

# CPSC-354 Report

Rohm Tandon  
Chapman University

07/01/2024

## Abstract

This report contains assignments throughout the fall 2024 semester and is intended for the purpose of documenting my work and showing my progress in CPSC 354 - Programming Languages, taught by Jonathan Weinberg.

## Contents

<b>1</b>	<b>Introduction</b>	<b>1</b>
<b>2</b>	<b>Week by Week</b>	<b>1</b>
2.1	Week 1 . . . . .	1
2.2	Week 2 . . . . .	4
2.3	Week 3 . . . . .	6
2.4	Week 4 . . . . .	7
2.5	Week 5 . . . . .	10
2.6	Week 6 . . . . .	11
2.7	Week 7 . . . . .	12
2.8	Week 8 . . . . .	12
2.9	Week 9 . . . . .	14
2.10	Week 10 . . . . .	14
2.11	Week 11 . . . . .	17
2.12	Week 11 . . . . .	20
<b>3</b>	<b>Lessons from the Assignments</b>	<b>21</b>
<b>4</b>	<b>Conclusion</b>	<b>21</b>

## 1 Introduction

This report, prepared for CPSC 354 - Programming Languages at Chapman University, is a comprehensive account of my academic voyage over the semester. It includes a detailed compilation of my notes, homework solutions, and critical reflections on the coursework. This report serves as a bridge between the theoretical knowledge imparted in lectures and the practical skills essential for future pursuits in both graduate studies and the software industry.

## 2 Week by Week

### 2.1 Week 1

#### Notes

In week 1 we learnt about Lean as a programming language and its correlation to discrete math. We also learnt about other proof assistants. We then shifted our focus to the NNG tutorial world as you can see below.

#### Homework

Tutorial world

Level 5 / 8 : Adding zero

Active Goal

Objects:  
 $a\ b\ c : \mathbb{N}$

Goal:  
 $a + (b + 0) + (c + 0) = a + b + c$

rw [add\_zero]

Active Goal

Objects:  
 $a\ b\ c : \mathbb{N}$

Goal:  
 $a + b + (c + 0) = a + b + c$

rw [add\_zero]

Active Goal

Objects:  
 $a\ b\ c : \mathbb{N}$

Goal:  
 $a + b + c = a + b + c$

rfl

level completed! 🎉

Level 5:

Discrete math's lemmas tell us that anything added to 0 will give the result of that number itself. So;  $A+0=A$ . Using this we can bring the left hand side down to  $a+b+c$ . From here we can use the property of reflexivity to show that both sides are equal, hence solving the puzzle.

## Level 6 / 8 : Precision rewriting

Objects:

$a, b, c : \mathbb{N}$

Goal:

$$a + (b + 0) + (c + 0) = a + b + c$$

```
rw [add_zero c]
```

Active Goal

Objects:

$a, b, c : \mathbb{N}$

Goal:

$$a + (b + 0) + c = a + b + c$$

```
rw [add_zero b]
```

Active Goal

Objects:

$a, b, c : \mathbb{N}$

Goal:

$$a + b + c = a + b + c$$

```
rfl
```

level completed! 🏆

Level 6:

Level 7 / 8 : add\_succ

**Theorem** `succ_eq_add_one` : For all natural numbers  $a$ , we have  $\text{succ}(a) = a + 1$ .

Active Goal

---

**Objects:**  
 $n : \mathbb{N}$

**Goal:**  
 $\text{succ } n = n + 1$

`rw [one_eq_succ_zero]`

Active Goal

---

**Objects:**  
 $n : \mathbb{N}$

**Goal:**  
 $\text{succ } n = n + \text{succ } 0$

`rw [add_succ]`

Active Goal

---

**Objects:**  
 $n : \mathbb{N}$

**Goal:**  
 $\text{succ } n = \text{succ } (n + 0)$

`rw [add_zero]`

Level 7:

Level 8 / 8 : 2+2=4

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

```

2 rw[three_eq_succ_two]
3 rw[two_eq_succ_one]
4 rw[one_eq_succ_zero]
5 rw[succ_eq_add_one]
6 rw[one_eq_succ_zero]
7 rw[add_succ]
8 rw[add_zero]
9 rw[add_succ]
10 rw[add_succ]
11 rw[add_zero]
12 rfl

```

Level completed! 🎉

No Goals

Level 8:

## 2.2 Week 2

### Notes

In week 2 we learnt about recursion and its application in other problems such as the Towers of Hanoi game we played. We also learnt about its various benefits such as breaking down complexity of problems and being more concise.

### Homework

Addition world

Level 1 / 5 : zero\_add

**Theorem** zero\_add: For all natural numbers  $n$ , we have  $0 + n = n$ .

```

theorem zero_add (n : ℕ) : 0 + n = n := by
  1 induction n with d hd
  2 rw[add_zero]
  3 rfl
  4 rw[add_succ]
  5 rw[hd]
  6 rfl
  7 |

```

Level 1:

Level completed! 🎉

Level 2 / 5 : succ\_add

**Theorem** succ\_add: For all natural numbers  $a, b$ , we have  $\text{succ}(a) + b = \text{succ}(a + b)$ .

```

theorem succ_add (a b : ℕ) : succ a + b = succ (a + b) := by
  1 induction b with d hd
  2 rw[add_zero]
  3 rw[add_zero]
  4 rfl
  5 rw[add_succ, add_succ]
  6 rw[hd]
  7 rfl
  8 |

```

Level 2:

Level completed! 🎉

Level 3 / 5 : add\_comm (level boss)

**Theorem** add\_comm: On the set of natural numbers, addition is commutative. In other words, if  $a$  and  $b$  are arbitrary natural numbers, then  $a + b = b + a$ .

```

theorem add_comm (a b : ℕ) : a + b = b + a := by
  1 induction b with d hd
  2 rw[add_zero, zero_add]
  3 rfl
  4 rw[add_succ, succ_add, hd]
  5 rfl
  6 |

```

Level 3:

Level completed! 🎉

Level 4 / 5 : add\_assoc (associativity of addition)

**Theorem** add\_assoc: On the set of natural numbers, addition is associative. In other words, if  $a, b$  and  $c$  are arbitrary natural numbers, we have  $(a + b) + c = a + (b + c)$ .

```

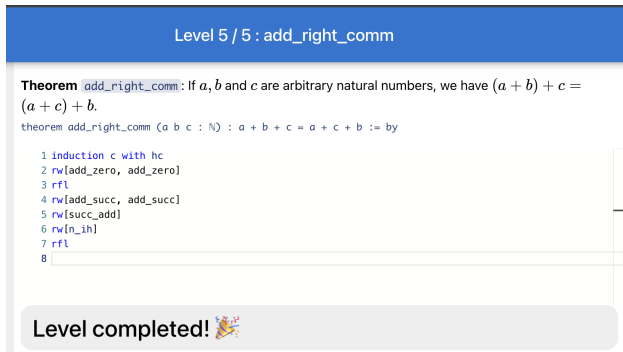
theorem add_assoc (a b c : ℕ) : a + b + c = a + (b + c) := by
  1 induction c with c hc
  2 rw[add_zero, add_zero]
  3 rfl
  4 rw[add_succ, add_succ]
  5 rw[add_succ]
  6 rw[hc]
  7 rfl
  8 |

```

Level 4:

Level completed! 🎉

Using induction on  $c$  we can initially create an easier medium to use reflexivity to solve for  $a+b$ . Then solving the other side we just use the mathematical definition of a successor function, until we can use the induction again to get the equation to the point where we can use reflexivity to prove it. This is a clear example of mathematical proofs by induction.



Level 5:

Once again this is proof by mathematical induction similar to the previous one. This time we add the zeroes and then use reflexivity for the first part of the proof. Then to prove the second part we use the successor function until bringing back the induction we used like the previous question. Then to complete the proof we use reflexivity again.

-

Discord Question: Since recursion has so many benefits and also breaks down the complexity of problems, why aren't we taught to use it as our primary method? In other words, why isn't it the first method of problem solving we're taught?

## 2.3 Week 3

### Notes

In week 3 we spoke about recursion further, and focused on our calculators in python. Eventually connecting the dots for our first Assignment that was due at the same time as Homework 4, where I used recursion in my python calculator in order to get it to function as efficiently as possible. We also discussed parsing, and derivation trees which is what we practiced in Homework 4.

### Homework

For homework 4 we did some practice on derivation trees for the strings you can see in the handwritten work below, along with the respective answers;

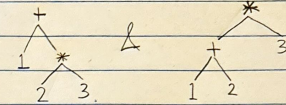
CPSC 354 - Programming Languages  
Homework-4

Q. Write out the derivation trees for the following strings:

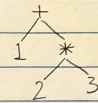
a)  $2+1$



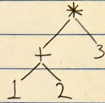
b)  $1+2\cdot3$



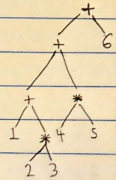
c)  $1+(2\cdot3)$



d)  $(1+2)\cdot3$



e)  $1+2\cdot3+4\cdot5+6$



Discord Question: Why do programming languages need different types of parsers, and how does this choice impact the way a computer understands the code?

## 2.4 Week 4

### Notes

In week 4 we spoke about parsing and trees further which is what we practiced in last weeks Homework. But now we delved into the more notation heavy side of it. This weeks homework contained the lean logic game that helped us put some of that into practice.

### Homework

For homework 5 we completed the Lean logic game tutorial world. I have provided the answers to the same below.

#### 1. Exhibit evidence that you're planning a party.

##### Level 1:

```
example (P : Prop) (todo_list : P) : P := by
```

##### Solution:

```
exact todo_list
```

**Level 2:**

```
example (P S : Prop) (p : P) (s : S) : P S := by
```

**Solution:**

```
exact and.intro p s
```

**Level 3:**

```
example (A I O U : Prop) (a : A) (i : I) (o : O) (u : U) : (A I) O U := by
```

**Solution:**

```
have ou := and_intro o u
have ai := and_intro a i
exact and_intro ai ou
```

**Level 4:**

```
example (P S : Prop) (vm: P S) : P := by
```

**Solution:**

```
exact vm.left
```

**Level 5:**

```
example (P Q : Prop) (h: P Q) : Q := by
```

**Solution:**

```
exact h.right
```

**Level 6:**

```
example (A I O U : Prop) (h1 : A I) (h2 : O U) : A U := by
```

**Solution:**

```
exact and_intro h1.left h2.right
```

**Level 7:**

```
example (C L : Prop) (h: (L ((L C) L) L L L)) (L L) L : C := by
```

**Solution:**



```

have a:= h.left
have b:= a.right
have c:= b.left
have d:= c.left
exact d.right

```

#### Level 8:

```

example (A C I O P S U : Prop)(h: ((P S) A)  $\neg$ I (C  $\neg$ O)  $\neg$ U) : A C P S := by

```

#### Solution:

```

have a:=h.left
have fin:=a.left
have a:=a.right
have r:=h.right
have t:=r.right
have k:= t.left
have c:=k.left
have e:= and_intro c fin
exact and_intro a e

```

#### Explanation for Solution 8:

We ultimately need A C P S, which means that we need to retrieve A, C, P, and S from the original expression h.

- (1) Seeing that A, P and S are on the left side of h-> extract h.left to get (P S) A.
- (2) Seeing that (P S) is already clearly made on the left side we can store (P S) together as 'fin' by extracting the left side.
- (3) Now from the same expression we can extract the right side to retrieve and store A as 'a'.
- (4) Now that we have A, P and S stored and easily accessible, we need C from right side of h.
- (5) After storing h.right in another expression, 'r', we take the right side of r to further narrow down the position of C.
- (6) Now we take the left side of that expression for the same purpose, store it as k.
- (7) Finally we can retrieve C by storing the k.left as 'c'.
- (8) Now to finally present our solution we can start by using and\_intro on c and fin, storing that as e.
- (9) To complete our solution we use exact and\_intro on a and e.

Discord Question: How does the choice of a parsing technique (e.g., top-down vs. bottom-up) impact the efficiency and clarity of the resulting derivation tree? In other words, when should I choose between the two approaches so that my chosen approach is significantly more beneficial than the other?

## 2.5 Week 5

### Notes

In week 5 we spoke about type theory in programming languages. Moreover, we spoke about proposition types, constructive logic, and functional programming. We also furthered our understanding of the lean logic that helped me understand some parts of the homework as well.

### Homework

For homework 6 we completed the Lean logic game on implication. I have provided the answers to the same below.

#### 1. Exhibit evidence that cake will be delivered to the party

##### Level 1:

```
example (P C: Prop) (p: P) (bakery_service : P → C) : C := by
```

##### Solution:

```
exact(bakery_service p)
```

##### Level 2:

```
example (C: Prop) : C → C := by
```

##### Solution:

```
exact h : C => h
```

##### Level 3:

```
example (I S: Prop) : I → S → S → I := by
```

##### Solution:

```
exact h : I → S => and_intro (and_right h) h.left
```

##### Level 4:

```
example (C A S: Prop) (h1 : C → A) (h2 : A → S) : C → S := by
```

##### Solution:

```
exact h : C => h2 (h1 h)
```

##### Level 5:

```
example (P Q R S T U: Prop) (p : P) (h1 : P → Q) (h2 : Q → R) (h3 : Q → T) (h4 : S → T) (h5 : T → U)
```

##### Solution:

```
exact h5 ( h3 ( h1 p))
```

**Level 6:**

```
example (C D S: Prop) (h : C → D → S) : C → D → S := by
```

**Solution:**

```
exact fun f : C => fun b : D => h <f, b>
```

**Level 7:**

```
example (C D S: Prop) (h : C → D → S) : C → D → S := by
```

**Solution:**

```
exact fun f : C → D => h f.left f.right
```

**Level 8:**

```
example (C D S : Prop) (h : (S → C) → (S → D)) : S → C → D := by
```

**Solution:**

```
exact fun s : S => and_intro (h.left s) ( h.right s)
```

**Level 9:**

```
example (R S : Prop) : R → (S → R) → (¬S → R) := by
```

**Solution:**

```
exact fun r : R => and_intro (fun s : S => r) ( fun ss: ¬S => r)
```

Discord Question: How can implication be used in functional programming, and more so, why is it not as seemingly embedded in some of the popular languages we use today?

## 2.6 Week 6

### Notes

In week 6 we spoke about Lambda Reduction and practiced a few questions to improve our understanding of the same.

### Homework

For homework 7 we discussed some theory related to capture avoiding substitution by reducing a lambda term. Then we read and discussed Church numerals.

1. **The purpose of this hw is to practice capture avoiding substitution.**

**Question 1:**

**Solution:**

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

Capture avoiding substitution is the practice of performing substitutions in lambda calculus, while ensuring that no variables are "captured" by new bindings. In other words, if we substitute a variable in an expression, we should not accidentally bind it to a lambda term that already uses that variable in a different scope.

This yields the result:  $\lambda x. ((\lambda f. \lambda x. f(f(fx)))(\lambda f. \lambda x. f(f(fx)))x)$

**Question 2:****Solution:**

Given  $\lambda m. \lambda n. mn$ , let's interpret it in terms of Church numerals.

This function takes two Church numerals,  $m$  and  $n$ .

This function implements addition for Church numerals. It combines the number of applications of  $f$  in  $m$  and  $n$ , resulting in  $m + n$ .

So this function represents the addition function in Church numerals.

Discord Question: When performing capture-avoiding substitution, we often rename bound variables to avoid conflicts. How might this affect the efficiency or complexity of the substitution operation?

## 2.7 Week 7

**Notes**

In week 7 we spoke more about beta reduction and dove into reducing expressions with Church numerals. Overall, we went over Turing completeness and the connection between that and our previous topic of Church numerals.

## 2.8 Week 8

**Notes**

In week 8 we spoke about different reduction strategies - by value and by name. We also looked at the VSCode debugger and learnt about the importance of working with a debugger. We then tried to understand the interpreter and then used our learning of the different reduction strategies in our homework.

**Homework**

For homework 8-9 we did Exercises 2-8 in <https://hackmd.io/@alexhkurz/S1R1F61yx>

2) For  $a\ b\ c\ d$ , the reduction occurs in a left-associative manner:  $a\ b\ c\ d \rightarrow (((a\ b)\ c)\ d)$ . Each application applies the leftmost term to the next in sequence, creating nested applications. For  $(a)$ , it's a single variable wrapped in parentheses, which reduces to  $a$  as there are no further applications or abstractions.

3) Capture-avoiding substitution ensures that when substituting a variable, it doesn't inadvertently capture free variables. This involves renaming bound variables to prevent conflicts, a common issue in lambda calculus interpreters. The substitute function receives three parameters: The expression in which substitution occurs,

the variable being replaced, and the expression replacing the variable. For application expressions (e.g., a b), it recursively applies substitution to both the function and argument. The Lambda abstraction is where the capture avoiding substitution comes into play. The interpreter checks for conflicts, and a fresh variable name is generated to replace the conflicting bound variable. After renaming, substitution proceeds within the lambda body with the newly renamed variable.

4) Generally, the interpreter provides the expected results when handling basic lambda expressions and straightforward applications. However, more complex expressions with multiple nested applications or ones that involve intricate substitutions may sometimes yield unexpected results if there are any implementation issues with capture-avoiding substitution or recursive application handling. Not all computations reduce to normal form. Some expressions, particularly those involving self-application lead to infinite loops because they lack a terminating condition. For other expressions designed to terminate, like Church numerals in basic operations, the interpreter generally reaches normal form as expected.

5) An example of such an expression is the "omega combinator"

```
((\x. x x) (\x. x x))
```

which results in an infinite loop of self-application. I added this to test.lc to observe this non-terminating behaviour in order to identify it as the MWE.

6) I followed the instruction as per, for the next two items.

7) The substitutions were seen at the breakpoints. A concise version of the same is below;

```
(\n.(m n))
(m n)
(m Var1)
((\f.(\x.(f (f x)))) Var1)
(\f.(\x.(f (f x))))
(f (f x))
(f (f Var3))
(\Var3.(Var2 (Var2 Var3)))
(Var2 (Var2 Var3))
(Var2 Var3)
(Var2 (Var2 Var4))
(\Var4.(Var2 (Var2 Var4)))
(Var2 (Var2 Var4))
(Var2 (Var2 Var5))
```

8) The trace is written out below;

```
13 (((\m.(\n.(m n))) (\f.(\x.(f (f x))))) (\f.(\x.(f (f (f x))))))
40 ((\m.(\n.(m n))) (\f.(\x.(f (f x)))))
40 (\m.(\n.(m n)))
54 ((\m.(\n.(m n))) (\f.(\x.(f (f x)))))
46 (\Var1.((\f.(\x.(f (f x)))) Var1))
54 ((\m.(\n.(m n))) (\f.(\x.(f (f x)))))
54 (((\m.(\n.(m n))) (\f.(\x.(f (f x))))) (\f.(\x.(f (f (f x))))))
46 ((\Var2.(\Var4.(Var2 (Var2 Var4)))) (\f.(\x.(f (f (f x))))))
40 (\Var2.(\Var4.(Var2 (Var2 Var4))))
54 ((\Var2.(\Var4.(Var2 (Var2 Var4)))) (\f.(\x.(f (f (f x))))))
46 (\Var5.((\f.(\x.(f (f (f x))))) (\f.(\x.(f (f (f x))))) Var5)))
54 ((\Var2.(\Var4.(Var2 (Var2 Var4)))) (\f.(\x.(f (f (f x)))))
54 (((\m.(\n.(m n))) (\f.(\x.(f (f x))))) (\f.(\x.(f (f (f x))))))
```

Discord Question: Capture avoiding substitution seems to be a fundamental theme to lambda calculus interpreters, but I still have problems understanding how to best implement it in Python. What other resources can I use to further understand its implementation?

## 2.9 Week 9

### Notes

In week 9 we continued to speak about the interpreter and modifying it. We also read about three languages - BLOOP, FLOOP, and GLOOP in order to further our understanding of recursion.

### Homework

For homework 10 we reflected on the previous week's homework: 8-9, as well as our group programming assignment 3. My reflection is given below.

1) I found keeping track of the trace was one of the most challenging parts of the homework. I was not completely sure of where exactly my breakpoints needed to be for exercise 7 and I had to do it multiple times to formulate a readable answer. However, this helped me understand what exactly was happening through the debugging process. 2) We took our time with the exercise and then spoke about it as a group before coming up with our key insight, realising that our evaluation strategy really matters. 3) My most interesting takeaway is that despite python being my most used language, there are still some seemingly necessary things that I am not aware of. I certainly want to master using these skills in order to be able to use them in my day to day coding. I rely on print statements more than I apparently should, and that is something I would have never expected to find out this way.

Discord Question: Many developers rely on print statements, but it seems like breakpoints and the debug console can provide more control and insight. What is the best way to make the switch over to using those, and should I completely stop relying on print statements?

## 2.10 Week 10

### Notes

In week 10 we spoke about Algorithms as Rewriting Systems (ARSs) and the homework reflects exercises regarding the same.

### Homework

For homework 11 we worked on ARSs given a list to work with. my work for the same is below.

### Question:

For each of the 8 possible combinations of the properties confluent ( $C$ ), terminating ( $T$ ), and having unique normal forms ( $UNF$ ), determine whether such an ARS exists. Provide an example if possible, or explain why it cannot exist. For the examples, draw the ARS diagrams.

### Answer:

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

*Example:*

Consider the ARS with elements  $S = \{a, b, c\}$  and reduction rules:

$$a \rightarrow b, \quad b \rightarrow c$$

*Diagram:*

$$a \longrightarrow b \longrightarrow c$$

*Analysis:*

- **Terminating:** Yes, all reduction sequences eventually reach  $c$ , which is a normal form. - **Confluent:** Yes, there are no diverging reduction paths. - **Unique Normal Forms:** Yes, every element reduces to  $c$ , the unique normal form.

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

*Explanation:*

This combination is **impossible**. In a confluent and terminating ARS, every element reduces to a unique normal form. Therefore, unique normal forms must exist.

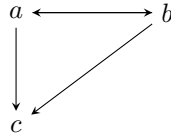
**3. Confluent: True, Terminating: False, Unique Normal Forms: True**

*Example:*

Consider the ARS  $S = \{a, b, c\}$  and rules:

$$a \rightarrow b, \quad b \rightarrow a, \quad a \rightarrow c, \quad b \rightarrow c$$

*Diagram:*



*Analysis:*

- **Terminating:** No there is an infinite loop between  $a$  and  $b$  - **Confluent:** Yes, both  $a$  and  $b$  reduce to  $c$ , and any divergent paths converge at  $c$ . - **Unique Normal Forms:** Yes, all elements reduce to  $c$ , the unique normal form.

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

*Example:*

Consider the ARS with elements  $S = \{a, b\}$  and rules:

$$a \rightarrow b, \quad b \rightarrow a$$

*Diagram:*

$$a \longleftrightarrow b$$

*Analysis:*

- **Terminating:** No, there is an infinite loop between  $a$  and  $b$ . - **Confluent:** Yes, there are no diverging paths; the reductions cycle between  $a$  and  $b$ . - **Unique Normal Forms:** No, neither  $a$  nor  $b$  is a normal form, and there are no normal forms in  $S$ .

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

*Explanation:*

This combination is **impossible**. If an ARS has unique normal forms, it must be confluent.

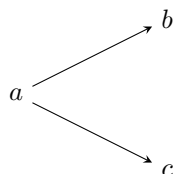
**6. Confluent: False, Terminating: True, Unique Normal Forms: False**

*Example:*

Consider the ARS  $S = \{a, b, c\}$  and rules:

$$a \rightarrow b, \quad a \rightarrow c$$

*Diagram:*



*Analysis:*

- **Terminating:** Yes, reductions terminate at  $b$  or  $c$ , which are normal forms. - **Confluent:** No, from  $a$ , we can reach two different normal forms,  $b$  and  $c$ , which are not joinable. - **Unique Normal Forms:** No, the element  $a$  has two distinct normal forms.

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

*Explanation:*

This combination is **impossible**. Unique normal forms imply that the ARS is confluent and normalising.

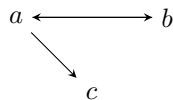
**8. Confluent: False, Terminating: False, Unique Normal Forms: False**

*Example:*

Consider the ARS with elements  $S = \{a, b, c\}$  and rules:

$$a \rightarrow b, \quad b \rightarrow a, \quad a \rightarrow c$$

*Diagram:*



*Analysis:*

- **Terminating:** No, there is an infinite loop between  $a$  and  $b$ . - **Confluent:** No, from  $a$ , we can either loop indefinitely between  $a$  and  $b$ , or reduce to  $c$ . - **Unique Normal Forms:** No,  $c$  is a normal form reachable from  $a$ , but  $b$  and  $a$  do not reduce to  $c$  if we stay in the loop.

Discord Question: In what real-world applications would rewriting systems provided unique advantages over other types of algorithms?



## 2.11 Week 11

### Notes

In week 11 we covered some string rewriting exercises while talking about operational and denotational semantics.

### Homework

For homework 12 we put our methodical learning into practice in order to learn the method of decidability via rewriting to normal form and the method of invariants. Questions 1- 5b were assigned for this.

#### Exercise 1:

The rewrite rule is:

`ba -> ab`

- a) Why does the ARS terminate?
- b) What is the result of a computation (the normal form)?
- c) Show that the result is unique (the ARS is confluent).
- d) What specification does this algorithm implement?

#### Answer 1:

- a) Each rewrite `ba->ab` reduces disorder. The string is finite, so rewriting stops when no `ba` remains.
- b) The normal form is the string in which all occurrences of `b` appear after all occurrences of `a`.
- c) The ARS is confluent if for any string, all rewriting sequences eventually lead to the same final result (normal form). Here, the system is confluent because no matter how the rule `ba->ab` is applied the final result will always be the lexicographically ordered string because the rewrite rule consistently moves `a`'s to the left and `b`'s to the right.
- d) This algorithm implements sorting, placing all `a`'s before all `b`'s.

#### Exercise 2:

Rewrite rules are

`aa -> a`

`bb -> a`

`ab -> b`

`ba -> b`

- a) Why does the ARS terminate?
- b) What are the normal forms?
- c) Is there a string `s` that reduces to both `a` and `b`?
- d) Show that the ARS is confluent.

The next questions have all essentially the same answer:

- e) Replacing `->` by `=`, which words become equal?
- f) Can you describe the equality `=` without making reference to the four rules above?
- g) Can you repeat the last item using modular arithmetic?
- h) Which specification does the algorithm implement?

#### Answer 2:

- a) The rewriting rules reduce the string's length, as every rewrite replaces two characters with one, eventually causing it to terminate.
- b) The normal forms are `a` and `b` as no rewrite rules apply to a single character.
- c) No, this does not exist.
- d) The ARS is confluent because, regardless of the rewrite order, all reduction paths lead to the same unique normal form.
- e) Any two strings reduce to the same normal form (`a` or `b`), so they are equal under this equivalence

relation.

- f) Two strings are equivalent if their counts of a and b modulo 2 are the same.
- g) A string's equivalence is determined by the count of a mod 2 and the count of b mod 2. Strings with the same parities of a and b are equal.
- h) It implements a parity check on the counts of a and b, reducing strings to one of two equivalence classes: a or b.

### Exercise 3:

Rewrite rules are

```
aa -> a
bb -> b
ba -> ab
ab -> ba
```

- a) Why does the ARS not terminate?
- b) What are the normal forms?
- c) Modify the ARS so that it is terminating, has unique normal forms (and still the same equivalence relation).
- d) Describe the specification implemented by the ARS.

### Answer 3:

- a) The rules  $ba \rightarrow ab$  and  $ab \rightarrow ba$  create an infinite loop because they allow back-and-forth rewriting without reducing the length or modifying the structure of the string.
- b) The ARS does not have normal forms because it can endlessly rewrite  $ab$  and  $ba$  without reaching a stable state.
- c) The modified rules would be;  

```
aa -> a
bb -> b
ba -> ab
```
- d) The ARS implements lexicographical sorting of a and b, while reducing sequences of identical characters to a single character. It achieves the equivalence relation where strings with the same counts of a and b reduce to the same normal form.

### Exercise 4:

Rewrite rules are

```
ab -> ba
ba -> ab
```

Same questions as above. (This is a variation of Exse 1.)

### Answer 4:

- a) The rules  $ba \rightarrow ab$  and  $ab \rightarrow ba$  create an infinite loop because they allow back-and-forth rewriting without reducing the length or modifying the structure of the string.
- b) There are no normal forms because the ARS does not terminate, due to the same reason.
- c) The modified rules would be;  

```
ab -> ba
```
- d) The ARS implements a lexicographical ordering of the string by sorting  $ab$  to  $ba$ . It ensures that all strings end in the lexicographically smaller arrangement of a and b.

### Exercise 5:

Consider the rewrite rules

```
ab -> ba
ba -> ab
```

- aa ->  
b ->
- Reduce some example strings such as abba and bababa.
  - Why is the ARS not terminating?
  - How many equivalence classes does \* have? Can you describe them in a nice way? What are the normal forms?  
[Hint: It may be easier to first answer the next question.]
  - Can you change the rules so that the ARS becomes terminating without changing its equivalence classes?
  - Write down a question or two about strings that can be answered using the ARS. Think about whether this amounts to giving a semantics to the ARS.  
[Hint: The best answers are likely to involve a complete invariant.]

#### Answer 5:

- They both loop indefinitely as you can see; abba->ab -> ba -> ab..., bababa->bab -> aba -> bab...
- The rules ba -> ab and ab -> ba create an infinite loop because they allow back-and-forth rewriting without reducing the length or modifying the structure of the string.
- There are 4 equivalence classes based on these modulo counts:  
(0, 0) - Even count of both a and b.  
(1, 0) - Odd a, even b.  
(0, 1) - Even a, odd b.  
(1, 1) - Odd a and b.  
The normal forms are strings reduced to either the empty string (if aa -> and b -> apply fully) or a single character representing the final parity:  
a for (1, 0)  
b for (0, 1)  
ab or ba for (1, 1)  
Empty string for (0, 0).
- Yes, firstly I would remove the ab->ba and ba->ab rules and use;  
aa ->  
b ->  
ab ->
- Q1. Does the string have an odd or even number of a's and b's?  
Q2. What is the equivalence class of a given string?

#### Exercise 5b:

As Exse 5, but change aa -> to aa -> a.

#### Answer 5b:

- They both loop indefinitely as you can see; abba -> ab -> ba -> ab..., bababa->bab -> aba -> bab...
- The rules ba -> ab and ab -> ba create an infinite loop because they allow back-and-forth rewriting without reducing the length or modifying the structure of the string.
- The equivalence classes remain the same as in Exse 5, as the new rule does not affect the equivalence relation.  
With the modified rule aa -> a, the normal forms are:  
a for (1, 0)  
b for (0, 1)  
ab or ba for (1, 1) (depending on how the cycle is resolved).  
Empty string for (0, 0).
- Yes, firstly I would remove the ab->ba and ba->ab rules and use;

- ```
aa -> a
b ->
```
- e) Q1. Does the string have an odd or even count of as or bs?  
 Q2. What is the smallest lexicographical representative of the string's equivalence class?

Discord Question: Are rules like  $ab \rightarrow ba$  and  $ba \rightarrow ab$  that create infinite loops ever useful, or should they always be avoided? So far most of our work with them has been related to replacing them to create terminating ARSs.

## 2.12 Week 11

### Notes

In week 12 we spoke about the fixed point combinator, discussed differences between `let` and `let rec`, and tested the functionality by taking the example of the factorial.

### Homework

For homework 13 we put the same example into practice and wrote out how it would be done.

Compute `fact 3`

#### GENERAL DEFINITION

- ```
- (let f = e1 in e2) = (\f.e2) e1          # def of let
- (let rec f = e, in e2) = fix(\fe1) e2    # def of let rec
- (fix F) = f(fix f)                      # def of fix
```

#### ABBREVIATION

```
F = (\fact. \n. if n=0 then 1 else n*fact (n-1))
```

#### COMPUTATION

```
let rec fact = \n. if n=0 then 1 else n* fact(n-1) in fact 3

= (let fact = (fix (\fact. \n. if n=0 then 1 else n* fact(n-1))) in fact 3)

= (let fact = (fix F) in fact 3)                                     #using def of fix

= (\fact. fact 3)(fix F)                                           #using def of let

= (fix F) 3                                                         #using beta reduction

= F(fix F) 3                                                        #computation of fix

= (\fact. \n. if n=0 then 1 else n* fact(n-1)) (fix F) 3          #using def of F

= (if 3=0 then 1 else 3 * (fix F) 2)                                #using beta reduction

= 3 * ((fix F) 2)

= etc
```

Discord Question: What happens if you attempt to use fix with a non-terminating function? Will it just lack efficiency, or would it not work at all?

### **3 Lessons from the Assignments**

### **4 Conclusion**

### **References**

[BLA] Author, [Title](#), Publisher, Year.