CPSC-354 Report

Jalen Myers
Chapman University

October 12, 2025

Abstract

Contents

1 Introduction

2 Week by Week

2.1 Week 1

2.1.1 The MU Puzzle

The MU Puzzle, introduced by Douglas Hofstadter in *Gödel, Escher, Bach*, is a formal string-rewriting puzzle. You start with the axiom **MI** and attempt to transform it into the target string **MU** using four simple rules. At first glance, it appears to be a game of clever substitutions and expansions, but underneath lies a lesson about **formal systems**, **invariants**, and **proof by impossibility**. The puzzle demonstrates that even though the rules allow endless local manipulations, there are hidden global constraints that block certain outcomes. In this case, the invariant is the number of I's modulo 3, which prevents ever reaching MU. It is less about "finding the clever sequence" and more about recognizing when a goal is **provably impossible**.

Rules:

1. Append U after I: $xI \to xIU$
2. Double after M: $Mx \to Mxx$
3. Swap III for U: $xIIIy \to xUy$
4. Delete UU: $xUUy \to xy$

Key invariant (why MU is impossible): Starting with $MI \to MII \to MIIII \to MUIUUIU$, no matter how the rules are applied, the number of I's never reaches zero. For example:

$$MI \to MII \to MIIII \equiv 1 \pmod 3, \quad MIIII \to MIU \ (I = 1)$$

The invariant that the number of I's modulo 3 is never 0 prevents reaching MU.

Rule interactions with I:

- Rule 1: Does not touch I's, but can create UU opportunities for Rule 4 cleanup.
- Rule 2: The only way to increase I's; preserves the parity cycle $\{1, 2\}$ modulo 3.
- Rule 3: The only way to reduce I's; reduces them by 3 at a time, keeping modulo 3 unchanged.
- Rule 4: Purely cosmetic with respect to I's; manages U's only.

Thus, I's are the crucial resource; U's are merely bookkeeping.

Example derivation:

$$MI\ (I = 1) \xrightarrow{\text{Rule 2}} MII\ (I = 2) \xrightarrow{\text{Rule 2}} MIIII\ (I = 4 \equiv 1 \pmod 3)$$

$$MIIII \xrightarrow{\text{Rule 3}} MIU\ (I = 1) \xrightarrow{\text{Rule 1}} MIUU\ (I = 1)$$

Deleting UU later still leaves $I = 1$. The cycle alternates between $I \equiv 1$ and $I \equiv 2$, never reaching 0.

Starting with Rules 1 or 2: You can start with either Rule 1 or Rule 2, but it does not matter for reachability of MU. The invariant blocks MU either way.

2.1.2 Rule Modification: UU → I

If Rule 4 is modified to $xUUy \to xIy$, then the invariant breaks down. For instance:

$$MU \xrightarrow{\text{Rule 2}} MUU \xrightarrow{\text{Rule 4}'} MI$$

This allows us to get from MU back to MI in two steps. More generally, this alteration lets us change the number of I's modulo 3, opening paths that were previously impossible.

2.1.3 Local Rules, Global Truths

This puzzle is a tiny string rewriting system. The surprising aspect is that local rewrites can create global invariants (like $I \pmod 3$) that dictate what outcomes are possible. The method of solving involves:

1. Finding a conserved quantity,
2. Classifying each rule's effect on that quantity,
3. Concluding reachability or non-reachability of the target.

A single change in the rule set (e.g., replacing UU deletion with UU → I) completely alters the invariant landscape, showing how local edits ripple globally.

2.1.4 Conclusion

The MU Puzzle highlights how simple rules can create surprising limits. By tracking the number of I's, we saw that modular arithmetic exposes an invariant invisible at first glance. Rules 1 and 4 do not change I's, Rule 2 doubles them, and Rule 3 reduces them by three. Together, they trap the system in a cycle that never reaches zero I's, so MU cannot be produced.

Changing just one rule, however, completely alters the system: the invariant breaks down, and new possibilities emerge. This shift shows the deeper lesson: local changes ripple globally, and sometimes the only way to understand a system is to invent a new measuring stick. In the end, the puzzle is less about actually getting to MU and more about learning how formal reasoning and invariants reveal truths that trial-and-error never could.

2.2 Week 2

2.2.1 Rewriting ARSs

ARSs (Abstract Rewriting Systems)

List of ARSs:

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

2.2.2 ARSs with drawings and Explanations:

1. $A = \varnothing$.

   This systems is terminating, confluent and has unique normal forms, because there are no elements and no rewrite rules. Nothing can happen.

2. $A = \{a\},\ R = \{\}$.

   $$a$$

   We have one element a, but no rules. Since no rewrites are possible, a is already normal form. The system is terminating, confluent and has a unique normal form, which is just a.

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

   $$\begin{tikzcd} a \arrow[loop above] \end{tikzcd}$$

   Now a rewrites to itself. This means the system is not terminating because you can loop forever. But, it is trivially confluent, since there are no diverging rewrite paths. There is no normal form, because it can always be rewritten again. It does not have unique normal forms.

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

   $$\begin{tikzcd} a \arrow[r] \arrow[dr] & b \\ & c \end{tikzcd}$$

   Here you can rewrite to two options. Since neither can be rewritten further, the system is terminating. However, it is not confluent because you can reduce to two different irreducible terms. As a result, it does not have unique normal forms.

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

   $$\begin{tikzcd} a \arrow[loop above] \arrow[r] & b \end{tikzcd}$$

   In this case, you can either loop back to itself or reduce. The presence of the self-loop makes the system non-terminating. But it is still confluent, because both paths eventually lead to it, and it is the only normal form. Even though the system does not terminate in general, it does have a unique normal form.

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

   $$\begin{tikzcd} & a \arrow[dl] \arrow[dr] & \\ b \arrow[loop above] & & c \end{tikzcd}$$

   Here, you can reduce to either two opetions. The element of one of them loops onto itself and so is not a normal form, while the other is irreducible and therefore a normal form. Because of this loop, the system is not terminating, and because it can reduce to either a looping form, the system is not confluent and does not have a unique normal forms.

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

$$\begin{tikzcd} & a \arrow[dl] \arrow[dr] & \\ b \arrow[loop above] & & c \arrow[loop above] \end{tikzcd}$$

This case can either go to two forms but both can loop on themselves. That means there are no normal forms at all. The system is not terminating, not confluent, and has no unique normal forms.

2.2.3 Mapping ARSs Covering All 8 Property Combinations:

Below is a classification of the 8 combinations of *confluence*, *termination*, and *unique normal forms*. We can match each with one of our ARSs whenever possible. Some rows are marked impossible due to logical constraints.

1. **Confluent, Terminating, Unique NFs:** Example: ARS #2 with $A = \{a\}, R = \{\}$.

$$a$$

This system terminates immediately and is trivially confluent, and a is the unique normal form.

2. **Confluent, Terminating, No Unique NFs:** Impossible. If an ARS is both terminating and confluent, then it must have unique normal forms.

3. **Confluent, Non-terminating, Unique NFs:** Example: ARS #5 with $A = \{a, b\}, R = \{(a, a), (a, b)\}$.

$$\begin{tikzcd} a \arrow[loop above] \arrow[r] & b \end{tikzcd}$$

Although a can loop forever both a and b reduce to the same unique normal form b, so the system is confluent with unique NFs.

4. **Confluent, Non-terminating, No Unique NFs:** Example: ARS #3 with $A = \{a\}, R = \{(a, a)\}$.

$$\begin{tikzcd} a \arrow[loop above] \end{tikzcd}$$

The system has an infinite loop so it does not terminate, and since a never reaches a normal form there are no unique normal forms. It is still trivially confluent.

5. **Non-confluent, Terminating, Unique NFs:** Impossible. If every eement has a unique normal form, then reductions from any peak must join at that NF, which implies confluence.

6. **Non-confluent, Terminating, No Unique NFs:** Example: ARS #4 with $A = \{a, b, c\}, R = \{(a, b), (a, c)\}$.

$$\begin{tikzcd} a \arrow[r] \arrow[dr] & b \\ & c \end{tikzcd}$$

This system is terminating but it is not confluent, since a reduces to two distinct forms b and c.

7. **Non-confluent, Non-terminating, Unique NFs:** Impossible. Having unique normal forms implies confluence so this case cannot exist.
8. **Non-confluent, Non-terminating, No Unique NFs:** Example: ARS #6 (or #7).

$$\begin{tikzcd}
& a \arrow[dl] \arrow[dr] & \\
b \arrow[loop above] & & c \arrow[loop above]
\end{tikzcd}$$

From a you can go to b or c but both loop on themselves. There are no normal forms so the system is non-terminating, non-confluent, and has no unique NFs.

Notes:

- Two is impossible since termination and confluence always gives unique normal forms.
- Five is impossible since if every element has a unique normal form then confluence must hold.
- Seven is impossible for that same reason.

2.3 Week 3

2.3.1 String Rewriting Exercise

Exercise 5 We consider the rewite rules:

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

Example reductions

$$\texttt{abba} \to \texttt{aba} \to \texttt{aa} \to \varepsilon,$$

$$\texttt{bababa} \to \texttt{aaaba} \to \texttt{aaaa} \to \texttt{aa} \to \varepsilon.$$

Because of the swap rules ($ab \leftrightarrow ba$), there are many possible paths, including infinite loops.

Why the ARS is not terminating The two opposite rules $ab \to ba$ and $ba \to ab$ allow endless swapping:

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

So the system is non-terminating.

Equivalence classes - We can insert and delete any b. - We can insert and delete aa. - We can swap a's and b's arbitrarily.

The only invariant is the **parity of the number of a's**. Two equivalnce classes arise:

$$\begin{cases} \text{Even number of } a\text{'s} & \leftrightarrow^* \ \varepsilon, \\ \text{Odd number of } a\text{'s} & \leftrightarrow^* \ a. \end{cases}$$

Normal Forms Delete all b's then repeatedly delete aa. - If the string has an even number of a's it reduces to ε. - If it has an odd number of a's it reduces to a. So the normal forms are ε and a.

Termination fix We can orient swap in only one direction:

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

Using the measure "# of ba substrings + length," each step decreases this value, so the system terminates. Equivalence classes remain unchanged.

Sematic Questions

- Does a string have an even number of a's? (Yes if it reduces to ε.)
- What is the canonical representative? (ε for even, a for odd.)

Exercise 5b Now change the rule $aa \to \varepsilon$ to $aa \to a$:

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

Equivalence classes - $aa \leftarrow a$ allows changing the number of a's by $+1$ or -1 (but never reaching a from ε). - Therefore there are two classes:

$$\begin{cases} \text{Strings with no } a & \leftrightarrow^* \ \varepsilon, \\ \text{Strings with at least one } a & \leftrightarrow^* \ a. \end{cases}$$

Normal forms Delete all b's then repeatedly apply $aa \to a$ until at most one a remains. Normal forms are again ε and a.

Termination fix The system still loops under $ab \leftrightarrow ba$. As before we make it terminating by keeping one swap direction. The measure (# of ba's + length) still decreases, so terminating holds and the equivalnce classes remain unchanged.

2.4 Week 4

2.4.1 Termination

HW 4.1 Consider the following algorithm:

Algorithm

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

Conditions for Termination

The algorithm is well defined and terminates under the following conditions:

- Inputs $a, b \in \mathbb{N}$ (non-negative integers).
- The `mod` operator is the standard remainder: for $a, b \in \mathbb{N}$ with $b > 0$, there exist unique q, r such that $a = qb + r$ with $0 \le r < b$.
- Thus, after `b = a mod b`, we always have $0 \le b' < b$.

Measure Function

Define the measure function:

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

Proof of Termination

Suppose $b \neq 0$ before an iteration. The updated state is:

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

By the remainder property, $0 \leq b' < b$. Hence:

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

Thus μ strictly decreases at every iteration. Since μ takes values in $\mathbb{N}$, it is bounded below by 0. Therefore, the loop must terminate.

Conclusion: Under the given conditions, the algorithm always terminates.

HW 4.2 Consider the following fragment of an implementation of merge sort:

Algorithm Fragment

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

Measure Function

Define:

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

This is the length of the current subarray.

Proof of Termination

Base Case. If $\mathit{left} \geq \mathit{right}$, then $\phi(\mathit{left}, \mathit{right}) \leq 1$, and the function returns immediately (no recursion).

Recursive Case. Assume $\mathit{left} < \mathit{right}$ so that $n = \phi(\mathit{left}, \mathit{right}) \geq 2$. Set $\mathit{mid} = \lfloor (\mathit{left} + \mathit{right})/2 \rfloor$.

- **First recursive call:**
$$\phi(\mathit{left}, \mathit{mid}) = (\mathit{mid} - \mathit{left} + 1) \leq \left\lfloor \frac{n}{2} \right\rfloor < n.$$

- **Second recursive call:**
$$\phi(\mathit{mid} + 1, \mathit{right}) = (\mathit{right} - (\mathit{mid} + 1) + 1) = (\mathit{right} - \mathit{mid}) \leq \left\lceil \frac{n}{2} \right\rceil < n.$$

Both recursive calls strictly reduce the measure.

Well-Foundedness. Since ϕ maps to $\mathbb{N}$ and decreases strictly with each recursive call, and $\mathbb{N}$ under $<$ is well-founded, infinite recursion is impossible.

Conclusion: $\phi(\mathit{left}, \mathit{right}) = \mathit{right} - \mathit{left} + 1$ is a valid measure function for `merge_sort`, hence the algorithm always terminates.

2.5 Week 5

2.5.1 Lambda Calculus Workout: α & β Rules

We conisder the lambda calculus expression:

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

For clarity, define:

$$M \equiv \lambda f.\, \lambda x.\, f(f\, x), \quad N \equiv \lambda f.\, \lambda x.\, f(f(f\, x)).$$

Here M represents the Church numeral 2 (apply f twice), and N represents the numeral 3 (apply f three times). The problem reduces to computing $M\, N$.

2.5.2 Step-by-Step β-Reduction

Produced by β-reduction (with safe α-renaming where necesary):

$$\begin{aligned} M\, N &= (\lambda f.\, \lambda x.\, f(f\, x))\, N \\ &\xrightarrow{\beta} \lambda x.\, N(N\, x). \end{aligned}$$

Now expand N:

$$N = \lambda f.\, \lambda y.\, f(f(f\, y)) \quad \text{(renamed inner } x \text{ to } y\text{)}.$$

Step 1: Evaluate $N\, x$.

$$N\, x = (\lambda f.\, \lambda y.\, f(f(f\, y)))\, x \xrightarrow{\beta} \lambda y.\, x(x(x\, y)) \;=\; G.$$

Step 2: Evaluate $N\, G$.

$$N\, G = (\lambda f.\, \lambda y.\, f(f(f\, y)))\, G \xrightarrow{\beta} \lambda y.\, G(G(G\, y)).$$

Since $G\, y = x(x(x\, y))$, repeated three times, this results in nine total applications of x.

Final Result:

$$M\, N \;\xrightarrow{*}\; \lambda f.\, \lambda x.\, f(f(f(f(f(f(f(f(f\, x)))))))).$$

This is the Church numeral 9.

2.5.3 Interpretation and Haskell Representation

Mathematical Meaning. In general,

$$(\lambda f.\, \lambda x.\, f^m x)\; (\lambda f.\, \lambda x.\, f^n x) \;\equiv\; \text{Church numeral for } n^m.$$

Here $m = 2$ and $n = 3$, giving $3^2 = 9$.

Haskell-style Version Using `*.hs` with Haskell highlighting in VS Code:

```
-- M: apply f twice
m = \f -> \x -> f (f x)

-- N: apply f thrice
n = \f -> \x -> f (f (f x))

term = m n

-- Normal form (Church numeral 9):
nine = \f -> \x -> f (f (f (f (f (f (f (f (f x))))))))
```

Remark 2.1. Working in VS code with Haskell syntax highlighting makes parenthesis and step-by-step redution clearer.

2.6 Week 6

2.6.1 Computing `fact 3` with `fix`, `let`, and `let rec`

Following the computation rules (as given in prompt):

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

For brevity, let

$$\begin{aligned}
F &\triangleq \lambda f.\, \lambda n.\ \texttt{if}\ n = 0\ \texttt{then}\ 1\ \texttt{else}\ n * f\,(n-1),\\
\mathit{FixF} &\triangleq \texttt{fix}\ F.
\end{aligned}$$

Goal Compute

$$E_0 \;=\; \texttt{let rec fact} = \lambda n.\ \texttt{if}\ n = 0\ \texttt{then}\ 1\ \texttt{else}\ n * \texttt{fact}\,(n-1)\ \texttt{in fact}\ 3.$$

Labeled derivation

```
  -- <def of let rec>
  E1 = let fact = fix (\f. \n. if n=0 then 1 else n * f (n-1) in fact 3
  = let fact = FixF in fact 3)

  -- <def of let>
  E2 = ((\fact. fact 3) FixF)

  -- <beta-rule: substitute FixF or fact>
  E3 = FixF 3

  -- <def of fix>
  E4 = (F FixF) 3

  -- <beta-rule: apply F to FixF>
  E5 = (\n. if n=0 then 1 else n * FixF (n-1)) 3

  -- <beta-rule>, <if-false>, <arith>
```

```
E6 = 3 * FixF (3-1) = 3 * FixF 2

-- Unfold FixF at 2
-- <def of fix>
E7 = 3 * (F FixF) 2
-- <beta-rule>
E8 = 3 * ((\n. if n=0 then 1 else n * FixF (n-1)) 2)
-- <beta-rule>, <if-else>, <arith>
E9 = 3 * (2 * FixF (2-1)) = 3 * (2 * FixF 1)

-- Unfold FixF at 1
-- <def of fix>
E10 = 3 * (2 * (F FixF) 1)
-- <beta-rule>
E11 = 3 * (2 * ((\n. if n=0 then 1 else n * FixF (n-1)) 1))
-- <beta-rule>, <if-else>, <arith>
E12 = 3 * (2 * (1 * FixF (1-1))) = 3 * (2 * (1 * FixF 0))

-- Base case at 0
-- <def of fix>
E13 = 3 * (2 * (1 * (F FixF) 0))
-- <beta-rule>
E15 = 3 * (2 * (1 *1))

-- <arith>
E16 = 3 * (2 * 1) = 3 * 2 = 6
```

Result `fact` $3 \Downarrow 6$

2.7 Week 7

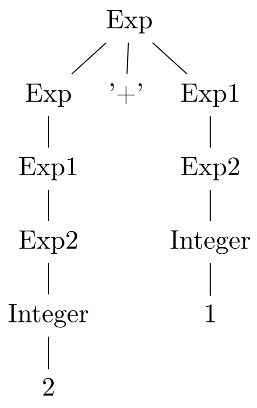
2.7.1 Parse trees for Arithmetic Expressions

In this hoemwork, I use the following context-free grammar:

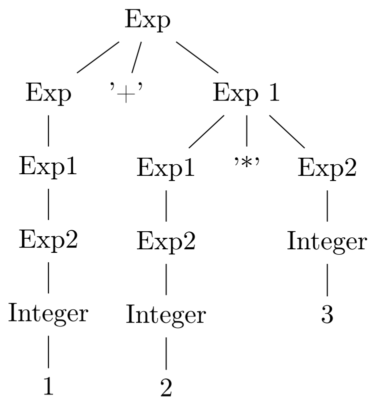
```
Exp -> Exp '+' Ex1 | Exp1
Exp1 -> Exp1 '*' Exp2 | Exp2
Exp2 -> Integer | '(' Exp ')'
```

Create derivation (parse) tress for each expression.

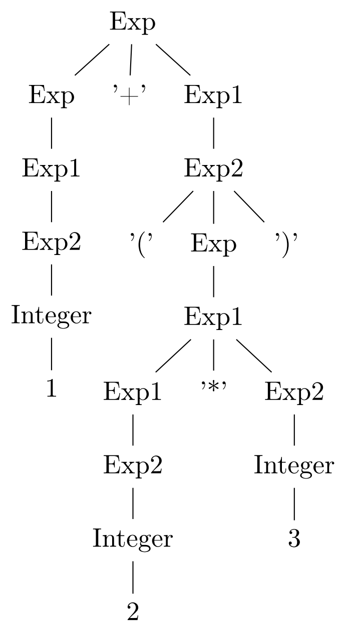
2.7.2 Expression: 2 + 1



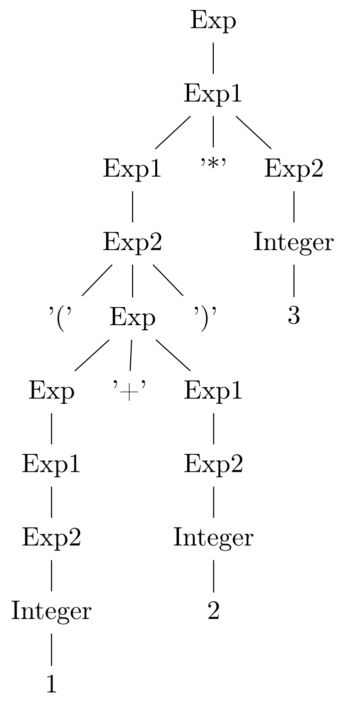
2.7.3 Expression: 1 + 2 * 3



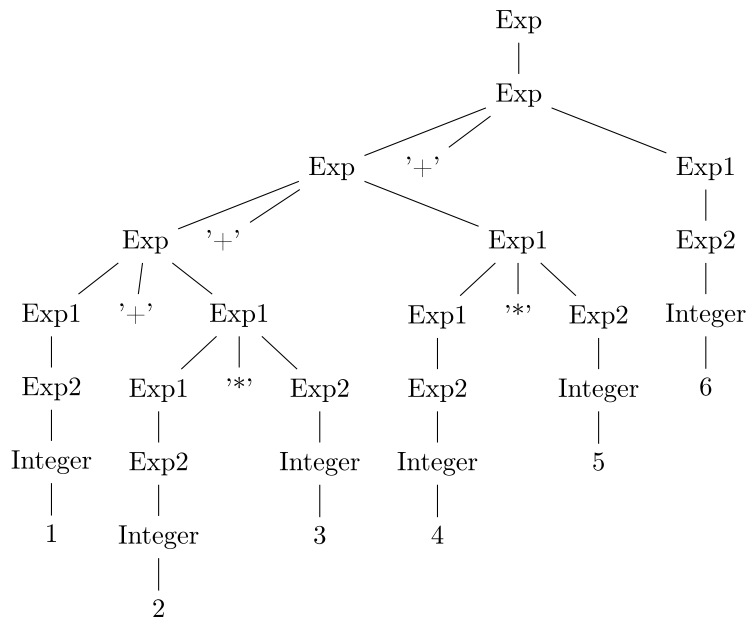
2.7.4 Expression: 1 + (2 * 3)



2.7.5 Expression: (1 + 2) * 3



2.7.6 Expression: 1 + 2 * 3 + 4 * 5 + 6


Exp
Exp
Exp '+' Exp1
Exp '+' Exp1 Exp2
Exp1 '+' Exp1 Exp1 '*' Exp2 Integer
Exp2 Exp1 '*' Exp2 Exp2 Integer 6
Integer Exp2 Integer Integer 5
1 Integer 3 4
2

3 Essay

4 Evidence of Participation

5 Conclusion

References

[BLA] Author, Title, Publisher, Year.