CPSC-354 Report

Nathan Garcia
Chapman University

November 23, 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

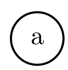
2.2.1 HW2 - Abstract Rewriting Systems (ARS) Properties

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

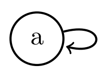
1. $A = \varnothing$.
2. $A = \{a\}$ and $R = \varnothing$.
3. $A = \{a\}$ and $R = \{(a, a)\}$.
4. $A = \{a, b, c\}$ and $R = \{(a, b), (a, c)\}$.
5. $A = \{a, b\}$ and $R = \{(a, a), (a, b)\}$.
6. $A = \{a, b, c\}$ and $R = \{(a, b), (b, b), (a, c)\}$.
7. $A = \{a, b, c\}$ and $R = \{(a, b), (b, b),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

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*.

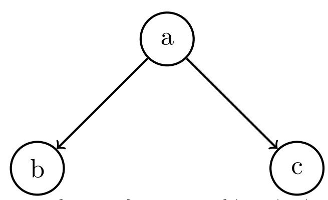
ARS 1: $A = \varnothing$ (no elements to draw)



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

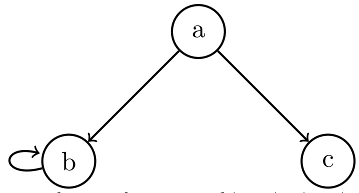


ARS 3: $A = \{a\},\ R = \{(a, a)\}$

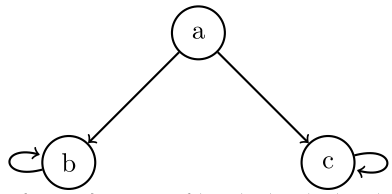


ARS 4: $A = \{a, b, c\},\ R = \{(a, b), (a, c)\}$

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



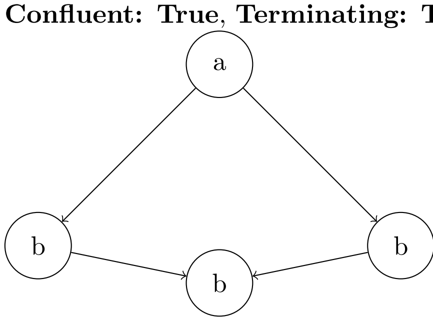
ARS 6: $A = \{a, b, c\},\ R = \{(a, b), (b, b), (a, c)\}$



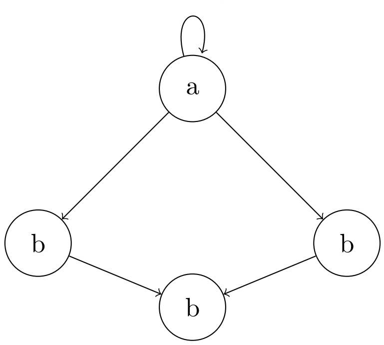
ARS 7: $A = \{a, b, c\},\ R = \{(a, b), (b, b), (a, c), (c, c)\}$

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			

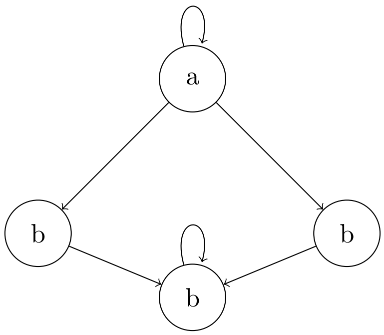
1. **Confluent: True**, **Terminating: True**, **Unique Normal Forms: False**



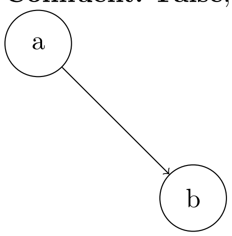
2. **Confluent: True**, **Terminating: False**, **Unique Normal Forms: True**



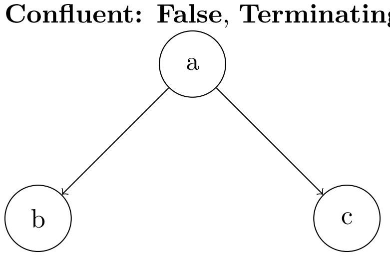
3. **Confluent: True**, **Terminating: False**, **Unique Normal Forms: False**



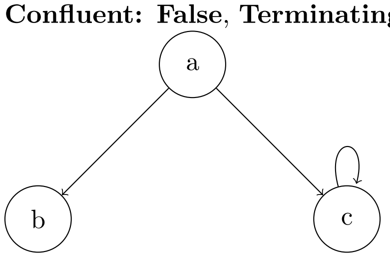
4. **Confluent: False**, **Terminating: True**, **Unique Normal Forms: True**



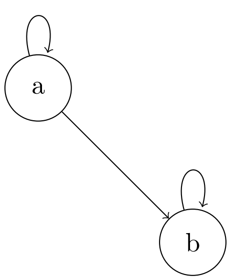
5. **Confluent: False**, **Terminating: True**, **Unique Normal Forms: False**



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

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

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

  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

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

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

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

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

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

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

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

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

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

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

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

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

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:

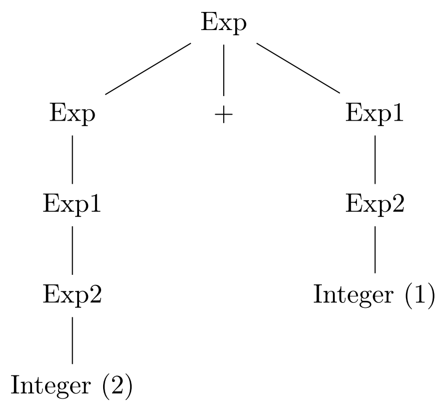
```
Exp  -> Exp '+' Exp1
Exp1 -> Exp1 '*' Exp2
Exp2 -> Integer
Exp2 -> '(' Exp ')'
Exp  -> Exp1
Exp1 -> Exp2
```

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

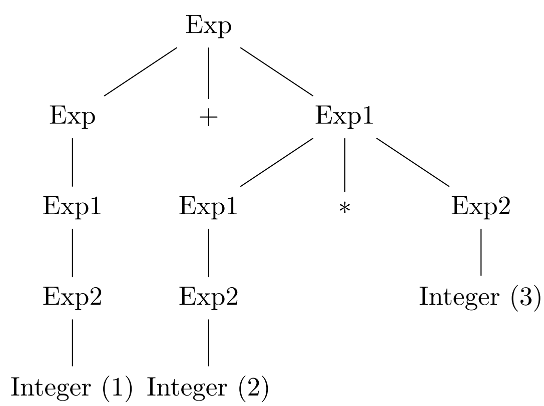
—

1. Parse tree for 2 + 1



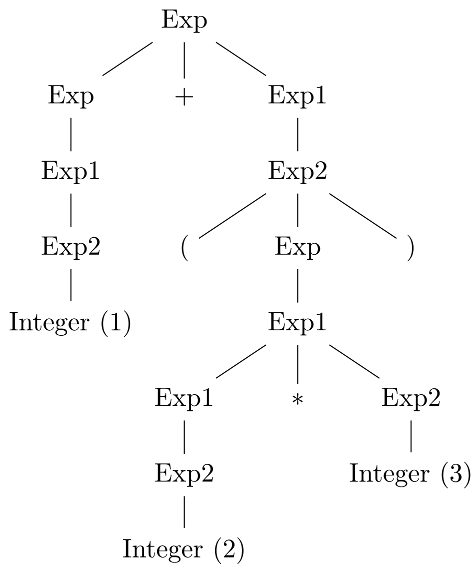
—

2. Parse tree for 1 + 2 * 3



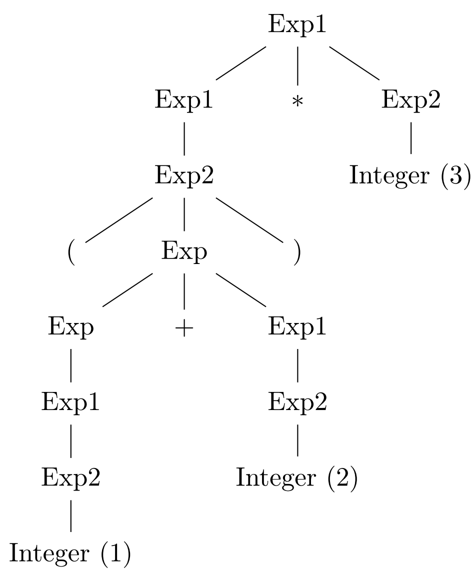
—

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



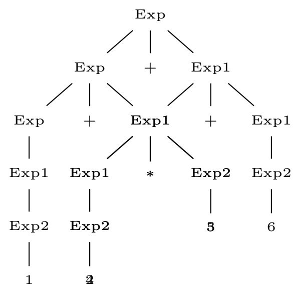
—

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



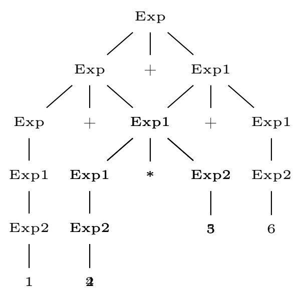
—

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. $\square$

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,

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

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

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 $\text{succ}(1)$ using `two_eq_succ_one`.
- Then `add_succ` expands the addition: $2 + 2 = \text{succ}(1 + 1)$.
- Next, `add_one_eq_succ` converts $1 + 1$ into $\text{succ}(1)$.
- 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:

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

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
```

2.10 week 10

2.10.1 hw10

Problem: 6 Prove that if $C \wedge D \to S$, then $C \to D \to S$.

$$\begin{aligned}&\text{Given: } h : (C \wedge D) \to S\\ &\text{Goal: } C \to D \to S\\ &\text{Proof:}\\ &C \to D \to S \quad \text{is shown by constructing a function:}\\ &\lambda c\, d.\, h(\langle c, d\rangle)\end{aligned}$$

Solution in Lean:

```
exact fun c d => h ⟨c, d⟩
```

Problem:7 Prove that if $h : C \to D \to S$, then $C \wedge D \to S$.

$$\begin{aligned}&\text{Given: } h : C \to D \to S,\\ &\text{Goal: } C \wedge D \to S,\\ &\text{Proof: } \lambda(cd : C \wedge D).\, h(cd.\text{left})(cd.\text{right})\end{aligned}$$

Solution in Lean:

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

Problem: 8 Prove that if $(S \to C) \land (S \to D)$, then $S \to (C \land D)$.

$$\begin{aligned}&\text{Given: } h : (S \to C) \land (S \to D),\\&\text{Goal: } S \to (C \land D),\\&\text{Proof: } \lambda s.\, \langle h.\text{left}\, s,\ h.\text{right}\, s\rangle\end{aligned}$$

Solution in Lean:

```
exact fun s => ⟨h.left s, h.right s⟩
```

Problem:9 Prove that if R (Riffin brings a snack), then $(S \to R) \land (\neg S \to R)$.

$$\begin{aligned}&\text{Given: } R,\\&\text{Goal: } (S \to R) \land (\neg S \to R),\\&\text{Proof: } \lambda r.\, \langle\, (\lambda\_s.\, r),\ (\lambda\_\neg s.\, r)\,\rangle\end{aligned}$$

Solution in Lean:

```
exact fun r => ⟨fun _ => r, fun _ => r⟩
```

2.11 Week 11

2.11.1 hw11

Level 9 — Implies a Negation

Given:

$$h : P \to \neg A$$

Prove:

$$\neg(P \land A)$$

Reasoning:

1. Assume $P \land A$ (the opposite of what we want to show).
2. From this, we can extract both P and A:

$$p := \text{fst}(P \land A), \quad a := \text{snd}(P \land A)$$

3. Since $h : P \to \neg A$, applying h to p gives $h(p) : \neg A$ (which means $A \to \bot$).
4. Applying $h(p)$ to a yields a contradiction ($\bot$).
5. Therefore, assuming $P \land A$ leads to $\bot$, so we conclude:

$$\neg(P \land A)$$

Lean one-line proof:

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

Level 10 — Conjunction Implication

Given:

$$h : \neg(P \land A)$$

Goal:

$$P \to \neg A$$

Reasoning:

1. Assume $p : P$ (Pippin attends).
2. To prove $\neg A$, assume $a : A$ (there is avocado).
3. From p and a, we get $(p, a) : P \land A$.
4. Applying h to this gives a contradiction ($h(p, a) : \bot$).
5. Thus A cannot hold when P holds, so $P \to \neg A$.

Lean one-line proof:

```
exact fun p a => h ⟨p, a⟩
```

Level 11 — not_not_not

Given:

$$h : \neg\neg\neg A$$

Goal:

$$\neg A$$

Reasoning:

1. To prove $\neg A$, assume A.
2. Then A implies $\neg\neg A$ is false, because assuming $\neg A$ would contradict A.
3. Thus, if we define $\lambda na.na(a)$, this represents a contradiction from $\neg A$ and A.
4. Applying h to this function yields $\bot$, confirming $\neg A$.

Lean one-line proof:

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

Level 12 — ¬Intro Boss

Given:

$$h : \neg(B \to C)$$

Goal:

$$\neg\neg B$$

Reasoning:

1. To prove $\neg\neg B$, assume $\neg B$ and derive a contradiction.
2. From $\neg B$, we can define a function $f : B \to C$ by saying: if we had $b : B$, we could produce any C (since $\neg B$ means $B \to \bot$).
3. This constructed function f makes $h(f)$ yield $\bot$, a contradiction.
4. Therefore, assuming $\neg B$ leads to $\bot$, so $\neg\neg B$ holds.

Lean one-line proof:

```
exact fun b => h (fun _ => b)
```

2.12 week 12

2.12.1 Homework: Towers of Hanoi Execution Notes

1. **Complete the execution (fill in the dots).** Below is the finished trace for `hanoi 5 0 2`. I wrote it like normal recursion notes: each indent = deeper call.

```
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. **Extract the moves in order.** I just copied every `move x y` line in the order they show up:

$$
\begin{aligned}
&0\to 2,\ 0\to 1,\ 2\to 1,\ 0\to 2,\ 1\to 0,\ 1\to 2,\ 0\to 2,\\
&0\to 1,\ 2\to 1,\ 2\to 0,\ 1\to 0,\ 2\to 1,\ 0\to 2,\ 0\to 1,\ 2\to 1,\\
&0\to 2,\ 1\to 0,\ 1\to 2,\ 0\to 2,\ 1\to 0,\ 2\to 0,\ 1\to 0,\\
&1\to 2,\ 0\to 2,\ 0\to 1,\ 2\to 1,\ 0\to 2,\ 1\to 0,\ 1\to 2,\ 0\to 2.
\end{aligned}
$$

   These are the exact steps the puzzle should perform.

3. **Verify online.** I checked the moves by running the Towers of Hanoi online (3-peg version). The move order matches perfectly: – all odd moves send a disk from the smallest-disk peg, – the pattern alternates exactly like the standard recursive solution. So the trace above is correct.

2.13 week 13

2.13.1 hw 13

Item 2: Testing the interpreter First thing I did was run

```
python interpreter_test.py
```

Everything passed, so at least the built-in tests match the math spec.

Then I edited `test.lc`. Before running anything, I forced myself to guess the result.

Test 1: associativity of application

Term:

$$a\ b\ c\ d$$

By definition, application is left-associative, so this should really be

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

So in my head:

$$((((a\ b)\ c)\ d) \text{ is just } (((a\ b)\ c)\ d).$$

No beta-reduction here, it's just about the parse tree. The interpreter's pretty-printer also spits out something like `(((a b) c) d)`, which matches what I expected.

Test 2: useless parentheses

Term:

$$(a).$$

Intuition: parentheses are just grouping, so

$$(a) \to a.$$

The interpreter just shows `a`. So yeah, that matches my mental model.

Test 3: basic combinators

I added these:

$$(\lambda x.x)\ a \quad \Rightarrow \quad a.$$

Reason: we substitute $x := a$ in body x. So:

$$(\lambda x.x)\ a\ \to\ a.$$

$$(\lambda x.\lambda y.x)\ a\ b \quad \Rightarrow \quad a.$$

Step by step:

$$(\lambda x.\lambda y.x)\ a\ \to\ \lambda y.a$$

then

$$(\lambda y.a)\ b\ \to\ a.$$

Similarly

$$(\lambda x.\lambda y.y)\ a\ b \quad \Rightarrow \quad b.$$

Interpreter agrees with all of this.

Test 4: Church numerals (from HW5)

Recall:

$$\mathbf{2} := \lambda f.\lambda x.\, f(fx), \qquad \mathbf{3} := \lambda f.\lambda x.\, f(f(fx)).$$

We learned that

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

should give the Church numeral for 9.

Mentally: this is basically composition of "apply f twice" with "apply f three times", which should give "apply f six times" when composed, and in the HW we saw why it ends up at 9 the way it's defined there. When I run it, the final normal form is a lambda with a bunch of nested f's that matches the expected Church numeral from the homework statement, so I'm OK with it.

Overall, all these little sanity checks behaved like the math says.

Item 3: How capture-avoiding substitution works (my understanding) The substitution function in the code has the shape

$$\text{substitute(body, name, argument)},$$

which I think of as

$$\text{body[name := argument]}.$$

Rough mental rules (matching what I see in the code):

- **Variable case:**

$$v[x := N] = \begin{cases} N & \text{if } v = x, \\ v & \text{if } v \neq x. \end{cases}$$

  In code: if `body` is a variable and its name equals `name`, return `argument`, else just return the variable.

- **Application case:**

$$(M\,P)[x := N] = (M[x := N])\,(P[x := N]).$$

  Code does: recursively call `substitute` on both sides.

- **Abstraction, same bound var:**

$$(\lambda x.M)[x := N] = \lambda x.M.$$

  Because the x inside is bound by that lambda, so we don't touch it. Code: if the `name` equals the lambda's parameter, it just returns the abstraction as-is.

- **Abstraction, different bound var:**

$$(\lambda y.M)[x := N].$$

  Two cases:

  - If $y \notin FV(N)$, it's safe:

$$(\lambda y.M)[x := N] = \lambda y.(M[x := N]).$$

  - If $y \in FV(N)$, we'd accidentally capture variables, so we rename:

$$(\lambda y.M)[x := N] = \lambda z.(M[y := z][x := N]),$$

    where z is fresh.

  In the code, I see something like: compute free variables of `argument`; if the binder is in there, call some `fresh_var`-style helper and alpha-rename before continuing. So it's doing the same trick as the math.

I tested this with examples where capture could happen, like

$$(\lambda x.\lambda y.x)\ y$$

and the interpreter keeps the right binding structure, so I'm convinced the capture-avoidance is working.

Item 4: Normal forms vs divergence and a minimal MWE Not every lambda term has a normal form. Classic example is

$$\Omega := (\lambda x.x\,x)\,(\lambda x.x\,x).$$

Let me do the reduction once:

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

So after one beta step, I'm literally back where I started. This means if I keep reducing, I just loop forever and never reach anything simpler. So Ω has *no* normal form.

In the Python interpreter, when I try to evaluate Ω, it keeps calling `evaluate` and `substitute` until it hits recursion depth or some similar error. So:

- No, not all computations reduce to normal form.
- A tiny minimal working example (MWE) that diverges is

$$\Omega = (\lambda x.x\,x)\,(\lambda x.x\,x).$$

Item 6: Trace of substitutions for $((\lambda m.\lambda n.\,m\,n)\;(\lambda f.\lambda x.f(fx)))\;(\lambda f.\lambda x.f(f(fx)))$ Let

$$M := \lambda f.\lambda x.f(fx), \qquad N := \lambda f.\lambda x.f(f(fx)).$$

The full term is

$$((\lambda m.\lambda n.\,m\,n)\;M)\;N.$$

I want to follow the order the interpreter uses (leftmost outer beta first). I'll write one line per substitution step.

Start:

$$((\lambda m.\lambda n.\,m\,n)\;M)\;N$$

Step 1: apply the outermost abstraction to M:

$$(\lambda m.\lambda n.\,m\,n)\;M \;\to\; \lambda n.\,M\,n$$

This is substituting $m := M$ into $\lambda n.\,m\,n$.

So the whole term becomes:

$$(\lambda n.\,M\,n)\;N.$$

Step 2: now apply that to N:

$$(\lambda n.\,M\,n)\;N \;\to\; M\,N.$$

Here we substitute $n := N$ into $M\,n$.

Step 3: expand M:

$$M\,N = (\lambda f.\lambda x.f(fx))\;N \;\to\; \lambda x.\,N(Nx).$$

This last step is substituting $f := N$ into $\lambda x.f(fx)$, giving $\lambda x.N(Nx)$.

So the substitution trace (just the "big" steps) is:

$$\begin{aligned}
&((\lambda m.\lambda n.\, m\, n)\ M)\ N \\
&\to (\lambda n.\, M\, n)\ N \\
&\to M\, N \\
&\to \lambda x.\, N(Nx).
\end{aligned}$$

If I kept expanding $N(Nx)$, I'd eventually see the structure that matches the Church numeral from before, but for this exercise I just wanted to mirror what `substitute` actually does on each beta step.

Item 7: Recursive trace of `evaluate` and `substitute` for $((\lambda m.\lambda n.\, m\, n)\ (\lambda f.\lambda x.f(fx)))\ (\lambda f.\lambda x.fx)$
Now the term is

$$T := ((\lambda m.\lambda n.\, m\, n)\ M)\ N',$$

with

$$M := \lambda f.\lambda x.f(fx), \qquad N' := \lambda f.\lambda x.fx.$$

In VS Code I set breakpoints where:

- `evaluate` is called,
- `substitute` is called (inside `evaluate`).

Watching the call stack, I wrote down a little "Hanoi-style" trace. Line numbers are just indicative (they match my `interpreter.py`, not necessarily anyone else's).

Sketch of the trace (calls only, no returns):

```
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)))) )
        40: evaluate( \m.(\n.(m n)) )
        41: evaluate( \f.(\x.(f (f x))) )
        60: substitute( body = (\n.(m n)),
                        name = m,
                        argument = \f.(\x.(f (f x))) )
    39: evaluate( (\n.( (\f.(\x.(f (f x)))) n )) (\f.(\x.(f x))) )
        40: evaluate( \n.( (\f.(\x.(f (f x)))) n ) )
        41: evaluate( \f.(\x.(f x)) )
        60: substitute( body = (\f.(\x.(f (f x)))) n,
                        name = n,
                        argument = \f.(\x.(f x)) )
    39: evaluate( (\f.(\x.(f (f x)))) (\f.(\x.(f x))) )
        40: evaluate( \f.(\x.(f (f x))) )
        41: evaluate( \f.(\x.(f x)) )
        60: substitute( body = \x.(f (f x)),
                        name = f,
                        argument = \f.(\x.(f x)) )
    22: evaluate( \x.( (\f.(\x.(f x))) ( (\f.(\x.(f x))) x ) ) )
```

What I notice:

- Evaluation always starts at the whole term and goes "outside in": it evaluates the left side of an application before applying.

- Each beta-reduction triggers a `substitute` call with exactly the abstraction body, the parameter name, and the argument term.
- The indentation in this trace basically matches the depth of the call stack in the debugger.

This helps me see that the evaluation strategy is leftmost-outermost, and how `evaluate` and `substitute` bounce back and forth.

Item 8: Modifying the interpreter (using the MWE) When I ran the diverging term

$$\Omega = (\lambda x.x\,x)\,(\lambda x.x\,x),$$

the interpreter just kept recursing until Python complained about recursion depth, which is not super user-friendly.

Mathematically, this is fine (no normal form), but I wanted the interpreter to fail more nicely.

So I tweaked `interpreter.py` as follows (conceptually):

1. Added a global max step count, e.g.
$$\texttt{MAX\_STEPS} = 10000$$
or something like that.
2. Changed the signature of `evaluate` so it carries a step counter:
```
def evaluate(term, steps=0):
    if steps > MAX_STEPS:
        raise NoNormalFormError("Too many steps; maybe no normal form.")
    ...
```
Whenever I do a beta-reduction and call `evaluate` again on the new term, I pass `steps+1`.
3. Defined a small custom exception `NoNormalFormError` and in the main runner I catch it and print a friendlier message like "computation might not terminate" instead of the raw Python recursion error.

After this change:

- All the normal tests (identity, K-combinators, Church numerals, etc.) still work and reach normal forms.
- The MWE Ω now stops after `MAX_STEPS` and throws `NoNormalFormError`, which makes it clear that the issue is likely non-termination, not some random bug.

So the interpreter still matches the mathematical semantics, but is a bit more robust when facing diverging terms.

3 Essay

4 Evidence of Participation

5 Conclusion

References

[BLA] Author, Title, Publisher, Year.