CPSC-354 Report

Mitchell Toney
Chapman University

October 12, 2025

Abstract

Contents

1 Introduction

2 Week by Week

2.1 Week 1: HW1

The MU puzzle is a puzzle created by Douglas Hofstadter. It consists of four rules that can be applied to a string MI.

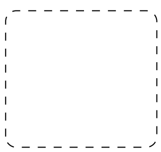
1. $xI \rightarrow xIU$
2. $Mx \rightarrow Mxx$
3. $xIIIy \rightarrow xUy$
4. $xUUy \rightarrow xy$

When first approaching this puzzle, the first strategy that came to mind was to take advantage of rule number 2 to keep duplicating the I's until there is a multiple of three, then using rules 3 and 4 to get rid of the I's and leave a remaining U.

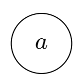
The issue with this is that $2^n \bmod 3$ will never equal 0, it infinitely cycles between equaling 1 and 2, and without being able to get rid of all the I's, which would require them being a multiple of 3, you will never be able to get MU.

Thus, the puzzle is not solvable.

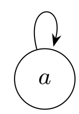
2.2 Week 2: HW2



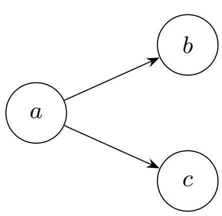
1. $A = \varnothing,\ R = \varnothing$



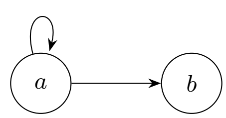
2. $A = \{a\},\ R = \varnothing$



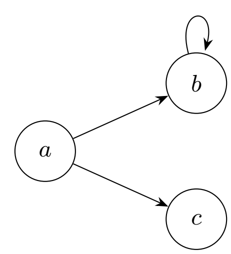
3. $A = \{a\},\ R = \{(a, a)\}$



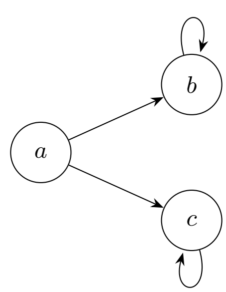
4. $A = \{a, b, c\},\ R = \{(a, b), (a, c)\}$



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



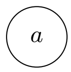
6. $A = \{a, b, c\},\ R = \{(a, b), (b, b), (a, c)\}$



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

#	Terminating	Confluent	Unique NFs
1	Yes	Yes	Yes
2	Yes	Yes	Yes
3	No	Yes	No
4	Yes	No	No
5	No	Yes	Yes
6	No	No	No
7	No	No	No

Confluent True, Terminating True, Unique NFs True



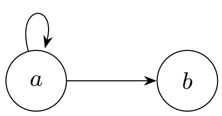
$A = \{a\},\ R = \varnothing$

Confluent True, Terminating True, Unique NFs False

impossible

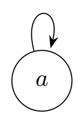
(no ARS exists)

Confluent True, Terminating False, Unique NFs True



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

Confluent True, Terminating False, Unique NFs False



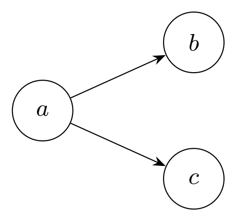
$A = \{a\},\ R = \{(a, a)\}$

Confluent False, Terminating True, Unique NFs True

impossible

(no ARS exists)

Confluent False, Terminating True, Unique NFs False



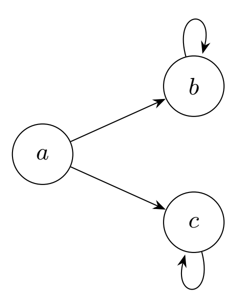
$A = \{a, b, c\},\ R = \{(a, b), (a, c)\}$

Confluent False, Terminating False, Unique NFs True

impossible

(no ARS exists)

Confluent False, Terminating False, Unique NFs False



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

2.3 Week 3: HW3

2.3.1 Exercise 5

Consider rewrite rules:

$$\begin{aligned} ab &\to ba \\ ba &\to ab \\ aa &\to \\ b &\to \end{aligned}$$

Example Reductions

Reducing `abba`:

$$\begin{aligned} abba &\to baba \quad (\text{using } ab \to ba) \\ baba &\to bbaa \quad (\text{using } ba \to ab) \\ bbaa &\to baa \quad (\text{using } b \to \varepsilon) \\ baa &\to aba \quad (\text{using } ba \to ab) \\ aba &\to baa \quad (\text{using } ab \to ba) \end{aligned}$$

There is an infinite loop between `aba` and `baa`.

Reducing `bababa`:

$$\begin{aligned} bababa &\to ababab \quad (\text{using } ba \to ab) \\ ababab &\to baabab \quad (\text{using } ab \to ba) \\ baabab &\to ababab \quad (\text{using } ba \to ab) \end{aligned}$$

This is an infinite loop between `ababab` and `baabab`.

Why the ARS is not terminating The ARS is not terminating because the rules $ab \to ba$ and $ba \to ab$ create infinite cycles. These rules allow us to swap adjacent a and b characters indefinitely, leading to non-terminating reduction sequences.

Non-equivalent strings Two strings that are not equivalent: `a` and `aa`.

The string `a` cannot be reduced further, while `aa` reduces to nothing using the rule $aa \to$. Since $\textit{nothing} \neq a$, these strings are in different equivalence classes.

Equivalence classes The equivalence relation $\leftrightarrow^*$ has infinitely many equivalence classes. Each equivalence class can be characterized by the number of a's modulo 2 and the number of b's modulo 1.

The equivalence classes are:

- $[\varepsilon]$: strings with even number of a's and no b's
- $[a]$: strings with odd number of a's and no b's
- $[b]$: strings with any number of a's and at least one b

The normal forms are: ε, a, and b.

Modifying the ARS to be terminating To make the ARS terminating without changing equivalence classes, we can remove the symmetric rules and keep only one direction:

$$\begin{aligned} ba &\to ab \\ aa &\to \varepsilon \\ b &\to \varepsilon \end{aligned}$$

This eliminates the infinite cycles while preserving the same equivalence relation.

2.3.2 Semantic question

Parity of a's: "Does this string contain an odd number of a's?"
Answer: Yes if the normal form is a, No if the normal form is ε.

Exercise 5b

Consider rewrite rules:

$$\begin{aligned} ab &\to ba \\ ba &\to ab \\ aa &\to a \\ b &\to \end{aligned}$$

Reducing `abba`:

$$\begin{aligned} abba &\to baba \\ baba &\to abba \end{aligned}$$

Reducing `bababa`:

$$\begin{aligned} bababa &\to ababab \\ ababab &\to bababa \end{aligned}$$

Why not terminating. The symmetric swaps $ab \leftrightarrow ba$ allow infinite rewriting.

Non-equivalent strings. a and ε are not equivalent.

Equivalence classes. Exactly two: $[\varepsilon]$ (no a's; all b's delete) and $[a]$ (at least one a; since $aa \sim a$). Normal forms (under a terminating orientation): ε and a.

Terminating variant.

$$\begin{aligned} ba &\to ab \\ aa &\to a \\ b &\to \end{aligned}$$

This gives a complete semantics to the ARS: it computes the invariant for any input string.

2.4 Week 4: HW4

2.4.1 HW4.1: Termination Proof for GCD

Consider the following algorithm:

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

Assume: Work over integers with Euclidean division: inputs $a \in \mathbb{Z}$, $b \in \mathbb{N}$; if $b \neq 0$ then $0 \leq a \bmod b < b$.

Model: States $A = \mathbb{Z} \times \mathbb{N}$. One step $(a, b) \to (a', b')$ is one loop iteration.

Measure: $\phi : A \to \mathbb{N},\ \phi(a, b) = b$.

Show: $(a, b) \to (a', b') \Rightarrow \phi(a', b') < \phi(a, b)$.

If $b \neq 0$, the update sets $b' = a \bmod b$ with $0 \leq b' < b$; hence $\phi(a', b') = b' < b = \phi(a, b)$.

ϕ strictly decreases in $\mathbb{N}$, so there is no infinite $\to$-chain; eventually $b = 0$ and the loop stops. Thus the algorithm terminates under the stated conditions.

2.4.2 HW4.2: Termination Proof for Merge Sort

Consider the following fragment of an implementation 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)
```

Prove that

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

is a measure function for merge_sort.

Show: For

```
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 strictly decreasing measure, hence `merge_sort` terminates.

Assume: `left`,`right`$\in \mathbb{Z}$ with $0 \leq \text{left} \leq \text{right} < |arr|$. Division for `mid` is integer. If `left < right`, then $\text{left} \leq \text{mid} < \text{right}$. `merge` makes no recursive calls.

Model: States are intervals $A = \{(l, r) \in \mathbb{Z}^2 \mid l \leq r\}$. A "step" is a recursive edge from (l, r) (with $l < r$) to each child (l, mid) and $(\text{mid} + 1, r)$.

Measure: $\phi : A \to \mathbb{N}, \quad \phi(l, r) = r - l + 1$.

Let $l < r$ and $\text{mid} = \lfloor (l + r)/2 \rfloor$ so $l \leq \text{mid} < r$.

First child: $\phi(l, \text{mid}) = \text{mid} - l + 1 \leq (r - 1) - l + 1 = r - l = \phi(l, r) - 1 < \phi(l, r)$.

Second child: $\phi(\text{mid}+1, r) = r - \text{mid} \le r - l = \phi(l, r) - 1 < \phi(l, r)$.

Every recursive edge strictly decreases ϕ in $\mathbb{N}$, which is well-founded. Thus no infinite recursion is possible; the calls bottom out at states with $\phi \in \{0, 1\}$ (i.e., `left` $\ge$ `right`), where the function returns. Therefore $\phi(\text{left}, \text{right}) = \text{right} - \text{left} + 1$ is a valid measure and `merge_sort` terminates under the stated conditions.

2.5 Week 5: HW5

2.5.1 Lambda Calculus Workout Evaluation (Corrected)

Evaluate: $(\lambda f.\lambda x.\, f(f\,x))(\lambda g.\lambda y.\, g(g(g\,y)))$

Use α-renaming to avoid capture.

$$
\begin{aligned}
(\lambda f.\lambda x.\, f(f\,x))(\lambda g.\lambda y.\, g(g(g\,y))) &\to \lambda x.\, (\lambda g.\lambda y.\, g(g(g\,y)))\big((\lambda g.\lambda y.\, g(g(g\,y)))\,x\big) \\
(\lambda g.\lambda y.\, g(g(g\,y)))\,x &\to \lambda y.\, x(x(x\,y)) \quad (\alpha\text{-rename inner } y) \\
\Rightarrow\; \lambda x.\, (\lambda g.\lambda y.\, g(g(g\,y)))\,(\lambda y.\, x(x(x\,y))) &\to \lambda x.\lambda y.\, x^9 y.
\end{aligned}
$$

Hence the normal form is $\lambda f.\lambda x.\, f^9 x$ (Church numeral 9).

2.6 Week 6: HW6

2.6.1 Fixed Point Combinator Exercise

Compute `fact 3` following the computation rules for `fix`, `let`, and `let rec`:

Given:

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

Abbreviations:

$$\mathsf{F} \stackrel{\text{def}}{=} \lambda f.\, \lambda n.\ \mathsf{if}\ n = 0\ \mathsf{then}\ 1\ \mathsf{else}\ n * f\,(n-1) \qquad\qquad \mathsf{FACT} \stackrel{\text{def}}{=} \mathsf{fix}\,\mathsf{F}$$

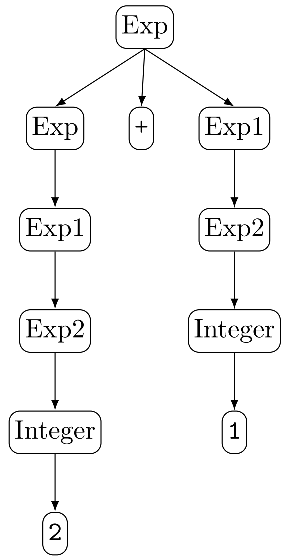
Computation:

$$
\begin{aligned}
E_0 &\xrightarrow{\text{def of let rec}} \mathsf{let\ fact} = \mathsf{fix}(\lambda f.\lambda n.\mathsf{if}\ n=0\ \mathsf{then}\ 1\ \mathsf{else}\ n*f\,(n-1))\ \mathsf{in\ fact}\,3\\
&\xrightarrow{\text{def of let}} (\lambda \mathsf{fact}.\ \mathsf{fact}\,3)\ \mathsf{fix}(\lambda f.\lambda n.\mathsf{if}\ n=0\ \mathsf{then}\ 1\ \mathsf{else}\ n*f\,(n-1))\\
&\xrightarrow{\beta} \big(\mathsf{fix}(\lambda f.\lambda n.\mathsf{if}\ n=0\ \mathsf{then}\ 1\ \mathsf{else}\ n*f\,(n-1))\big)\ 3\\
&\equiv \mathsf{FACT}\,3\\
&\xrightarrow{\text{def of fix}} (\mathsf{F\,FACT})\ 3\\
&\xrightarrow{\beta} (\lambda n.\ \mathsf{if}\ n=0\ \mathsf{then}\ 1\ \mathsf{else}\ n*\mathsf{FACT}\,(n-1))\ 3\\
&\xrightarrow{\beta} \mathsf{if}\ 3=0\ \mathsf{then}\ 1\ \mathsf{else}\ 3*\mathsf{FACT}\,(3-1)\\
&\xrightarrow{\text{def of if}} 3*\mathsf{FACT}\,(3-1)\\
&\xrightarrow{\text{arith}} 3*\mathsf{FACT}\,2\\
&\xrightarrow{\text{def of fix}} 3*(\mathsf{F\,FACT})\ 2\\
&\xrightarrow{\beta} 3*(\lambda n.\ \mathsf{if}\ n=0\ \mathsf{then}\ 1\ \mathsf{else}\ n*\mathsf{FACT}\,(n-1))\ 2\\
&\xrightarrow{\beta} 3*\mathsf{if}\ 2=0\ \mathsf{then}\ 1\ \mathsf{else}\ 2*\mathsf{FACT}\,(2-1)\\
&\xrightarrow{\text{def of if}} 3*(2*\mathsf{FACT}\,(2-1))\\
&\xrightarrow{\text{arith}} 3*(2*\mathsf{FACT}\,1)\\
&\xrightarrow{\text{def of fix}} 3*\big(2*(\mathsf{F\,FACT})\ 1\big)\\
&\xrightarrow{\beta} 3*\big(2*(\lambda n.\ \mathsf{if}\ n=0\ \mathsf{then}\ 1\ \mathsf{else}\ n*\mathsf{FACT}\,(n-1))\ 1\big)\\
&\xrightarrow{\beta} 3*\big(2*\mathsf{if}\ 1=0\ \mathsf{then}\ 1\ \mathsf{else}\ 1*\mathsf{FACT}\,(1-1)\big)\\
&\xrightarrow{\text{def of if}} 3*\big(2*(1*\mathsf{FACT}\,(1-1))\big)\\
&\xrightarrow{\text{arith}} 3*\big(2*(1*\mathsf{FACT}\,0)\big)\\
&\xrightarrow{\text{def of fix}} 3*\big(2*(1*(\mathsf{F\,FACT})\ 0)\big)\\
&\xrightarrow{\beta} 3*\big(2*(1*(\lambda n.\ \mathsf{if}\ n=0\ \mathsf{then}\ 1\ \mathsf{else}\ n*\mathsf{FACT}\,(n-1))\ 0)\big)\\
&\xrightarrow{\beta} 3*\big(2*(1*\mathsf{if}\ 0=0\ \mathsf{then}\ 1\ \mathsf{else}\ 0*\mathsf{FACT}\,(0-1))\big)\\
&\xrightarrow{\text{def of if}} 3*\big(2*(1*1)\big)\\
&\xrightarrow{\text{arith}} 3*(2*1)\ \xrightarrow{\text{arith}}\ 3*2\ \xrightarrow{\text{arith}}\ 6\ .
\end{aligned}
$$

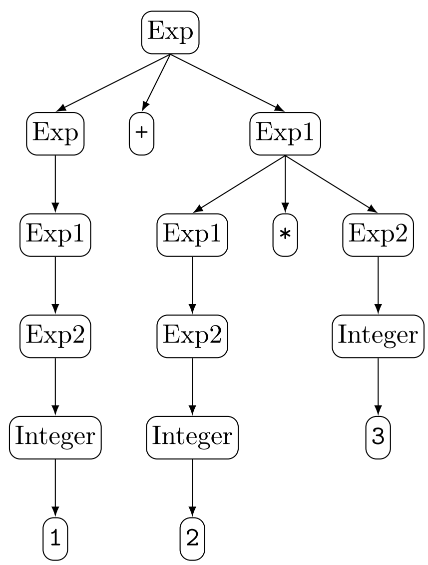
Result: `fact 3 = 6`

2.7 Week 7: Parse Trees Exercise

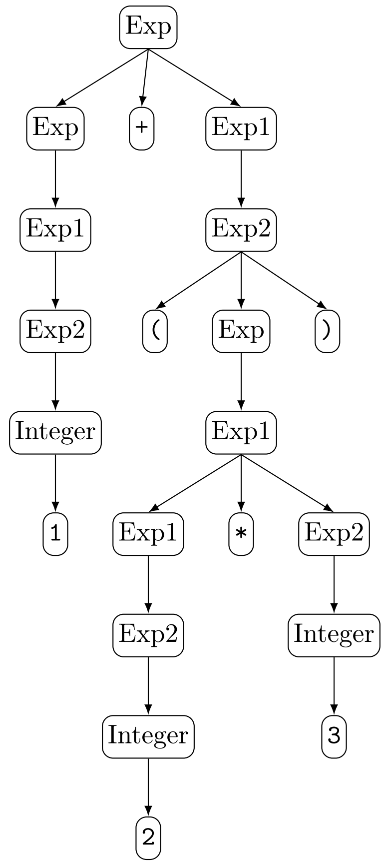
2.7.1 String: $2 + 1$



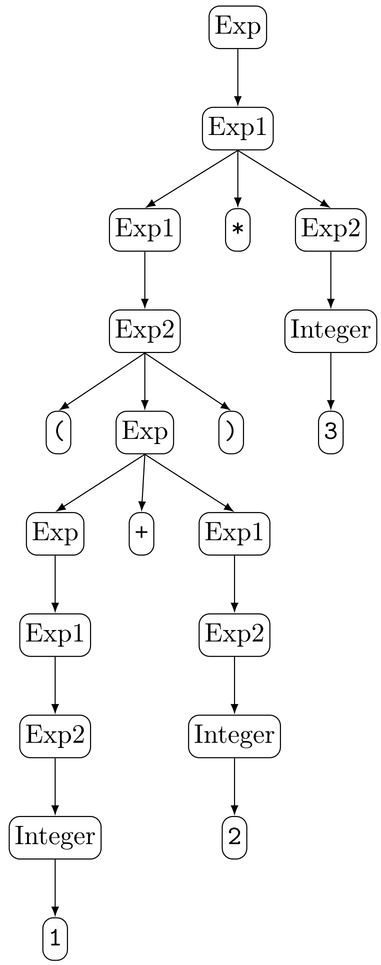
2.7.2 String: $1 + 2 * 3$



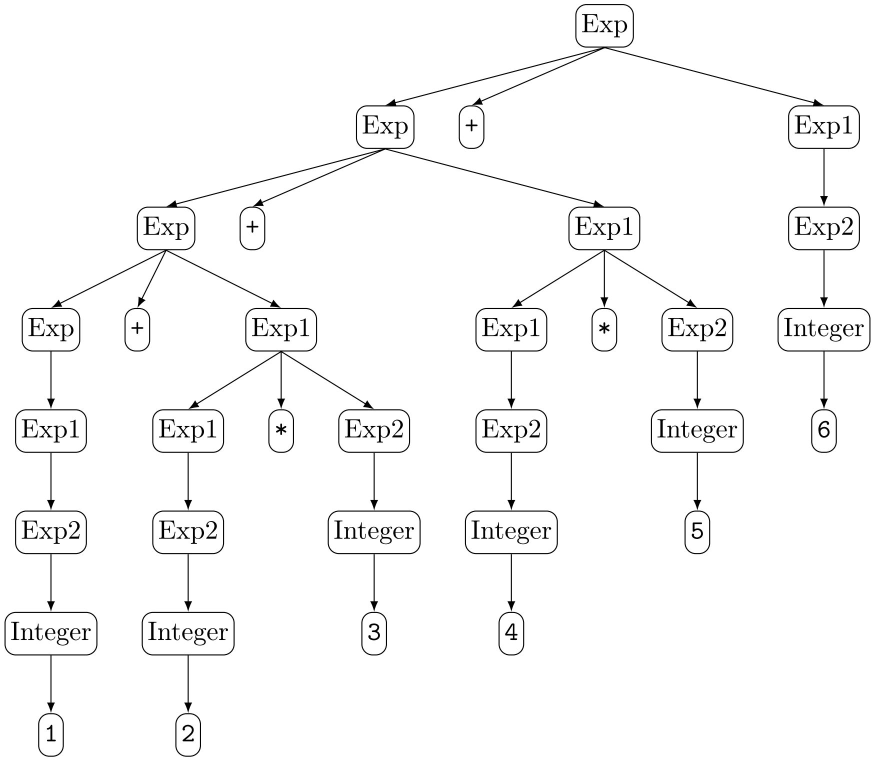
2.7.3 String: $1 + (2 * 3)$



2.7.4 String: $(1+2)*3$



2.7.5 String: $1 + 2 * 3 + 4 * 5 + 6$



3 Essay

4 Evidence of Participation

5 Conclusion

References

[BLA] Author, Title, Publisher, Year.