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

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:

```
example (a b c : ) : a + (b + 0) + (c + 0) = a + b + c := by
```

Solution:

```
rw [add_zero]
rw [add_zero]
rfl
```

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:

```
example (a b c : ) : a + (b + 0) + (c + 0) = a + b + c := by
```

Solution:

```
rw [add_zero c]
rw [add_zero b]
rfl
```

Level 7:

```
theorem succ_eq_add_one n : succ n = n + 1 := by
```

Solution:

```
rw [one_eq_succ_zero]
rw [add_succ]
rw [add_zero]
rfl
```

Level 8:

```
example : (2 : ) + 2 = 4 := by
```

Solution:

```
rw[four_eq_succ_three]
rw[three_eq_succ_two]
rw[two_eq_succ_one]
rw[one_eq_succ_zero]
rw[succ_eq_add_one]
rw[one_eq_succ_zero]
rw[add_succ]
rw[add_zero]
rw[add_succ]
rw[add_succ]
rw[add_zero]
rfl
```

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:

```
theorem zero_add (n : ) : 0 + n = n := by
```

Solution:

```
induction n with d hd
rw[add_zero]
rfl
rw[add_succ]
rw[hd]
rfl
```

Level 2:

```
theorem succ_add (a b : ) : succ a + b = succ (a + b) := by
```

Solution:

```
induction b with d hd
rw[add_zero]
rw[add_zero]
rfl
rw[add_succ, add_succ]
rw[hd]
rfl
```

Level 3:

```
theorem add_comm (a b : ) : a + b = b + a := by
```

Solution:

```
induction b with d hd
rw[add_zero, zero_add]
rfl
rw[add_succ, succ_add, hd]
rfl
```

Level 4:

```
theorem add_assoc (a b c : ) : a + b + c = a + (b + c) := by
```

Solution:

```
induction c with c hc
rw[add_zero, add_zero]
rfl
rw[add_succ, add_succ]
```

```
rw[add_succ]
rw[hc]
rfl
```

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 definiton 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 prrofs by induction.

Level 5:

```
theorem add_right_comm (a b c : ) : a + b + c = a + c + b := by
```

Solution:

```
induction c with hc
rw[add_zero, add_zero]
rfl
rw[add_succ, add_succ]
rw[succ_add]
rw[n_ih]
rfl
```

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?

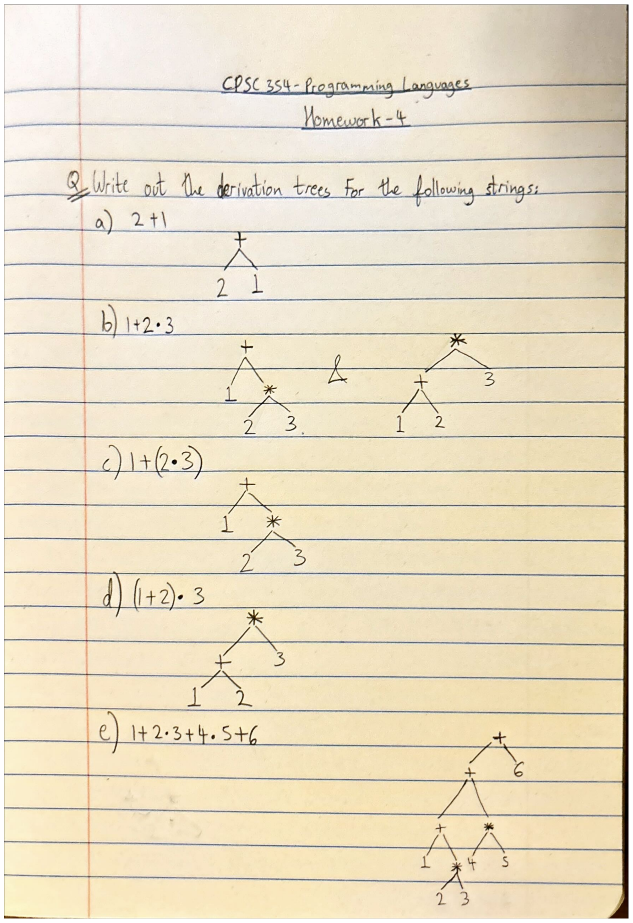
2.3 Week 3

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+2·3

&

c) 1+(2·3)

d) (1+2)·3

e) 1+2·3+4·5+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)  ¬I  (C  ¬O)  ¬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) i 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:

```
- (\n.(\f.\x.f(fx))n)(\f.\x.f(f(fx)))
- (\f.\x.f(fx))(\f.\x.f(f(fx)))
- \x.((\f.\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: \x.((\f.\x.f(f(fx)))((\f.\x.f(f(fx)))x))
```

Question 2:

Solution:

```
Given \m.\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/S1R1F6$_1yx$

2) For a b c d, the reduction occurs in a left-associative manner: a b c d → (((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 exctly 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 intersting 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:

$$a \to b, \quad b \to c$$

Diagram:

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

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:

$$a \to b, \quad b \to a, \quad a \to c, \quad b \to c$$

Diagram:

$$\begin{tikzcd} a \arrow[r, leftrightarrow] \arrow[d] & b \arrow[dl] \\ c & \end{tikzcd}$$

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:

$$a \to b, \quad b \to a$$

Diagram:

$$\begin{tikzcd} a \arrow[r, leftrightarrow] & b \end{tikzcd}$$

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:

$$a \to b, \quad a \to c$$

Diagram:

$$\begin{tikzcd}
 & b \\
a \arrow[ru] \arrow[rd] & \\
 & c
\end{tikzcd}$$

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:

$$a \to b, \quad b \to a, \quad a \to c$$

Diagram:

$$\begin{tikzcd}
a \arrow[rr, leftrightarrow] \arrow[rd] & & b \\
 & c &
\end{tikzcd}$$

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 -> 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.
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 -> 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.
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 ->
a) Reduce some example strings such as abba and bababa.
b) Why is the ARS not terminating?
c) 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.]
d) Can you change the rules so that the ARS becomes terminating without changing its equivalence
   classes?
e) 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:

```
a) They both loop indefinitely as you can see; abba->ab -> ba -> ab..., bababa->bab -> aba -> bab...
b) 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.
c) 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).
d) Yes, firstly I would remove the ab->ba and ba->ab rules and use;
   aa ->
   b ->
   ab ->
e) 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:

```
a) They both loop indefinitely as you can see; abba -> ab -> ba -> ab..., bababa->bab -> aba -> bab...
b) 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.
c) 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).
d) 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 -¿ ba and ba -¿ 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 12

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 2) = (\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 Group Assignments and Project

Working on the group assignments and projects for this course provided me with a deeper understanding of programming language concepts, the importance of testing, and the value of collaboration in solving complex problems.

My primary contribution in Milestone 1, where we developed a basic interpreter for lambda calculus and arithmetic, involved writing and adding test cases for our interpreter and ensuring they ran correctly. Here, I designed and executed tests for operations such as addition, subtraction, and nested expressions. This task emphasized the importance of exhaustive testing in validating correct order of precedence and lazy evaluation. For example, ensuring expressions like ..x y behaved correctly revealed how subtle changes in evaluation strategies could break expected behavior.

In Milestone 2, I extended my work on test cases to verify new features, including conditional expressions (if-then-else), recursive definitions (let rec), and the fix combinator. The lecture really helped with understanding this concept, and I remember the factorial example we took later as well which helped me understand the role and implementation of fic and let rec. This milestone introduced challenges in ensuring the interpreter could handle recursion effectively. I tested edge cases for recursive functions such as factorial, observing how the fix combinator guarantees the resolution of self-referential calls. This deepened my understanding of recursion and its formal implementation, reinforcing its role in functional programming languages.

While test cases were my primary focus, I also contributed to debugging and understanding the overall interpreter implementation. In particular, I collaborated with my teammates to identify and resolve issues with precedence and parenthetical expressions during Milestone 2. This experience highlighted the need for careful grammar design in ambiguous contexts. With the detail on different rules I realized how small modifications to grammar could simplify or complicate the overall evaluation process.

For Milestone 3, I supported the integration of sequencing (;) and list operations (hd, tl, cons, nil). Adding test cases for sequencing proved particularly insightful, as it involved ensuring that expressions like 1;2;3 returned the correct order of evaluations. Testing lists required validating operations like destructuring and ensuring correctness of head (hd) and tail (tl) behavior. Through this, I gained exposure to language constructs for data structures and their formal semantics.

One particularly valuable example was during Milestone 3, where we implemented sequencing and list operations as mentioned above. While testing these features, I observed how sequencing allows multiple expressions to be evaluated in a specified order, which is fundamental for combining computations in many programming languages. This simple addition mirrored concepts seen in real-world programming, where order of operations can determine program behavior. Similarly, implementing list operations highlighted the elegance of functional programming: complex structures like lists could be reduced to fundamental operations like hd and tl, which act as building blocks for more advanced constructs. These examples reinforced how seemingly abstract theoretical concepts, such as expression evaluation and data structure manipulation, directly translate into practical tools for designing powerful and expressive programming languages.

The milestone projects also provided a deeper appreciation for the theory behind programming languages, particularly concepts from lambda calculus and fixed-point combinators. Implementing features like let rec and fix during Milestone 2 helped me see how recursion is grounded in formal theory. Understanding that the fix combinator is a way to express self-referential definitions allowed me to connect theoretical ideas to practical implementations of recursion. Similarly, the interpreter's handling of lazy evaluation in Milestone 1 highlighted the importance of evaluation strategies, such as call-by-name and call-by-value, which are central to many programming paradigms. These theoretical underpinnings gave me a more rigorous understanding of how programming languages manage computation and function calls.

The projects as a whole reinforced several key lessons:

- Testing is critical: Writing robust test cases not only verifies correctness but also exposes subtle errors that may not appear during manual testing.

- Recursion and fixed points: Implementing and testing recursive features deepened my understanding of concepts like the fix combinator and their significance in lambda calculus.

- Grammar design and precedence: I learned the importance of carefully designing ambiguous grammars to avoid errors in parsing and evaluation.

- Collaboration: Group work allowed us to combine our individual strengths. By focusing on testing, I was able to contribute meaningfully while also learning from others who worked on grammar rules or interpreter implementation.

Overall, the projects demonstrated how theoretical concepts like lambda calculus, recursion, and evaluation strategies translate into practical programming language implementations. These lessons will continue to inform my approach to designing, testing, and debugging systems in the future.

4 Conclusion

This course offered a deep dive into the theoretical and practical aspects of programming languages, bridging abstract mathematical concepts with real-world implementation challenges. Through lectures, group projects, and homework assignments, I developed a clearer understanding of core principles like lambda calculus, recursion, evaluation strategies, and rewriting systems.

The class showed how programming languages are built on a solid foundation of mathematics and logic. Beyond the technical skills I gained—such as debugging, testing, and interpreter design—it also highlighted the importance of understanding semantics and formal reasoning when implementing systems.

I found the discussion of fixed-point combinators particularly insightful. Seeing how recursion, a foundational tool in programming, can be implemented formally using constructs like fix was insightful. Similarly, exploring ambiguous grammars and the role of precedence made me realise the importance of formal syntax design in programming languages.

I also spent time working with the Lean Games Server, which was both an enjoyable and educational experience. The interactive nature of the server helped me practice and learn Lean's syntax in a hands-on way. By solving puzzles and writing proofs, I gained a clearer understanding of how Lean formalizes logic and mathematics. However, I found the syntax and rules of Lean to be quite confusing at times, especially when it came to applying tactics in the correct order or understanding error messages. Despite the occasional challenges, the Lean Games Server gave me a solid foundation in formal reasoning and proof construction, which reinforced many of the theoretical concepts discussed in class.

The projects also revealed the value of careful collaboration and problem-solving in a team setting. My work on test cases complemented the contributions of others working on grammar rules or interpreter functionality, and our discussions often helped clarify complex concepts. This experience taught me to approach programming challenges systematically, breaking problems into smaller tasks while keeping the broader system in mind.

The group projects, in particular, allowed me to see how these theoretical ideas can be implemented. Writing test cases for grammar rules, recursive functions, and list operations revealed the importance of systematic testing in validating both correctness and edge cases. For instance, handling expressions like let rec or ensuring sequencing operations evaluated in the correct order reinforced the practical challenges of implementing interpreters for functional languages.

In the future, I believe the knowledge from this course will help me approach programming problems with a more analytical mindset. Whether designing a new feature, debugging a system, or analyzing the correctness of a program, I will carry forward the lessons learned from this class.

Overall, this course has provided a strong foundation in programming languages, both theoretically and practically, while encouraging critical thinking, collaboration, and precision—skills that are invaluable in the wider world of software engineering.