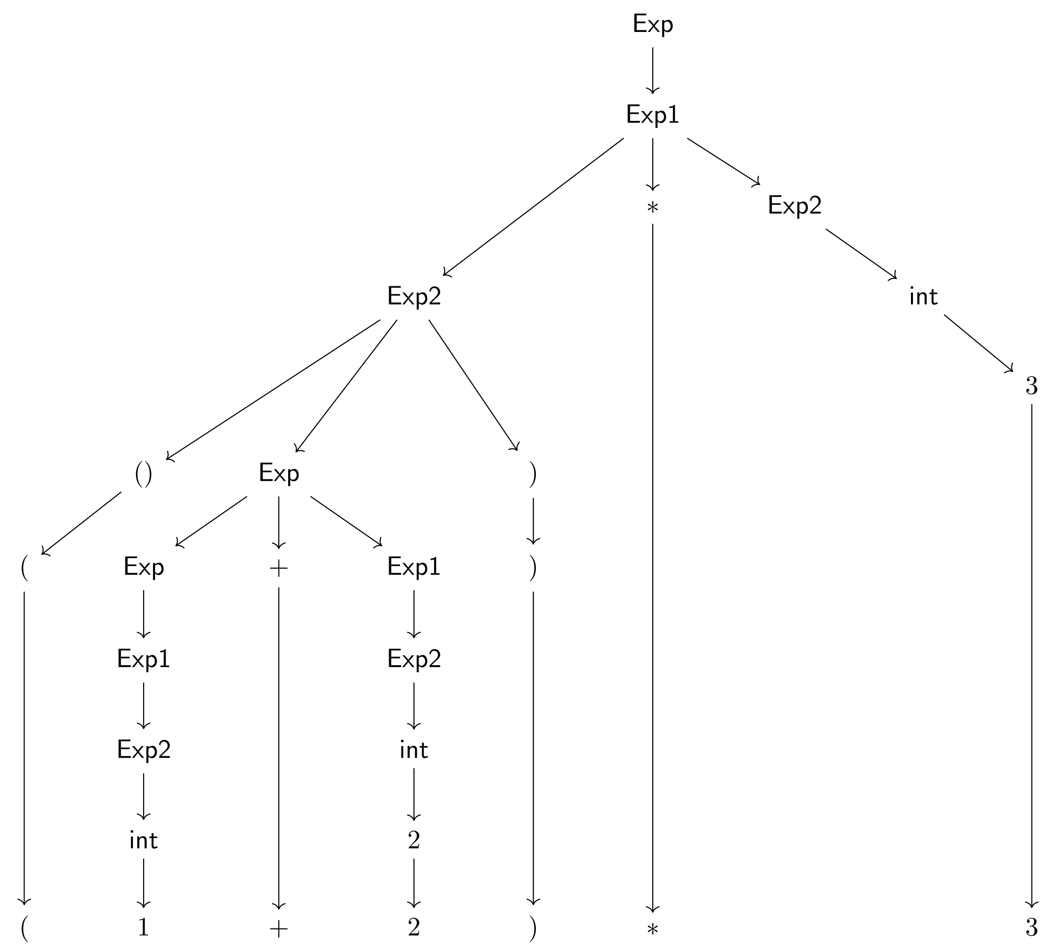
Tree #2


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

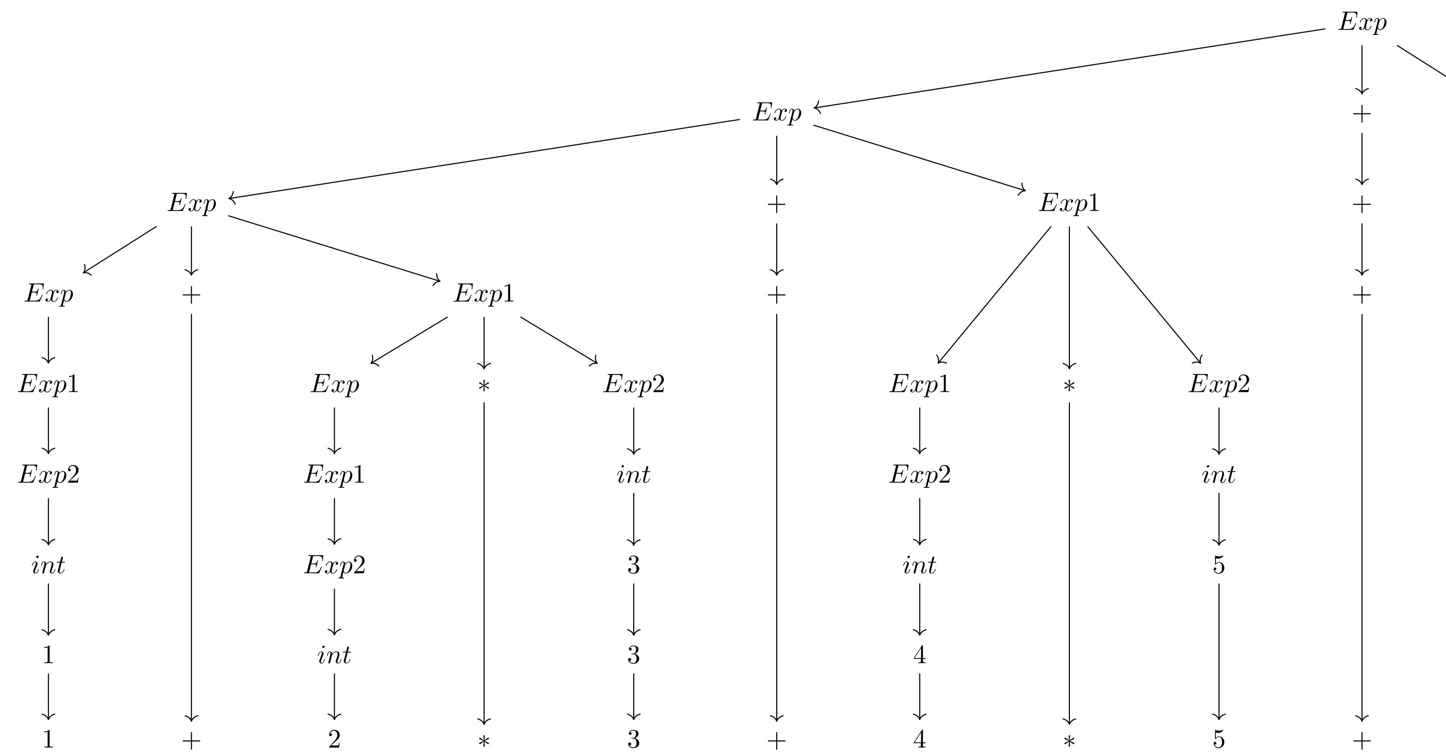
Tree #3


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

Tree #4


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

Tree #5



2.8 Week 8, Tutorial world

HW 8 (NNG Tutorial World: Levels 5–8)

Level 5.

$$\forall a, b, c \in \mathbb{N}, \quad a + (b + 0) + (c + 0) = a + b + c.$$

Lean (tactic script).

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

Natural-language proof. Use the right identity of addition on naturals: for any $x \in \mathbb{N}$, $x+0 = x$. Applying it to b gives $b+0 = b$; applying it to c gives $c+0 = c$. Substitute to get $a+(b+0)+(c+0) = a+b+c$. $\square$

Level 6. (Goal) $a + (b + 0) + (c + 0) = a + b + c.$

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

Level 7. succ_eq_add_one

Theorem 2.1. *For all $n \in \mathbb{N}$, $\operatorname{succ}(n) = n + 1$.*

Lean.

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

Level 8.

$$(2 : \mathbb{N}) + 2 = 4.$$

Lean.

```
example : (2 : Nat) + 2 = 4 := by
  nth_rewrite 2 [two_eq_succ_one]
  rw [add_succ]
  rw [succ_eq_add_one]
  nth_rewrite 1 [two_eq_succ_one]
  rw [one_eq_succ_zero]
  rw [four_eq_succ_three]
  rw [three_eq_succ_two]
  rw [two_eq_succ_one]
  nth_rewrite 5 [succ_eq_add_one]
  nth_rewrite 5 [succ_eq_add_one]
  nth_rewrite 5 [succ_eq_add_one]
  nth_rewrite 1 [succ_eq_add_one]
  rw [one_eq_succ_zero]
  rfl
```

2.9 Week 9, Addition World

HW 9 (Level 5: $(P \wedge Q) \Rightarrow Q$) [?]

Problem. From $P \wedge Q$ infer Q. In Lean:

```
example (P Q : Prop) (h : P ∧Q) : Q := by
  exact h.right
```

Solution A (without induction): $\wedge$-elimination / projection.

Theorem 2.2. *If $h : P \wedge Q$, then Q.*

Pen-and-paper proof. By conjunction elimination on the right ($\wedge$-elim$_2$), from $P \wedge Q$ we obtain Q. Formally, the second projection $\pi_2 : P \wedge Q \to Q$ applied to h yields Q. □

Lean (one-liner).

```
example (P Q : Prop) (h : P ∧Q) : Q := h.2
```

Solution B (with induction/case analysis on And).

Theorem 2.3. *If $h : P \wedge Q$, then Q.*

Pen-and-paper proof using the inductive/elimination principle of And. The proposition $P \wedge Q$ is an inductive type with constructor $\mathsf{And.intro} : P \to Q \to (P \wedge Q)$. Its eliminator says: to produce a result R from $P \wedge Q$, it suffices to provide a function $f : P \to Q \to R$. Choose $R := Q$ and $f(p, q) := q$. Applying the eliminator to h yields Q. Concretely, "case-analyze" h into $p : P$ and $q : Q$, then return q. □

Lean (case analysis).

```
example (P Q : Prop) (h : P ∧Q) : Q := by
  cases h with
  | intro hp hq => exact hq
-- Equivalent:
-- example (P Q : Prop) (h : P ∧Q) : Q := And.rec (fun _ q => q) h
```

3 Essay

4 Evidence of Participation

5 Conclusion

References

[BLA] Author, Title, Publisher, Year.

[kurz-termination] Alexander Kurz, *Termination*, HackMD, https://hackmd.io/42apdFwaSp6ooCQ1woFkEg.

[kurz-hw4] Alexander Kurz, *HW 4, PL 2025, Termination*, HackMD, https://hackmd.io/@alexhkurz/HkdH9cPjxx.

[kurz-hw5] Alexander Kurz, *Lambda Calculus, Syntax and Semantics*, HackMD, 2025, https://hackmd.io/@alexhkurz/rJR2H3YCR#Workout.