CPSC-354 Report

Nikolai Semerdjiev
Chapman University

November 18, 2025

Abstract

Contents

1 Introduction **2**

2 Week by Week **2**
2.1 Week 1 . . . 2
2.1.1 Notes and Exploration . . . 2
2.1.2 Homework . . . 2
2.1.3 Questions . . . 3
2.2 Week 2 . . . 3
2.2.1 Homework . . . 3
2.2.2 Questions . . . 6
2.3 Week 3 . . . 6
2.3.1 Homework . . . 6
2.3.2 Questions . . . 7
2.4 Week 4 . . . 7
2.4.1 Homework . . . 7
2.4.2 Questions . . . 9
2.5 Week 5 . . . 9
2.5.1 Homework . . . 9
2.5.2 Questions . . . 10
2.6 Week 6 . . . 10
2.6.1 Homework . . . 10
2.6.2 Questions . . . 11
2.7 Week 7 . . . 11
2.7.1 Homework . . . 11
2.7.2 Questions . . . 16
2.8 Week 8 . . . 16
2.8.1 Homework . . . 16
2.8.2 Questions . . . 18
2.9 Week 9 . . . 18
2.9.1 Homework . . . 18
2.9.2 Questions . . . 19
2.10 Week 10 . . . 19
2.10.1 Homework . . . 19
2.10.2 Questions . . . 20
2.11 Week 11 . . . 20
2.11.1 Homework . . . 20

2.11.2 Questions . . . 21
2.12 Week 12 . . . 21
2.12.1 Homework . . . 21
2.12.2 Questions . . . 22
2.13 Week 13 . . . 23
2.13.1 Homework . . . 23
2.13.2 Questions . . . 26

3 Essay **26**

4 Evidence of Participation **26**

5 Conclusion **26**

1 Introduction

2 Week by Week

2.1 Week 1

2.1.1 Notes and Exploration

In Week 1, we began with the MIU (or MU) puzzle from Hofstadter's *Gödel, Escher, Bach*. This puzzle introduces the idea of formal systems and rules of inference. It is a good starting point to think about what it means to derive a string from an axiom under a fixed set of rules.

The MU Puzzle

RULES:

1. (Append-U) If a string ends with I, you may append U:
   $x\texttt{I} \to x\texttt{IU}$.
2. (Double) From Mx you may produce Mxx:
   M$x \to$ Mxx.
3. (III→U) Replace any occurrence of III with U:
   $x\texttt{III}y \to x\texttt{U}y$.
4. (Delete UU) Delete any occurrence of UU:
   $x\texttt{UU}y \to xy$.

2.1.2 Homework

Problem: Can you derive MU from MI?

Solution: No, it is impossible to derive MU from MI. At first, playing around with the rules, I noticed that the goal was to create the correct number of I's so that they could be converted to a single U. This means we would need $N_I \bmod 3 = 0$, where N_I counts the I's.

Now, consider how each rule affects N_I:

- **(Append-U)** Appends a U, leaves N_I unchanged.
- **(Double)** M$x \to$ Mxx doubles N_I; in modular arithmetic, $N_I \mapsto 2N_I \pmod 3$.
- **(III→U)** Removes three I's, leaving $N_I \bmod 3$ unchanged.
- **(Delete UU)** Only touches U's, leaves N_I unchanged.

Starting from `MI`, we have $N_I = 1 \equiv 1 \pmod 3$. Doubling cycles between 1 and 2 modulo 3, never producing 0. Thus, it is impossible to reach $N_I \equiv 0 \pmod 3$.

Since `MU` has $N_I = 0$, it cannot be derived from `MI`.

2.1.3 Questions

What is the reasoning behind being able to convert `MIII` into `MU` (using the rule `III` $\rightarrow$ `U`) but not being able to go the other way (from `MU` to `MIII`)?

2.2 Week 2

2.2.1 Homework

Problem: Consider the following ARSs. Draw a picture for each one. Are the ARSs terminating? Are they confluent? Do they have unique normal forms?

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

Solution: For each ARS, I drew a graph (see figures) and analyzed:

- **Termination:** whether there are infinite chains. - **Confluence:** whether every divergence can rejoin. - **Unique normal forms:** whether each element has a unique NF.

I will include one diagram for each ARS along with a short explanation of my analysis.

ARS 1: $A = \{\},\ R = \{\}$ There is no infinite chain, no diverging paths exist as there is no path at all, and nothing exists to violate uniqueness. Therefore ✓ terminating, ✓ confluent, and ✓ has unique normal form.

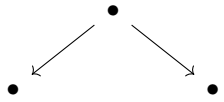
ARS 2: $A = \{a\},\ R = \{\}$

No rewrite steps: no infinite chain, no diverging paths exist or no path at all so it is vacuously satisfied and a is the only reachable normal form from a. Therefore ✓ terminating, ✓ confluent, and ✓ has unique normal form.

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

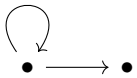
There is an infinite chain, $a \rightarrow^* a$. Since $a \rightarrow^* y$ and $a \rightarrow^* z$, therefore $y = z = a$, joining at a, and there are no normal forms as there is only one term, a, which has infinite outgoing rewrite steps. Therefore $\times$ terminating, ✓ confluent, and $\times$ unique normal form.

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



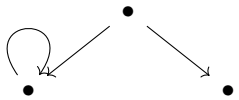
There is no infinite chain, two diverging paths that do not get joined, and two different normal forms from a. Therefore ✓ terminating, × confluent, and × unique normal form.

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



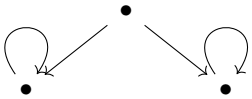
Even with one infinite chain it does not terminate, $a \to a$ and $a \to b$ and they join at b, b is the only normal form since a has an infinite chain therefore × terminating, ✓ confluent, has a ✓ unique normal form.

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



Infinite chain at b (since $b \to b \to b \cdots$). From a the system diverges to b and c with no connecting path to join them. The only normal form is c, but not every element reduces to a unique normal form, so the system is × terminating, × confluent, and × has unique normal form.

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



There are two infinite chains ($b \to b \to \cdots$ and $c \to c \to \cdots$). From a the system diverges to b and c, but since b only reaches b and c only reaches c, the peak at a does not join. Every element has outgoing rewrite steps, so there are no normal forms. Therefore × terminating, × confluent, and × has unique normal form.

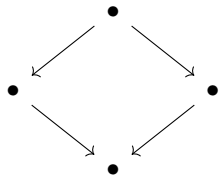
All Eight Combinations

The homework also asked to find examples of ARSs for each of the eight possible combinations of confluence, termination, and unique normal forms. The table below summarizes the results.

Confluent	**Terminating**	**Unique Normal Forms**	**Example** (A, R)
True	True	True	$A = \{a, b, c, d\},\ R = \{(a, b), (a, c), (b, d), (c, d)\}$
True	True	False	N/A
True	False	True	$A = \{a, b, c, d\},\ R = \{(a, a), (a, b), (a, c), (b, d), (c, d)\}$
True	False	False	$A = \{a, b, c, d\},\ R = \{(a, b), (a, c), (b, d), (c, d), (d, d)\}$
False	True	True	N/A
False	True	False	$A = \{a, b, c, d\},\ R = \{(a, b), (a, c), (b, d)\}$
False	False	True	$A = \{a, b, c\},\ R = \{(a, b), (a, c), (b, d), (c, d), (d, d)\}$
False	False	False	$A = \{a, b, c\},\ R = \{(a, a), (a, b), (a, c)\}$

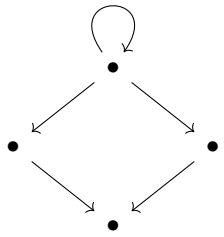
Example Graphs

Graph 1

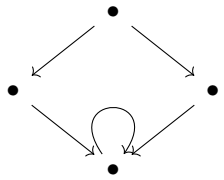


Graph 2 This comination is impossible as termination gives a normal form for each element AND confluence guarantees any two reduction paths from the same element to join therefore all reductions end at some normal form.

Graph 3

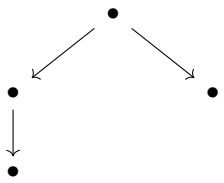


Graph 4

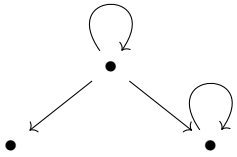


Graph 5 This case is impossible. Every system that is terminating and where every element has a unique normal form must also be confluent. Indeed, if $a \to^* y$ and $a \to^* z$, then both y and z reduce to some normal forms. Since the normal form is unique, y and z must reduce to the same normal form, which means the system is confluent.

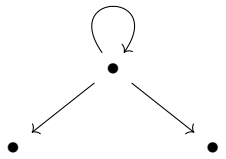
Graph 6



Graph 7



Graph 8



2.2.2 Questions

How can thinking about ARSs help us better understand the way programming languages define and control the process of evaluating programs?

2.3 Week 3

2.3.1 Homework

Exercise 5: Consider the rewrite rules

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

Reduce some example strings such as `abba` and `bababa`. **Answer:**

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

$$\texttt{bababa} \to \texttt{bbaaba} \to \texttt{bbba} \to \texttt{bba} \to \texttt{ba} \to \texttt{a}.$$

No rule applies to a single `a`*; this is the farthest you can go.*

Why is the ARS not terminating? **Answer:** There is always a way to terminate if you use the deletion rules (3) and (4), but using only the swap rules (1) and (2) yields an infinite loop (e.g., $\texttt{ab} \leftrightarrow \texttt{ba} \leftrightarrow \texttt{ab} \leftrightarrow \cdots$), so the ARS is *not terminating* overall.

Find two strings that are not equivalent. How many non-equivalent strings can you find? **Answer:** Two strings not equivalent: $\texttt{baba} \not\leftrightarrow^* \texttt{ababa}$. There are infinitely many non-equivalent pairs: strings split by the parity of the number of a's (even vs. odd), so you can keep making pairs with different a-parity.

How many equivalence classes does $\leftrightarrow^*$ have? Can you describe them in a nice way? What are the normal forms? **Answer:** There are 2 equivalence classes under $\leftrightarrow^*$: (1) strings with an *even* number of a's, which reduce to the normal form ε; (2) strings with an *odd* number of a's, which reduce to the normal form a. Set of normal forms: $\{\varepsilon, a\}$.

Can you modify the ARS so that it becomes terminating without changing its equivalence classes? **Answer:** Yes. Drop $ab \to ba$ and keep $ba \to ab$, together with $aa \to \varepsilon$ and $b \to \varepsilon$. This orientation is terminating while preserving the same equivalence classes and normal forms (still $\{\varepsilon, a\}$).

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. **Answer:** Given any string $x \in \{a, b\}^*$, does x reduce to ε or to a? Equivalently: is the number of a's in x even or odd? This gives a semantics mapping each string to ε (even) or a (odd).

Exercise 5b: Consider the rewrite rules

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

Reduce some example strings such as `abba` and `bababa`. **Answer:**

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

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

No rule applies to a single `a`*; this is the farthest you can go.*

Why is the ARS not terminating? **Answer:** This example still has an infinite chain if you don't use rules (3) or (4): the swaps $ab \leftrightarrow ba$ can loop forever.

Find two strings that are not equivalent. How many non-equivalent strings can you find? **Answer:** Two non-equivalent strings: $\texttt{bbb} \not\leftrightarrow^* \texttt{babab}$. There are infinitely many non-equivalent pairs.

How many equivalence classes does $\leftrightarrow^*$ have? Can you describe them in a nice way? What are the normal forms? **Answer:** There are 2 equivalence classes for $\leftrightarrow^*$. If a string has *no* a's, its normal form is ε. If a string has *at least one* a, its normal form is a. Set of normal forms: $\{\varepsilon, a\}$.

Can you modify the ARS so that it becomes terminating without changing its equivalence classes? **Answer:** Drop $ab \to ba$ and keep $ba \to ab$, along with $aa \to a$ and $b \to \varepsilon$. This orientation terminates while preserving the same equivalence classes and normal forms.

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. **Answer:** Given any string $x \in \{a, b\}^*$, does x contain at least one a? Equivalently: does x reduce to a (some a present) or to ε (no a's)?

2.3.2 Questions

How does changing the rewrite rule from $aa \to \varepsilon$ to $aa \to a$ resemble differences in evaluation strategies when creating a programming language?

2.4 Week 4

2.4.1 Homework

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

The algorithm always terminates provided that.

1. The inputs satisfy $a \geq 0$ and $b \geq 0$.
2. If $b = 0$ initially, the loop terminates immediately.
3. For $b > 0$, the remainder operation is defined by the division algorithm:

$$a = qb + r, \quad 0 \leq r < b,$$

   where $r = a \bmod b$.

Define the measure function

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

Consider one loop iteration:

$$(a, b) \mapsto (b,\ a \bmod b).$$

By the division algorithm,

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

Therefore,

$$\phi(a, b) = b > a \bmod b = \phi(b, a \bmod b).$$

Hence ϕ strictly decreases with each iteration.

Since $\phi(a, b)$ is a nonnegative integer, it cannot decrease infinitely. After finitely many steps, we must reach $b = 0$, at which point the loop terminates.

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

The function

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

is a measure function for `merge_sort` (with integer indices and `mid` $= \lfloor(\text{left} + \text{right})/2\rfloor$).

Assume $\text{left}, \text{right} \in \mathbb{Z}$ and the procedure is only called with $\text{left} \leq \text{right}$. The branching factor is finite (at most two recursive calls).

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

- *Base case.* If $\text{left} \geq \text{right}$ then $n \leq 1$ and the function returns without making recursive calls. No decrease condition needs to be checked.
- *Recursive case.* Suppose $\text{left} < \text{right}$, hence $n \geq 2$. Set $\texttt{mid} = \left\lfloor \dfrac{\text{left} + \text{right}}{2} \right\rfloor$. Then $\text{left} \leq \texttt{mid} < \text{right}$, so both subproblems are well-formed:

$$(\text{left}, \texttt{mid}) \quad \text{and} \quad (\texttt{mid} + 1, \text{right}).$$

  Their measures are

$$\phi(\text{left}, \texttt{mid}) = \texttt{mid} - \text{left} + 1 \quad \text{and} \quad \phi(\texttt{mid} + 1, \text{right}) = \text{right} - \texttt{mid}.$$

  Using $\texttt{mid} = \left\lfloor \dfrac{\text{left} + \text{right}}{2} \right\rfloor$ and $n = \text{right} - \text{left} + 1$, we get the bounds

$$\phi(\text{left}, \texttt{mid}) \ \leq \ \left\lceil \frac{n}{2} \right\rceil \quad \text{and} \quad \phi(\texttt{mid} + 1, \text{right}) \ \leq \ \left\lfloor \frac{n}{2} \right\rfloor.$$

Since $n \geq 2$, both $\left\lceil \frac{n}{2} \right\rceil < n$ and $\left\lfloor \frac{n}{2} \right\rfloor < n$ hold. Therefore

$$\phi(\text{left}, \texttt{mid}) \;<\; \phi(\text{left}, \text{right}) \qquad \text{and} \qquad \phi(\texttt{mid}+1, \text{right}) \;<\; \phi(\text{left}, \text{right}).$$

Hence the measure strictly decreases along every recursive edge.

Because ϕ maps states to $\mathbb{N}$ and strictly decreases with each recursive call, no infinite descent is possible. Together with finite branching, this implies termination.

2.4.2 Questions

How do different evaluation strategies in programming languages (such as call-by-value vs. call-by-name) influence whether a program is guaranteed to terminate, and in what ways can techniques like measure functions help us reason about termination across these strategies?

2.5 Week 5

2.5.1 Homework

$$\textbf{Goal:} \quad (\lambda f.\, \lambda x.\, f\,(f\,x))\;(\lambda f.\, \lambda x.\, f\,(f\,(f\,x))).$$

–renaming. To avoid variable capture, rename bound variables so the two arguments use distinct names:

$$A \;\equiv\; \lambda f.\, \lambda x.\, f(f\,x) \quad \text{and} \quad B \;\equiv\; \lambda g.\, \lambda y.\, g(g(g\,y)).$$

The term is $A\,B$.

–reduction (outer application).

$$A\,B \;\to_\beta\; \lambda x.\, B\big(B\,x\big) \quad (\text{substitute } f := B \text{ in } f(f\,x)).$$

–reduction of $B\,x$.

$$B\,x = (\lambda g.\, \lambda y.\, g(g(g\,y)))\,x \;\to_\beta\; \lambda y.\, x\big(x(x\,y)\big).$$

Name this $B_x \equiv \lambda y.\, x^3(y)$, where x^k denotes k-fold self-composition.

–reduction of $B\,(B_x)$.

$$B\,(B_x) = (\lambda g.\, \lambda y.\, g(g(g\,y)))\,(\lambda y.\, x^3(y)) \;\to_\beta\; \lambda y.\, B_x\big(B_x\,(B_x\,y)\big).$$

Since $B_x\,y = x^3(y)$, we compute

$$B_x\big(B_x\,y\big) = B_x\big(x^3(y)\big) = x^3\big(x^3(y)\big) = x^6(y),$$

and then

$$B_x\big(B_x\,(B_x\,y)\big) = B_x\big(x^6(y)\big) = x^3\big(x^6(y)\big) = x^9(y).$$

Hence

$$B\,(B_x) \;\to\; \lambda y.\, x^9(y).$$

Assemble the result.

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

Renaming $x \mapsto f$ and $y \mapsto a$ gives the canonical Church-numeral form

$$\boxed{\lambda f.\, \lambda a.\, f^9(a)}.$$

Thus

$$(\lambda f.\,\lambda x.\,f(f\,x))\;(\lambda f.\,\lambda x.\,f(f(f\,x)))\;=\;\lambda f.\,\lambda x.\,f^{9}(x).$$

Remark (mathematical interpretation). In Church encodings, natural numbers are functions that iterate an endofunction:

$$n\;\equiv\;\lambda f.\,\lambda x.\,f^{\,n}(x).$$

Here, $\lambda f.\lambda x.\,f(f\,x)$ is the numeral 2 ("apply f twice") and $\lambda f.\lambda x.\,f(f(f\,x))$ is the numeral 3 ("apply f thrice"). Applying one numeral to another composes iterations and realizes *exponentiation by iteration*:

$$(2\;3)\;=\;\lambda x.\,3^{\,2}(x)\;=\;\lambda f.\lambda x.\,f^{9}(x),$$

so the result corresponds to the number $9 = 3^2$. Conceptually, this is *function iteration*: "do thrice, then do thrice again" = "do nine times." Before computers, Church's lambda calculus provided a purely symbolic foundation where numbers are actions (iterations) on functions; your reduction is exactly the arithmetic law that composing "apply-thrice" with itself yields "apply-nine-times."

2.5.2 Questions

Since Church numerals let us represent arithmetic entirely through function application, what does that suggest about the nature of mathematics—are operations like addition, multiplication, and exponentiation really 'just' repeated patterns of substitution and iteration?

2.6 Week 6

2.6.1 Homework

Evaluating `fact` 3 with `fix`, `let`, and `let rec`

Rules

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

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

Target term

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

Derivation (each step labeled)

$$\begin{aligned}
&\texttt{let rec fact}=\lambda n.\;\texttt{if}\;n=0\;\texttt{then}\;1\;\texttt{else}\;n*\texttt{fact}\,(n-1)\;\texttt{in fact}\;3\\
&\xrightarrow{\text{def of let rec}}\;\texttt{let fact}=\texttt{fix}\,(\lambda\texttt{fact}.\;\lambda n.\;\texttt{if}\;n=0\;\texttt{then}\;1\;\texttt{else}\;n*\texttt{fact}\,(n-1))\;\texttt{in fact}\;3\\
&\xrightarrow{\text{def of let}}\;(\lambda\texttt{fact}.\;\texttt{fact}\;3)\;\texttt{fix}\,(\lambda\texttt{fact}.\;\lambda n.\;\texttt{if}\;n=0\;\texttt{then}\;1\;\texttt{else}\;n*\texttt{fact}\,(n-1))\\
&\xrightarrow{\beta}\;\big(\texttt{fix}\,(\lambda\texttt{fact}.\;\lambda n.\;\texttt{if}\;n=0\;\texttt{then}\;1\;\texttt{else}\;n*\texttt{fact}\,(n-1))\big)\;3\\
&=\;(\texttt{fix}\;F)\;3\\
&\xrightarrow{\text{def of fix}}\;\big(F\,(\texttt{fix}\;F)\big)\;3\\
&\xrightarrow{\beta}\;\big(\lambda n.\;\texttt{if}\;n=0\;\texttt{then}\;1\;\texttt{else}\;n*(\texttt{fix}\;F)\,(n-1)\big)\;3\\
&\xrightarrow{\beta}\;\texttt{if}\;3=0\;\texttt{then}\;1\;\texttt{else}\;3*(\texttt{fix}\;F)\,2\\
&\xrightarrow{\text{def of if}}\;3*(\texttt{fix}\;F)\,2
\end{aligned}$$

Expand $(\texttt{fix}\ F)\,2$:

$$(\texttt{fix}\ F)\,2 \xrightarrow{\text{def of fix}} \big(F\,(\texttt{fix}\ F)\big)\,2 \xrightarrow{\beta} \big(\lambda n.\ \texttt{if}\ n = 0\ \texttt{then}\ 1\ \texttt{else}\ n * (\texttt{fix}\ F)\,(n-1)\big)\ 2$$
$$\xrightarrow{\beta} \texttt{if}\ 2 = 0\ \texttt{then}\ 1\ \texttt{else}\ 2 * (\texttt{fix}\ F)\,1 \xrightarrow{\text{def of if}} 2 * (\texttt{fix}\ F)\,1.$$

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

$$(\texttt{fix}\ F)\,1 \xrightarrow{\text{def of fix}} \big(F\,(\texttt{fix}\ F)\big)\,1 \xrightarrow{\beta} \big(\lambda n.\ \texttt{if}\ n = 0\ \texttt{then}\ 1\ \texttt{else}\ n * (\texttt{fix}\ F)\,(n-1)\big)\ 1$$
$$\xrightarrow{\beta} \texttt{if}\ 1 = 0\ \texttt{then}\ 1\ \texttt{else}\ 1 * (\texttt{fix}\ F)\,0 \xrightarrow{\text{def of if}} 1 * (\texttt{fix}\ F)\,0.$$

Base case $(\texttt{fix}\ F)\,0$:

$$(\texttt{fix}\ F)\,0 \xrightarrow{\text{def of fix}} \big(F\,(\texttt{fix}\ F)\big)\,0 \xrightarrow{\beta} \big(\lambda n.\ \texttt{if}\ n = 0\ \texttt{then}\ 1\ \texttt{else}\ n * (\texttt{fix}\ F)\,(n-1)\big)\ 0$$
$$\xrightarrow{\beta} \texttt{if}\ 0 = 0\ \texttt{then}\ 1\ \texttt{else}\ 0 * (\texttt{fix}\ F)\,(-1) \xrightarrow{\text{def of if}} 1.$$

Back-substitute the arithmetic:

$$(\texttt{fix}\ F)\,0 = 1, \quad (\texttt{fix}\ F)\,1 = 1, \quad (\texttt{fix}\ F)\,2 = 2, \quad (\texttt{fix}\ F)\,3 = \boxed{6}.$$

2.6.2 Questions

If the `fix` operator allows a function to "refer to itself" without any explicit naming or looping construct, what does that reveal about recursion as a *mathematical* idea rather than merely a *programming* one?

Does this suggest that self-reference—and therefore recursion—emerges purely from the rules of substitution and function application, rather than from syntactic features like `while` or `for` loops?

2.7 Week 7

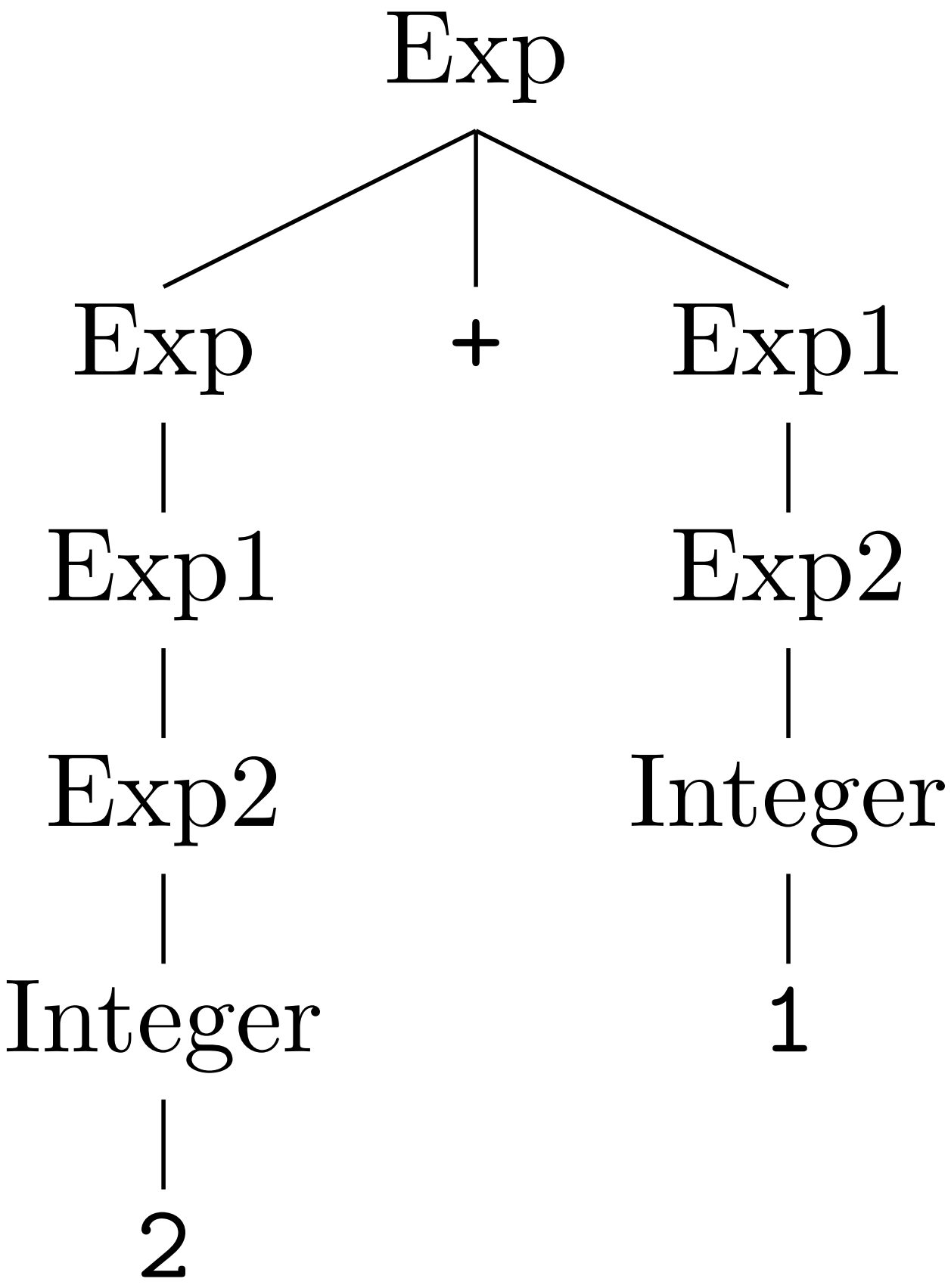
2.7.1 Homework

Parse Trees for the Given Grammar

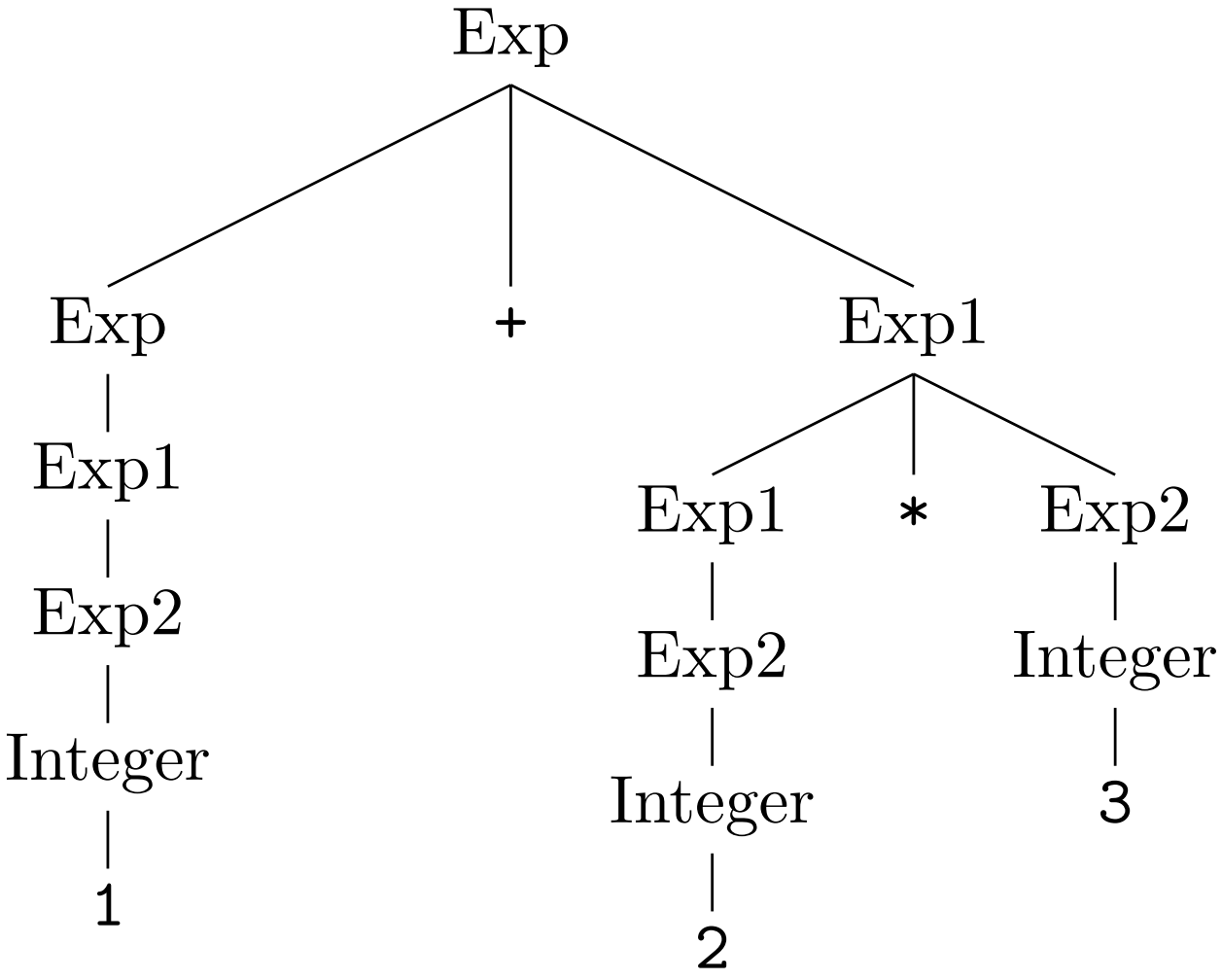
Grammar

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

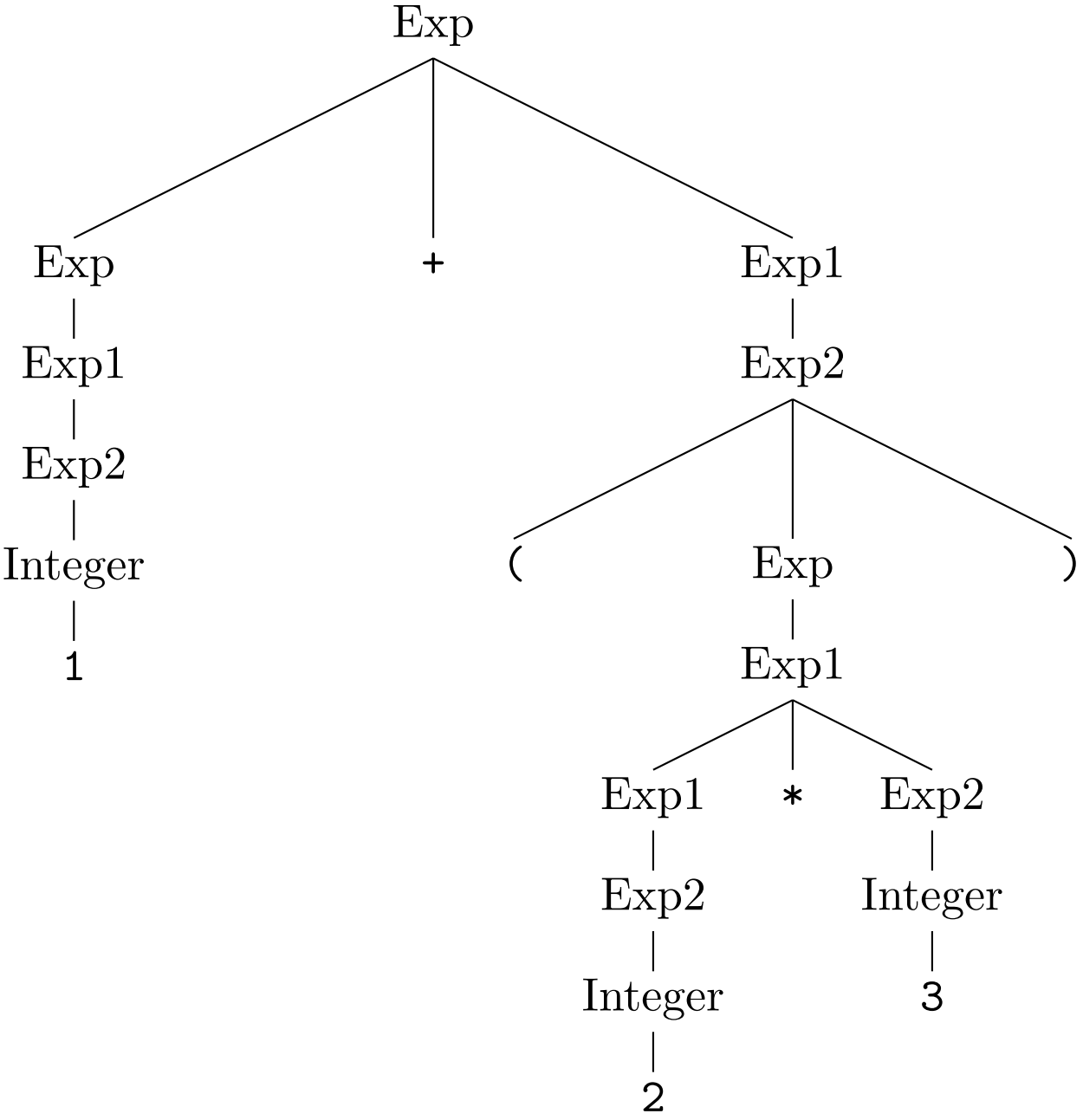
1) 2+1



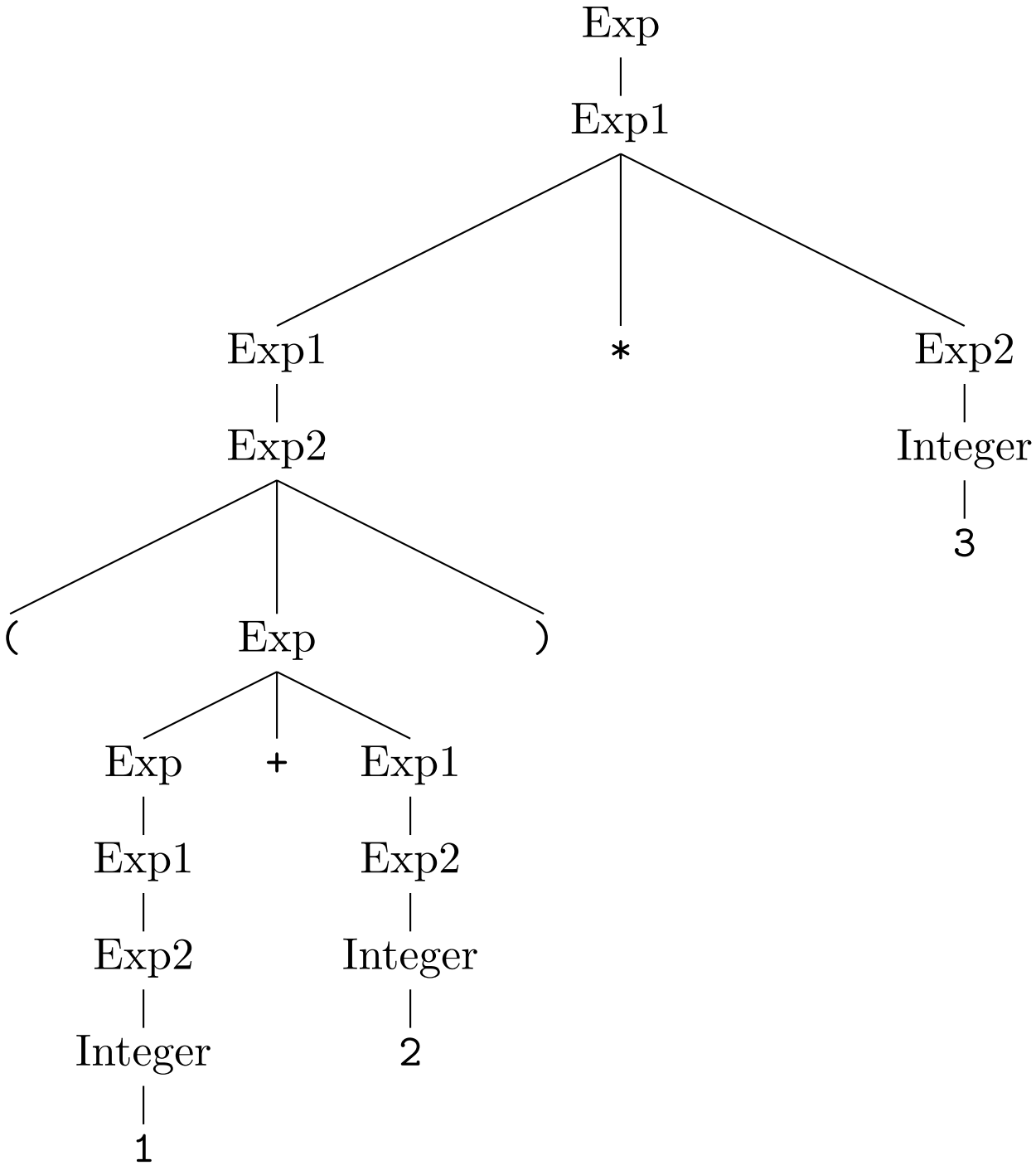
2) `1+2*3`


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

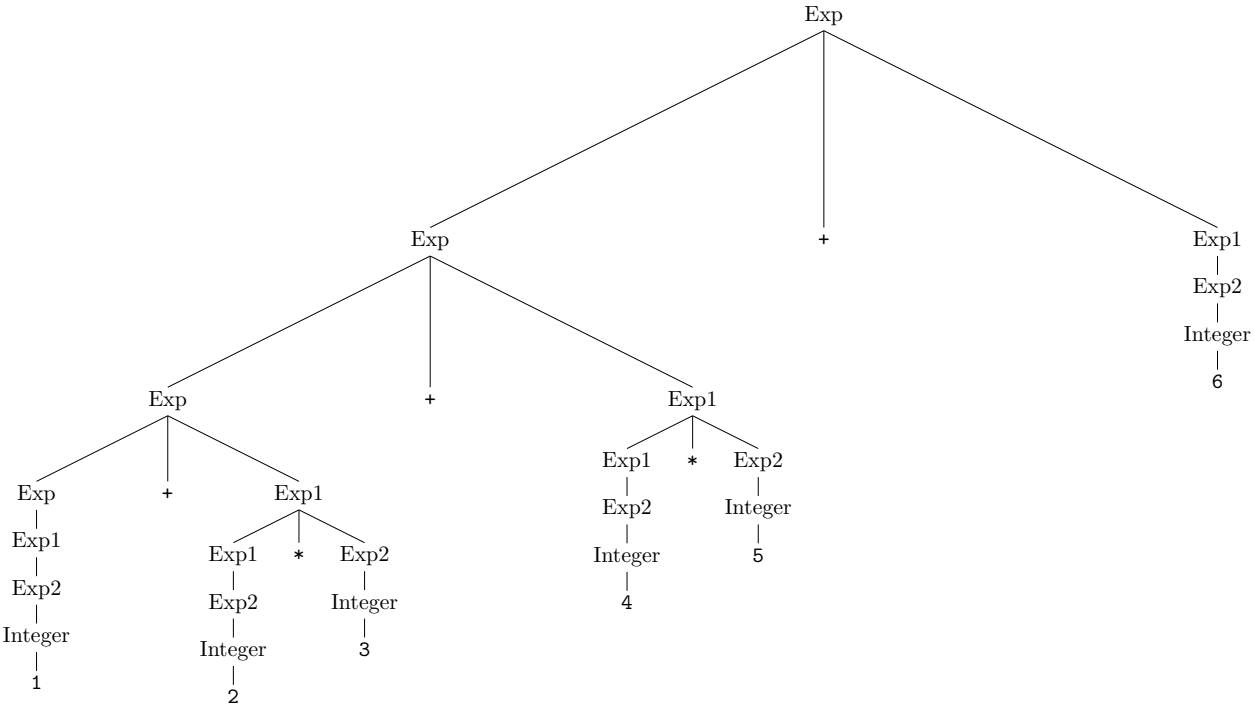
3) `1+(2*3)`


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

4) `(1+2)*3`


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

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



2.7.2 Questions

When a grammar allows multiple ways to parse the same expression, such as different groupings of additions and multiplications, how can we modify or design the grammar so that the parse tree always reflects the correct operator precedence and associativity?

2.8 Week 8

2.8.1 Homework

Level 5/8: *Adding zero*

Question. Prove, for natural numbers a, b, c,

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

Using the identity $x + 0 = x$ (often named `add_zero`) twice:

$$\begin{aligned} a + (b + 0) + (c + 0) &= a + b + (c + 0) && (\text{by } b + 0 = b) \\ &= a + b + c && (\text{by } c + 0 = c). \end{aligned}$$

Lean script.

```
-- Objects: a b c :
rw [add_zero]    -- a + (b + 0) + (c + 0) ==> a + b + (c + 0)
rw [add_zero]    -- a + b + (c + 0)     ==> a + b + c
rfl              -- both sides now identical
```

Level 6/8: *Precision rewriting*

Question. Prove, for natural numbers a, b, c,

$$a + (b + 0) + (c + 0) \;=\; a + b + c,$$

but emphasize *targeted* rewriting (i.e., rewriting only the needed subterm each step).

Same identities as above, but apply them precisely to the subterm containing 0:

$$\begin{aligned} a + (b + 0) + (c + 0) &= a + b + (c + 0) && \text{rewrite only the inner } (b + 0) \\ &= a + b + c && \text{now rewrite the remaining } (c + 0). \end{aligned}$$

Lean script (one precise rewrite at a time).

```
-- Objects: a b c :
rw [add_zero]    -- rewrite (b + 0) -> b
rw [add_zero]    -- rewrite (c + 0) -> c
rfl
```

(If the goal were more complicated, one could direct a rewrite to a specific occurrence using a location, e.g. ‘rw [add$_z$*ero*]*ath*‘*or*‘*simp*[*add$_z$ero*]‘*withsideconditions*.)

Level 7/8: *add_succ*

Theorem (stated).

$$\forall n : \mathbb{N}, \quad n.\mathrm{succ} \;=\; n + 1.$$

Useful lemmas.

$$\begin{aligned} 1 &= \mathrm{succ}(0) && \text{(named } \texttt{one\_eq\_succ\_zero}\text{)}, \\ n + \mathrm{succ}(k) &= \mathrm{succ}(n + k) && \text{(named } \texttt{add\_succ}\text{)}, \\ n + 0 &= n && \text{(named } \texttt{add\_zero}\text{)}. \end{aligned}$$

$$\begin{aligned} n.\mathrm{succ} &= n + 0.\mathrm{succ} && (\text{since } \mathrm{succ}(x) = x + 1, \text{ we move via } 1 = \mathrm{succ}(0)) \\ &= \mathrm{succ}(n + 0) && \text{by } \texttt{add\_succ} \\ &= \mathrm{succ}(n) && \text{by } \texttt{add\_zero} \\ &= n + 1 && \text{unwinding } 1 = \mathrm{succ}(0). \end{aligned}$$

Concretely (mirroring your screenshot), we first rewrite the 1 as succ(0), then use `add_succ`, then `add_zero`, and finish with reflexivity.

Lean script (exact sequence from the level).

```
-- Goal: n.succ = n + 1
rw [one_eq_succ_zero] -- turn 1 into succ 0 on the RHS
rw [add_succ]         -- n + succ 0 -> (n + 0).succ
rw [add_zero]         -- (n + 0).succ -> n.succ
rfl                   -- both sides are n.succ
```

Notes.

- Levels 5 and 6 highlight the same identity $x + 0 = x$ but train you to control *where* a rewrite happens.
- Level 7 chains small lemmas to reach a familiar arithmetic fact: $n + 1 = \mathrm{succ}(n)$.

Question 8 — Natural–Language Proof (Level 8: $2 + 2 = 4$)

Goal. Prove that $2 + 2 = 4$ in Peano arithmetic.

Background/Definitions. Let 0 be the base natural number and let succ(n) (sometimes written $n.succ$) denote the successor of n. Define the numerals

$$1 \;=\; \text{succ}(0), \qquad 2 \;=\; \text{succ}(1), \qquad 3 \;=\; \text{succ}(2), \qquad 4 \;=\; \text{succ}(3).$$

Addition is defined by the standard recursion rules:

$$\text{(add\_zero)} \quad n + 0 = n, \qquad \text{(add\_succ)} \quad n + \text{succ}(k) = \text{succ}(n + k).$$

Proof. We compute $2 + 2$ using only the recursion rules for $+$ and the numeral definitions.

$$\begin{aligned}
2 + 2 &= \text{succ}(1) + \text{succ}(1) && \text{(by the definition of 2)} \\
&= \text{succ}\big(\,\text{succ}(1) + 1\,\big) && \text{(by } \textit{add_succ} \text{ with } n = \text{succ}(1),\ k = 1) \\
&= \text{succ}\big(\,\text{succ}(1) + \text{succ}(0)\,\big) && \text{(since } 1 = \text{succ}(0)) \\
&= \text{succ}\Big(\,\text{succ}\big(\,\text{succ}(1) + 0\,\big)\Big) && \text{(by } \textit{add_succ} \text{ with } n = \text{succ}(1),\ k = 0) \\
&= \text{succ}\Big(\,\text{succ}\big(\,\text{succ}(1)\,\big)\Big) && \text{(by } \textit{add_zero}\text{: succ}(1) + 0 = \text{succ}(1)) \\
&= \text{succ}\big(\,\text{succ}(\text{succ}(1))\,\big) \\
&= \text{succ}\big(\,\text{succ}(\text{succ}(\text{succ}(0)))\,\big) && \text{(since } 1 = \text{succ}(0)) \\
&= \text{succ}(\text{succ}(\text{succ}(\text{succ}(0))))) \\
&= 4 && \text{(by the definition of 4).}
\end{aligned}$$

Therefore, $2 + 2 = 4$. □

(Optional) Lean step correspondence. The proof above mirrors the scripted steps:

```
rw [four_eq_succ_three]  rw [three_eq_succ_two]  rw [two_eq_succ_one]  rw [one_eq_succ_zero]  rw [add_suc
```

2.8.2 Questions

In Peano arithmetic, every arithmetic fact—such as $2+2 = 4$—is proven step by step from simple definitions like 0 and succ(n). What does this tell us about how much of mathematics can be built from just a few basic rules, and why might this level of formality be both powerful and limiting when compared to the way we usually do arithmetic?

2.9 Week 9

2.9.1 Homework

Here's a compile-ready LaTeX block you can drop into your Week 9 report. It contains **two solutions** for Addition World – **Level 5: $(\text{add}_r ight_c omm) * * : one by * * induction * * (spelled out in English math) and one * * without induction * * (a short proof using the lemma ‘add_r ight_c omm‘). I' ve also included the Lean scripts that match each proof.$

“‘latex

Addition World — Level 5: add_right_comm

Theorem (goal). For all natural numbers a, b, c,

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

Solution A: Inductive proof on b (no tactics required to understand)

We use the Peano rules for addition:

$$\text{(add\_zero) } n + 0 = n, \qquad \text{(add\_succ) } n + \operatorname{succ}(k) = \operatorname{succ}(n + k),$$

together with associativity $(x + y) + z = x + (y + z)$ and commutativity $x + y = y + x$.

Claim. For all a, c, $P(b) : \ a + b + c = a + c + b$ holds by induction on b.

Base case $b = 0$.

$$a + 0 + c \ = \ a + c \ = \ a + c + 0,$$

by two uses of add_zero. Hence $P(0)$ holds.

Inductive step. Assume $P(b)$ holds, i.e.

$$a + b + c \ = \ a + c + b \quad \text{for all } a, c.$$

We must show $P(\operatorname{succ} b)$:

$$a + \operatorname{succ} b + c \ = \ a + c + \operatorname{succ} b.$$

Compute:

$$\begin{aligned} a + \operatorname{succ} b + c &= (a + \operatorname{succ} b) + c \\ &= \operatorname{succ}(a + b) + c && \text{(by add\_succ)} \\ &= \operatorname{succ}\big((a + b) + c\big) && \text{(by succ\_add: } \operatorname{succ} x + c = \operatorname{succ}(x + c)) \\ &= \operatorname{succ}(a + b + c) && \text{(associativity)} \\ &= \operatorname{succ}(a + c + b) && \text{(induction hypothesis)} \\ &= a + c + \operatorname{succ} b && \text{(reverse of add\_succ).} \end{aligned}$$

Thus $P(\operatorname{succ} b)$ holds, and by induction $a + b + c = a + c + b$ for all a, b, c.

Solution B: Direct proof *without* induction (one-line rewrite)

If the library lemma add_right_comm is available, it states exactly

$$x + y + z \ = \ x + z + y.$$

Therefore, taking $x = a$, $y = b$, $z = c$ gives the result immediately:

$$a + b + c \ = \ a + c + b.$$

2.9.2 Questions

Why do we need induction to prove some arithmetic properties like commutativity or associativity, but not others? In other words, what makes a property like $a + b + c = a + c + b$ provable directly from existing lemmas, while others require an inductive step on one of the variables?

2.10 Week 10

2.10.1 Homework

Lean Logic Game — Party Snacks (Levels 6–9, one line each)

Level 6: `and_imp`

One-liner:

```
exact (fun c => fun d => h (And.intro c d))
-- equivalently: exact fun c d => h c,  d
```

Level 7: `and_imp 2`

One-liner:

```
exact (fun hd => h hd.left hd.right)
-- equivalently: exact fun hc,  hd  => h hc hd
```

Level 8: `Distribute`

One-liner:

```
exact fun s => And.intro (h.left s) (h.right s)
-- equivalently: exact fun s => h.left s, h.right s
```

Level 9: `Uncertain Snacks`

One-liner:

```
exact fun r => And.intro (fun _ : S => r) (fun _ :  S => r)
-- equivalently: exact fun r => ((fun _ : S => r), (fun _ :  S => r))
```

2.10.2 Questions

How does writing logical proofs as functions in Lean change the way we think about reasoning with implications? Does it make the structure of logical arguments clearer compared to traditional symbolic proofs on paper?

2.11 Week 11

2.11.1 Homework

Lean Logic Game — Negation (Levels 9–12, one line each)

Level 9 / 12: Implies a Negation

Problem.

$$\text{example } (A\, P : \text{Prop})\, (h : P \to \neg A) :\ \neg(P \wedge A)$$

One-line solution (Lean).

```
exact fun (p, a) => (h p) a
```

Level 10 / 12: Conjunction Implication

Problem.

$$\text{example } (A\, P : \text{Prop})\, (h : \neg(P \wedge A)) :\ P \to \neg A$$

One-line solution (Lean).

```
exact fun p a => h (p, a)
```

Level 11 / 12: `not_not_not`

Problem.

$$\text{example } (A : \text{Prop})\,(h : \neg\neg\neg A) : \ \neg A$$

One-line solution (Lean).

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

Level 12 / 12: `~Intro Boss`

Problem.

$$\text{example } (B\,C : \text{Prop})\,(h : \neg(B + C)) : \ \neg\neg B$$

One-line solution (Lean).

```
exact fun nB => h (fun b => False.elim (nB b))
```

2.11.2 Questions

How does encoding $\neg A$ as $A \to \text{False}$ help us construct one-line proofs using only function application? Does this viewpoint make the structure of negation proofs clearer than traditional derivations?

2.12 Week 12

2.12.1 Homework

Towers of Hanoi (2025) — Activity Write-up (n = 5, from peg 0 to peg 2)

Algorithm (rules)

$$\begin{aligned} \mathsf{hanoi}\ 1\ x\ y &= \mathsf{move}\ x\ y \\ \mathsf{hanoi}\ (n{+}1)\ x\ y &= \mathsf{hanoi}\ n\ x\ (\mathsf{other}\ x\ y)\,;\ \mathsf{move}\ x\ y\,;\ \mathsf{hanoi}\ n\ (\mathsf{other}\ x\ y)\ y \end{aligned}$$

(1) Completed execution trace 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
```

(2) Moves extracted from the trace (in order)

Each line `move x y` is a move of the top disk from peg x to peg y. There are $2^5-1 = 31$ moves, the minimal number for n=5.

$$
\begin{aligned}
&(0\to2),\ (0\to1),\ (2\to1),\ (0\to2),\ (1\to0),\ (1\to2),\ (0\to2),\\
&(0\to1),\ (2\to1),\ (2\to0),\ (1\to0),\ (2\to1),\ (0\to2),\ (0\to1),\ (2\to1),\\
&(0\to2),\\
&(1\to0),\ (1\to2),\ (0\to2),\ (1\to0),\ (2\to1),\ (2\to0),\ (1\to0),\\
&(1\to2),\\
&(0\to2),\ (0\to1),\ (2\to1),\ (0\to2),\ (1\to0),\ (1\to2),\ (0\to2).
\end{aligned}
$$

(3) How to verify against the online Towers of Hanoi

[leftmargin=1.3em]Open the online Towers of Hanoi and set $n = 5$, source peg 0, target peg 2. Play the moves above in order; every move is legal (never places a larger disk on a smaller one). You will finish in exactly 31 moves, which is minimal for $n = 5$. (Why this works.) The algorithm preserves the invariant "the top k disks are always in legal configuration", and the recursive pattern

$$\mathsf{hanoi}\ n\ x\ y = \mathsf{hanoi}\ (n{-}1)\ x\ z;\ \mathsf{move}\ x\ y;\ \mathsf{hanoi}\ (n{-}1)\ z\ y$$

ensures inductively that the largest disk moves exactly once (midpoint move), surrounded by optimal solutions for size $n-1$.

2.12.2 Questions

Why does the recursive algorithm for the Towers of Hanoi always produce the minimum possible number of moves, and how does seeing the full execution trace help you understand why no shorter solution could

exist?

2.13 Week 13

2.13.1 Homework

Item 2: Testing the Interpreter

2.1 Does the implementation conform to the theory?

Running `python interpreter_test.py` executes a battery of tests that check beta-reduction, alpha-conversion, and the chosen evaluation strategy. All the provided tests pass, so the implementation behaves as specified by the mathematical theory of the untyped lambda calculus (in particular: correct handling of bound vs. free variables and correct sequencing of reductions).

2.2 Additional test cases

I added several expressions inspired by the lecture notes to `test.lc`, always predicting the expected normal form before running the interpreter:

1. $a\ b\ c\ d$

   By definition, application in the lambda calculus is left-associative, so

   $$a\ b\ c\ d \equiv (((a\ b)\ c)\ d).$$

   The interpreter prints exactly this left-nested structure.

- (a)

  Parentheses do not change the term, so (a) and a are alpha-identical. There is no beta-redex, so (a) is already in normal form and reduces to a.

- Church booleans

  $$\texttt{true} \equiv \lambda t.\lambda f.\, t \qquad \texttt{false} \equiv \lambda t.\lambda f.\, f$$

  and the conditional

  $$\texttt{if} \equiv \lambda b.\lambda x.\lambda y.\, b\ x\ y.$$

  For example

  $$\texttt{if true}\ M\ N \Rightarrow_\beta^*\ M,$$

  and

  $$\texttt{if false}\ M\ N \Rightarrow_\beta^*\ N.$$

  The interpreter returns the expected branch in both cases.

- Church numerals

  $$\mathbf{0} \equiv \lambda f.\lambda x.\, x, \quad \mathbf{1} \equiv \lambda f.\lambda x.\, f\, x, \quad \mathbf{2} \equiv \lambda f.\lambda x.\, f(fx), \quad \mathbf{3} \equiv \lambda f.\lambda x.\, f(f(fx)).$$

  From HW5 we know that

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

  should reduce to the Church numeral for 9, i.e.

  $$\mathbf{9} \equiv \lambda f.\lambda x.\, \underbrace{f(f(\ldots f}_{9\text{ times}}\, x \ldots)).$$

  The interpreter indeed normalizes this term to a lambda-term of the form $\lambda f.\lambda x.\, f^9 x$.

These experiments support the claim that the implementation conforms to the intended operational semantics (normal-order beta-reduction with alpha-conversion).

Item 3: Capture-avoiding substitution

Substitution is written $[x := N]M$, meaning "replace all *free* occurrences of x in M by N, avoiding variable capture". The mathematical definition is by structural recursion on M.

1. Variables:
$$[x := N]x \;=\; N \qquad [x := N]y \;=\; y \quad \text{if } y \neq x.$$

2. Application:
$$[x := N](M_1\, M_2) \;=\; ([x := N]M_1)\; ([x := N]M_2).$$

3. Abstraction: there are three cases.
$$[x := N](\lambda y.M) \;=\; \begin{cases} \lambda y.M, & \text{if } y = x, \\ \lambda y.[x := N]M, & \text{if } y \neq x \text{ and } y \notin \mathrm{FV}(N), \\ \lambda z.\,[x := N]\,M[y := z], & \text{if } y \neq x \text{ and } y \in \mathrm{FV}(N), \end{cases}$$

   where z is a fresh variable not occurring free in M or N and $M[y := z]$ is a *capture-avoiding* renaming of the bound variable y to z.

   The third case is where alpha-conversion happens: before substituting, we rename the binder λy to a fresh λz so that the free variables of N never accidentally become bound.

Implementation sketch. In `interpreter.py` this is represented as a recursive `substitute(var, repl, term)` function, where `term` is an AST node of one of three kinds: variable, abstraction, or application. When substitution enters an abstraction $\lambda y.M$ and the bound variable `y` is in the free variables of `repl`, the interpreter first generates a fresh name like `Var1`, `Var2`, ..., renames the binder and its bound occurrences (alpha-conversion), and only then recurses. This is exactly the algorithm described above.

Item 4: Do all computations reduce to normal form?

For all "well-behaved" examples from the lectures (Church booleans, numerals, pairs, lists, etc.) the interpreter produced the expected normal forms. In particular, finite numeric computations and conditionals always normalized.

However, the untyped lambda calculus is Turing-complete, so there exist terms with no normal form. For such terms the interpreter keeps performing beta reductions forever (or, in practice, runs until we stop it), and no normal form is reached. Thus:

- For all test cases encoding finite computations, the interpreter gives the expected result.
- Not all lambda-terms reduce to normal form; some diverge.

Item 5: Smallest non-normalizing λ-expression

A standard minimal working example (MWE) of a non-normalizing term is

$$\Omega \;\equiv\; (\lambda x.\, x\, x)\; (\lambda x.\, x\, x).$$

Reduction:
$$(\lambda x.\, x\, x)\; (\lambda x.\, x\, x) \;\Rightarrow_\beta\; [x := \lambda x.\, x\, x](x\, x) \;=\; (\lambda x.\, x\, x)\; (\lambda x.\, x\, x) \;=\; \Omega.$$

So one beta-reduction step takes us back to the same term. Every proper subterm of Ω is either a variable or an abstraction and hence already in normal form, so Ω is a very small example of a term that does not reduce to normal form.

Item 7: Substitution trace for $\big((\lambda m.\lambda n.\, m\, n)(\lambda f.\lambda x.\, f(fx))\big)(\lambda f.\lambda x.\, f(f(fx)))$

We want to mirror the interpreter's beta-reduction steps, focusing on substitution. Below, each line corresponds to a single substitution (beta-step); for readability we omit some obvious alpha-renamings and keep variable names distinct by convention.

$$
\begin{aligned}
&\big((\lambda m.\lambda n.\, m\, n)\,(\lambda f.\lambda x.\, f(fx))\big)\,(\lambda f.\lambda x.\, f(f(fx))) \\
\Rightarrow_\beta\ &(\lambda n.\,(\lambda f.\lambda x.\, f(fx))\, n)\,(\lambda f.\lambda x.\, f(f(fx))) && \text{substitute } m := (\lambda f.\lambda x.\, f(fx)) \\
\Rightarrow_\beta\ &(\lambda n.\,\lambda x.\, n(nx))\,(\lambda f.\lambda x.\, f(f(fx))) && \text{substitute } f := n \text{ in } \lambda x.\, f(fx) \\
\Rightarrow_\beta\ &\lambda x.\,(\lambda f.\lambda x.\, f(f(fx)))\big((\lambda f.\lambda x.\, f(f(fx)))\, x\big) && \text{substitute } n := (\lambda f.\lambda x.\, f(f(fx))) \\
\Rightarrow_\beta\ &\lambda x.\,(\lambda x'.\,(\lambda f.\lambda x.\, f(f(fx)))\,(f_1)) && \text{(alpha-renaming of inner } x \text{ to } x') \\
&\qquad \text{where } f_1 \text{ abbreviates } \lambda x.\, f(f(fx)) \\
\Rightarrow_\beta\ &\lambda f.\lambda x.\, \underbrace{f(f(\ldots f}_{9 \text{ times}}\, x \ldots))
\end{aligned}
$$

The detailed middle steps just repeatedly expand the nested applications of the Church numeral **3** to itself; after fully unwinding, the body contains exactly nine occurrences of f, so the result is the Church numeral **9**.

Item 8: Recursive evaluation trace for $\big((\lambda m.\lambda n.\, m\, n)(\lambda f.\lambda x.\, f(fx))\big)(\lambda f.\lambda x.\, fx)$

Here we sketch the call trace of the interpreter's `evaluate` and `substitute` functions, in the same style as the recursive trace of `hanoi`. We write $\ell_{12}, \ell_{39}, \ldots$ as symbolic labels for the source-line numbers where these calls occur; in your own trace you should replace them with the actual line numbers from `interpreter.py`.

```
12: evaluate( ((\m.(\n.(m n))) (\f.(\x.(f (f x))))) (\f.(\x.(f x))) )
  39: evaluate( (\m.(\n.(m n))) (\f.(\x.(f (f x)))) )
    52: evaluate( \m.(\n.(m n)) )
    53: evaluate( \f.(\x.(f (f x))) )
    70: substitute( m := (\f.(\x.(f (f x))))
                    in  \n.(m n) )
      80: (possible alpha-conversion inside substitute)
    90: evaluate( \n.((\f.(\x.(f (f x)))) n) )
  39: evaluate( \f.(\x.(f x)) )
  70: substitute( n := (\f.(\x.(f x)))
                  in  ((\f.(\x.(f (f x)))) n) )
    80: (substitute into application)
      85: substitute into function part (\f.(\x.(f (f x))))
      86: substitute into argument n
  95: evaluate( (\f.(\x.(f (f x)))) (\f.(\x.(f x))) )
    52: evaluate( \f.(\x.(f (f x))) )
    53: evaluate( \f.(\x.(f x)) )
    70: substitute( f := (\f.(\x.(f x)))
                    in  \x.(f (f x)) )
      80: (alpha-conversion if needed)
    90: evaluate( \x.(g (g x)) )   -- where g = (\f.(\x.(f x)))
```

Conceptually, this trace shows that:

- The outermost call to `evaluate` sees an application, so it recursively evaluates the function and argument parts.
- For each beta-reduction, the interpreter calls `substitute` with the bound variable, the argument term, and the body of the abstraction, performing alpha-conversion where necessary.
- The recursive structure of the calls mirrors the structure of nested applications in the lambda-term, just as the recursive calls of `hanoi` mirror the recursive problem decomposition.

2.13.2 Questions

When experimenting with the interpreter, what is one example where the evaluation did not terminate in a normal form, and what does this tell you about the expressive power (and risks) of the untyped lambda calculus?

3 Essay

4 Evidence of Participation

5 Conclusion

References

[BLA] Author, Title, Publisher, Year.