CPSC-354 Report

Annika Brown
Chapman University

December 15, 2025

Abstract

This a report that summarizes my work done in CPSC 354 during Fall 2025. There is weekly homework, an essay, and evidence of participation.

Contents

1 Introduction

In this course we covered topics such as the MU Puzzle, rewriting, Lean Logic, recursion and many other concepts of programming languages. This is a collection of my notes, homework solutions, and critical reflections on the content of the course.

2 Week by Week

2.1 Week 1

2.1.1 MU Puzzle

It is impossible to solve the MU puzzle. The only way to change the amount of I's is to double them with rule 2, or subtract 3 with rule 3. In order to get rid of all the I's you would have to have them in groups of 3. A power of x is divisible by 3 only if x is a multiple of 3, because the prime factorization of a number must include the prime number 3 for the number to be divisible by 3. So, you would have to get an original group divisible by 3, which is impossible.

2.2 Week 2

2.2.1 Rewriting

1. A={}
Empty ARS
Terminating: Yes
Confluent: Yes
Has Unique Normal Forms: Yes

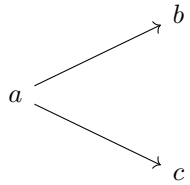
2. A={a} and R={}

Terminating: Yes
Confluent: Yes
Has Unique Normal Forms: Yes

3. A={a} and R={(a,a)}

Terminating: No
Confluent: Yes
Has Unique Normal Forms: No

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



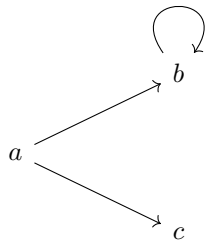
Terminating: Yes
Confluent: No
Has Unique Normal Forms: No

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

Terminating: No

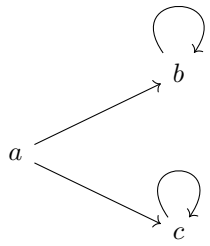
Confluent: Yes
Has Unique Normal Forms: Yes

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

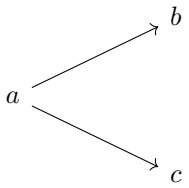
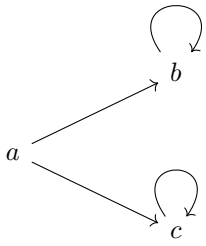


Terminating: No
Confluent: No
Has Unique Normal Forms: No

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



Terminating: No
Confluent: No
Has Unique Normal Forms: No

confluent	terminating	has unique normal forms	example
True	True	True	a
True	True	False	IMPOSSIBLE
True	False	True	
True	False	False	
False	True	True	IMPOSSIBLE
False	True	False	
False	False	True	IMPOSSIBLE
False	False	False	

2.3 Week 3

2.3.1 Homework

Exercise 5

ab → ba
ba → ab
aa →
b →

abba → aba → aa →
bababa → ababa → aaba → aaa → a

The ARS isn't terminating because it has the loop ab → ba, ba → ab.

In the above example, abba → , bababa → a, which makes them non-equivalent. There are infinitely many non-equivalent strings, because all strings with an even number of a's are only equivalent to each other, and all strings with an odd number of a's are only equivalent to each other.

There are 2 equivalence classes, all strings with an odd number of a's that have normal form a, and all strings with an even number of a's that have normal form of an empty string.

By getting rid of the first 2 rules, the ARS is terminating, with the same equivalence classes. There is no longer a loop, and you can still get rid of the same letters.

How would you have to modify the ARS so that there is only one equivalence class?

Exercise 5b

ab → ba
ba → ab
aa → a
b →

abba → aba → aa → a
bababa → ababa → aaba → aaa → aa → a

The ARS isn't terminating because it has the loop ab → ba, ba → ab.

bbb and aba are not equivalent. There are infinitely many non-equivalent strings, any string that has all b's is not equivalent to any string that has at least one a in it.

There are 2 equivalence classes, all strings that have no a's have the normal form of an empty string. All strings that have at least one a have the normal form a.

By getting rid of the first 2 rules, the ARS is terminating, with the same equivalence classes. There is no longer a loop, and you can still get rid of the same letters.

What would happen if aa → a was changed to aaa → a?

2.4 Week 4

2.4.1 Homework

4.1

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

This algorithm always terminates under the condition that $a, b \in \mathbb{N}$.

We can define this ARS as: $A = (a, b)|a, b \in \mathbb{N}$, with transition, (a, b) → (b, a mod b).

A measure function is: $\phi(a, b) = b$.

Since a → b every iteration, if b is decreasing, so is a. In the transition, b → a mod b, a mod b will always be less than b, so, $\phi(a, b) > \phi$(b, a mod b). Since this algorithm has a measure function, it terminates.

4.2

```
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 $\phi(left, right) = right - left + 1$ is a measure function.

For recursive calls, $left \leq mid < right$.

For the left side recursive call:
merge sort(arr, left, mid) = mid - left + 1
right > mid, so, right - left + 1 > mid - left + 1

$\phi(left, right) > \phi(left, mid)$
This call terminates

For the right side recursive call:
merge sort(arr, mid + 1, right) = right - (mid + 1) + 1
$mid \geq left$, so, mid + 1 > left
right - left + 1 > right - (mid + 1) + 1
$\phi(left, right) > \phi(mid + 1, right)$
This call terminates

Since both calls terminate, $\phi(left, right) = right - left + 1$ is a measure function for merge sort.

2.5 Week 5

2.5.1 Homework

(λf.λx.f(f(x))) (λf.λx.(f(f(f x))))

alpha rule:
(λg.λy.g(g(y))) (λf.λx.(f(f(f x))))

beta rule:
(λy.(λf.λx.(f(f(f x)))) ((λf.λx.(f(f(f x))))(y)))

beta rule:
(λy.(λf.λx.(f(f(f x)))) (λx.(y(y(y x)))))

alpha rule:
(λy.(λf.λz.(f(f(f z)))) (λx.(y(y(y x)))))

beta rule:
(λy.(λz.((λx.(y(y(y x))))((λx.(y(y(y x))))((λx.(y(y(y x))))z)))))

beta rule:
(λy.(λz.((λx.(y(y(y x))))((λx.(y(y(y x))))(y(y(y z)))))))

beta rule:
(λy.(λz.((λx.(y(y(y x))))(y(y(y (y(y(y z))))))))))

beta rule:
(λy.(λz.(y(y(y(y(y(y(y(y z))))))))))

2.6 Week 6

2.6.1 Homework

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

-> <def of let rec>
let fact = (fix (λfact. λn. if n=0 then 1 else n * fact (n-1))) in fact 3

-> <def of let>
(λfact. fact 3) (fix (λfact. λn. if n=0 then 1 else n * fact (n-1)))

-> <beta rule: substitute fix F>
(fix (λfact. λn. if n=0 then 1 else n * fact (n-1))) 3

-> <def of fix>
((λfact. λn. if n=0 then 1 else n * fact (n-1)) (fix (λfact. λn. if n=0 then 1 else n * fact (n-1)))) 3

-> <beta rule: substitute fix F>
(λn. if n=0 then 1 else n * (fix (λfact. λn. if n=0 then 1 else n * fact (n-1))) (n-1)) 3

-> <beta rule: substitute 3>
if 3=0 then 1 else 3 * (fix (λfact. λn. if n=0 then 1 else n * fact (n-1))) (3-1)

-> <def of if>
3 * (fix (λfact. λn. if n=0 then 1 else n * fact (n-1))) 2

-> <def of fix>
3 * ((λfact. λn. if n=0 then 1 else n * fact (n-1)) (fix (λfact. λn. if n=0 then 1 else n * fact (n-1)))) 2

-> <beta rule>
3 * (λn. if n=0 then 1 else n * (fix (λfact. λn. if n=0 then 1 else n * fact (n-1))) (n-1)) 2

-> <beta rule>
3 * (if 2=0 then 1 else 2 * (fix (λfact. λn. if n=0 then 1 else n * fact (n-1))) (2-1))

-> <def of if>
3 * (2 * (fix (λfact. λn. if n=0 then 1 else n * fact (n-1))) 1)

-> <def of fix>
3 * (2 * ((λfact. λn. if n=0 then 1 else n * fact (n-1)) (fix (λfact. λn. if n=0 then 1 else n * fact (n-1)))) 1)

-> <beta rule>
3 * (2 * (λn. if n=0 then 1 else n * (fix (λfact. λn. if n=0 then 1 else n * fact (n-1)))(n-1)) 1)

-> <beta rule>
3 * (2 * (if 1=0 then 1 else 1 * (fix (λfact. λn. if n=0 then 1 else n * fact (n-1)))(1-1)))

-> <def of if>
3 * (2 * (1 * (fix (λfact. λn. if n=0 then 1 else n * fact (n-1))) 0))

-> <def of fix>
3 * (2 * (1 * ((λfact. λn. if n=0 then 1 else n * fact (n-1)) (fix (λfact. λn. if n=0 then 1 else n * fact (n-1))))
0))

-> <beta rule>
3 * (2 * (1 * (λn. if n=0 then 1 else n * (fix (λfact. λn. if n=0 then 1 else n * fact (n-1))) (n-1)) 0))

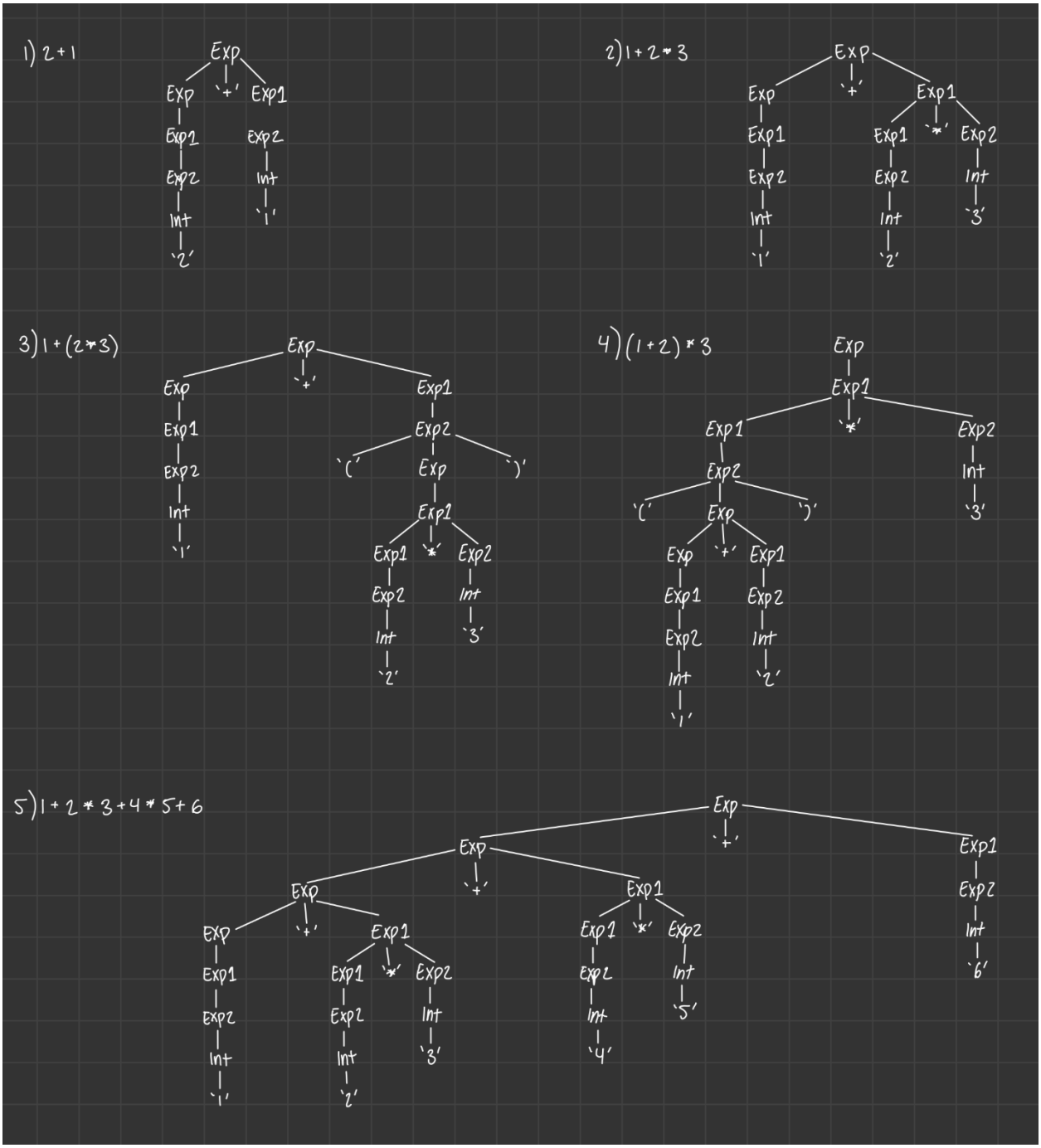
-> <beta rule>
3 * (2 * (1 * (if 0=0 then 1 else 0 * (fix (λfact. λn. if n=0 then 1 else n * fact (n-1))) (0-1))))

-> <def of if>
3 * (2 * (1 * (1)))

-> <multiplication>
6

2.7 Week 7

2.7.1 Homework



2.8 Week 8

2.8.1 Homework

```
Level 5: a + (b + 0) + (c + 0) = a + b + c
rw[add_zero]
rw[add_zero]
rfl

Level 6: a + (b + 0) + (c + 0) = a + b + c
rw[add_zero c]
rw[add_zero]
rfl

Level 7: For all natural numbers a, we have succ(a) = a + 1
rw[one_eq_succ_zero]
rw[add_succ]
rw[add_zero]
rfl

Level 8: 2 + 2 = 4
rw[four_eq_succ_three]
rw[three_eq_succ_two]
rw[two_eq_succ_one]
rw[one_eq_succ_zero]
rw[add_succ]
rw[add_succ]
rw[add_zero]
rfl

Natural Language Proof of Level 5:
a + (b + 0) + (c + 0) = a + b + c
a + b + c = a + b + c               by algorithm 1: addition (m + 0 = 0)
True by reflexivity
```

2.9 Week 9

2.9.1 Homework

```
ADDITION WORLD LEVEL 5

WITH INDUCTION: a + b + c = a + c + b

induction b
rw[add_zero]
rw[add_zero]
rfl
rw[add_succ]
rw[succ_add]
rw[add_succ]
rw[n_ih]
rfl
```

```
a + b + c = a + c + b

Induction on b:
Basis:   a + 0 + c = a + c + 0
   a + c = a + c + 0       def of +
   a + c = a + c        def of +
   True by reflexivity

Inductive Hypothesis: a + n + c = a + c + n

Inductive Step: a + n + c = a + c + n
       (a + n) + c = a + c + n     def of +
       (a + n + c) =  a + c + n     def of +
       (a + n + c) =  a + (c + n) def of +
       (a + n + c) =  (a + c + n) def of +
       (a + c + n) = (a + c + n) inductive hypothesis
       True by reflexivity

WITHOUT INDUCTION: a + b + c = a + c + b

rw[add_assoc]
rw[add_comm b]
rw[add_assoc]
rfl

(a + b) + c = (a + c) + b
a + (b + c) = (a + c) + b associativity
a + (c + b) = (a + c) + b commutativity
(a + c) + b = (a + c) + b associativity
True by reflexivity
```

2.10 Week 10

2.10.1 Homework

PARTY INVITES

LEVEL 6: example (C D S: Prop) (h : C D → S) : C → D → S := by
exact $\lambda cd \mapsto h<c, d>$

LEVEL 7: example (C D S: Prop) (h : C → D → S) : C D → S := by
exact $\lambda c \mapsto hc.leftc.right$

LEVEL 8: example (C D S : Prop) (h : (S → C) (S → D)) : S → C D := by
exact $\lambda s \mapsto <h.lefts, h.rights>$

LEVEL 9: example (R S : Prop) : R → (S → R) (¬S → R) := by
exact $\lambda r \mapsto <\lambda_{\text{ı}} \to r, \lambda_{\text{ı}} \to r>$

Discord Question: Why do we need to use a placeholder when there is no evidence?

2.11 Week 11

2.11.1 Homework

FALSIFICATION

LEVEL 9: example (A P : Prop) (h : P → ¬A) : ¬(P A) := by
exact λ ⟨p, a⟩ => (h p) a

LEVEL 10: example (A P : Prop) (h: ¬(P A)) : P → ¬A := by
exact λ p a => h ⟨p, a⟩

LEVEL 11: example (A : Prop)(h : ¬¬¬A) : ¬A := by
exact λ a => h (λ na => na a)

LEVEL 12: example (B C : Prop) (h : ¬(B → C)) : ¬¬B := by
exact λ nb => h (λ b => False.elim (nb b))

2.12 Week 12

2.12.1 Homework

```
hanoi 5 0 2
  hanoi 4 0 1
    hanoi 3 0 2
      hanoi 2 0 1
        hanoi 1 0 2 = move 0 2
        move  0 1
        hanoi 1 2 1 = move 2 1
      move 0 2
      hanoi 2 1 2
        hanoi 1 1 0 = move 1 0
        move  1 2
        hanoi 1 0 2 = move 0 2
    move 0 1
    hanoi 3 2 1
        hanoi 2 2 0
            hanoi 1 2 1 = move 2 1
            move 2 0
            hanoi 1 1 0 = move 1 0
        move 2 1
        hanoi 2 0 1
            hanoi 1 0 2 = move 0 2
            move 0 1
            hanoi 1 2 1 = move 2 1
  move 0 2
  hanoi 4 1 2
      hanoi 3 1 0
        hanoi 2 1 2
            hanoi 1 1 0 = move 1 0
            move 1 2
            hanoi 1 0 2 = move 0 2
        move 1 0
        hanoi 2 2 0
            hanoi 1 2 1 = move 2 1
            move 2 0
            hanoi 1 1 0 = move 1 0
      move 1 2
```

```
hanoi 3 0 2
    hanoi 2 0 1
        hanoi 1 0 2 = move 0 2
        move 0 1
        hanoi 1 2 1 = move 2 1
    move 0 2
    hanoi 2 1 2
        hanoi 1 1 0 = move 1 0
        move 1 2
        hanoi 1 0 2 = move 0 2
```

2.13 Week 13

2.13.1 Homework

2. I added ((ab)c). It should reduce to a b c. It reduced to (ab c).

I added (λx. (λy. x)) y. It became (λVar1.y), which is correct.

I added (λx. x x) (λx. x x). It resulted in: recursion error: maximum recursion depth exceeded.

3. Avoiding capture substitution works by adding new variables like Var1 since in some cases there are two variables with the same name, like x, but they are actually different. I added (λx. (λy. x)) y, and it became (λVar1.y), so it avoided capture substitution. It is implemented by having it generate a new variable name if the expression starts with a lambda and has a different variable name for tree[1].

4. I don't always get the expected result. I added ((ab)c), and it should reduce to a b c, but it reduced to (ab c), which is correct, but not what I expected. Not all computations reduce to normal form. I added (λx. x x) (λx. x x) It resulted in: recursion error: maximum recursion depth exceeded

5. The smallest expression I can find that does not reduce to normal form is (λx. x x) (λx. x x)

7.

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

8.

```
12: eval (((\m.(\n.(m n))) (\f.(\x.(f (f x))))) (\f.(\x.(f x))))
39: eval ((\m.(\n.(m n))) (\f.(\x.(f (f x)))))
   39: eval (\m.(\n.(m n)))
   44: sub ( (\n.(m n)), m, (\f.(\x.(f (f x)))) )
      82: sub ( (m Var1), m, (\f.(\x.(f (f x)))) )
         82: sub ( (m n), n, Var1 )
            85: sub ( m, n, Var1 )
            85: sub ( n, n, Var1 )
         85: sub ( m, m, (\f.(\x.(f (f x)))) )
         85: sub ( Var1, m, (\f.(\x.(f (f x)))) )
   45: eval (\Var1.((\f.(\x.(f (f x)))) Var1))
44: sub ( ((\f.(\x.(f (f x)))) Var1), Var1, (\f.(\x.(f x))) )
85: sub ( (\f.(\x.(f (f x)))), Var1, (\f.(\x.(f x))) )
```

```
   82: sub ( (\Var3.(Var2 (Var2 Var3))), Var1, (\f.(\x.(f x))) )
   82: sub ( (\x.(f (f x))), f, Var2 )
      82: sub ( (f (f x)), x, Var4 )
         85: sub ( f, x, Var4 )
         85: sub ( (f x), x, Var4 )
            85: sub ( f, x, Var4 )
85: sub ( x, x, Var4 )
            82: sub ( (f (f Var4)), f, Var2 )
               85: sub ( f, f, Var2 )
               85: sub ( (f Var4), f, Var2 )
                  85: sub ( f, f, Var2 )
                  85: sub ( Var4, f, Var2 )
85: sub ( Var1, Var1, (\f.(\x.(f x))) )
   45: eval ((\Var2.(\Var5.(Var2 (Var2 Var5)))) (\f.(\x.(f x))))
      39: eval (\Var2.(\Var5.(Var2 (Var2 Var5))))
      44: sub ( (\Var5.(Var2 (Var2 Var5))), Var2, (\f.(\x.(f x))) )
      45: eval (\Var6.((\f.(\x.(f x))) ((\f.(\x.(f x))) Var6)))
```

Discord Question: Are there any situations where the debugger can't or shouldn't be used?

3 Essay

One topic that was covered in this class that I found very interesting was the prisoner hat riddle. At first I only thought of this problem as a fun riddle, but then I realized that in many ways it is actually very similar to how computational systems encode information, execute algorithms, and reason under constraints. The solution also includes concepts like parity, boolean operations, and states that update.

In the riddle, it is said that you and nine other people have been captured by aliens. You will only not be eaten if you are proven to be highly logical and cooperative beings, so the aliens give you this test and if you pass you will be spared. They state that the ten of you will be placed in a single-file line facing forward, in order of height, where the tallest is in the back of the line, and the shortest is in the front. You are not allowed to turn, or step out of line. Each of you will randomly have either a black or a white hat placed on your head. You do not know the color of your own hat, but you can see all of the colors in front of you. Each person must then guess the color of their own hat, starting with the person in the back of the line and going in order towards the front. You can only say either "black" or "white", and you can't make any other noise or signal some other way. Nine out of the ten of you must guess correctly to be spared. You are allowed to discuss with your fellow humans before they start, what plan do you come up with?

In class, we discussed how one could solve this riddle, and I was the one to finally figure out the answer. The person at the back of the line (P1) would say "white" if they saw an even number of black hats, and "black" if they saw an odd number of black hats. Considering zero to be even, each person after that (P2-P10) would say "white" if they agree with the number of black hats the person behind them is seeing, and "black" if they see a different number of black hats than the person behind them.

Although this riddle does not directly discuss programming languages, it can still be modeled as a computational system. It has rules, inputs, and outputs. Each person receives input from looking at the people in front of them, and everyone except for the person at the back of the line also receives input from the people behind them. Based on these inputs, each person produces an output by either saying "black" or "white". Some rules include, people can't turn around, people can't signal, people can only speak once, people can only speak in order and people can only say "black" or "white".

Another relation to computation is that the solution to the riddle uses parity. Everything people say is in relation to if they see an even or odd number of hats. They do not say anything about the color of anyone's individual hat, but about the entire system. This is similar to encoding a single bit of information that is about the whole system. Parity is often used in programming and computation to efficiently encode data without using a lot of resources.

The riddle also illustrates how state is updated over time, similar to program execution. Each spoken answer becomes new information that updates the state each prisoner is currently in. When each prisoner says their color, all of the prisoners in front of them must keep track of if that prisoner is seeing an even or odd number of hats, which will eventually influence their own response. This reflects how variables are updated and read during a program's runtime. Instead of directly accessing hidden information, prisoners are constantly updating their knowledge with new information, which is very similar to states in programming.

The prisoner hat riddle serves as a clear and effective model of computation and programming languages. Through parity, and updating states, the puzzle demonstrates how abstract rules can be transformed into reliable computational procedures. Even though this problem is presented as a riddle, the solution relies on the same principles that are used in modern programming languages and computational systems. Analyzing the puzzle in this way highlights how computation is fundamentally about structured reasoning, not just writing code.

4 Evidence of Participation

4.1 Procedural Generation in Games - Computerphile

In this video, Zac Garby explains how procedural generation could work, using the board game, Carcassonne. He starts out by explaining the game, each player takes turns placing down a square piece of the map, but each edge must line up with an edge that's on the board. So, if a road is currently going off the map, the piece you put down there must also have a road that will line up with it. He writes a program to simulate the game, and has tiles first exist with all of the possible tiles that one could place in that spot. He does this using wave functions. First, he just has the program pick one of the possibilities randomly, and then recalculate possibilities for other tiles based on that. He states that this is not very life-like, so he then adds weights to some of the possibilities, to make it more likely that the program would add a tile to continue a road, as opposed to just adding the tile to the side. In the actual game, the player will just get the first tile at the top of the stack, and have to place that one, so he also implements that, instead of having the program pick the tile with the least amount of possibilities.

Question: What is more similar to the way that video games actually procedurally generate maps? Putting down the tile that has the least amount of possibilities, or finding a place for the tile on top of the deck?

4.2 Reinforcement Learning - Computerphile

In this video, Nick Hawes explains how reinforcement learning works. He states that in this system you have an agent, and the environment. The agent will make a decision, and then that will affect the environment. The agent will then be rewarded more if the outcome for the environment is better, and then it can improve based on that feedback. It is this cycle of being in a certain state, then performing an action, getting some sort of reward, and then seeing the next state, so then it can perform another action. In order to ensure that the program does not get stuck doing the same thing over and over again, there is a certain probability that it will explore another new action vs exploit the action that has been working for every time that it performs an action. He explains this through the example of you choosing a method of transportation to get to work. You will be rewarded because your boss is happier the earlier you get to work. You wouldn't want to be constantly trying new methods to get to work, because some would be very inefficient, and would waste a lot of time, like a jetski, or hot air balloon.

Question: How could reinforcement learning be integrated into another AI system?

4.3 Making Game Design Accessible to Youth

In this video, Jonathan Weinberger, who has been making video games for sixteen years, is helping kids who are around 11 or 12 learn how to program video games. He explains how math applies to video games. Some examples he gives are that parabolas apply to Angry Birds, where you shoot the towers, and the Pythagorean theorem can calculate distance in video games. He emphasizes the vast amount of fields that skills learned in game development can be applied to. Some examples are the film, real estate, medicine, and automotive industries. Even if the kids do not end up wanting to go into game development, the skills they will learn while making games will make them great candidates for many other jobs. He encourages the kids to start making games now, while they are young so that they can get a lot of experience early on. He says that this will put them ahead of all of the other people who wait until college to make games, and that they might not even need to go to college, and could get hired to work for a game developer right out of high school. He says that right out of high school they could be making 70 to 90 thousand dollars a year, or they could go to the best schools for computer science if they decide to go to college.

Question: Does this advice still apply, considering how saturated the market is now?

4.4 The Problem with A.I. Slop! - Computerphile

In this video, Mike Pound, from the University of Nottingham, discussed AI generated content. He states that most articles are generated by AI. People are looking to just generate a bit of money with not very much work. They will just have AI generate a website, and a story for something like a recipe in order to draw people to the website. They know that not many people will actually read through the website, but they will just use it to profit off of the advertisements from these websites. We know that the things AI writes are not always good or accurate, so this makes it harder for people to find the correct information when looking things up. It also makes it harder to train AI, since the data that is being used to train the AI models is largely generated by AI. Since not all data generated by AI is true, AI is now being trained by more false data, and therefore producing more false data. He suspects that with this increased use of AI, and there being more AI produced content, people will start to just stick with outlets that they know to have reputable, human verified information.

Question: How can those who make AI prevent people from abusing it, and generating a large amount of articles with false information?

5 Conclusion

This course really strengthened my understanding of the theory behind computation. Although most of this class was about theory, and not actual coding, I still learned a lot about software engineering. Concepts such as lambda calculus, recursion, and riddles like the prisoner hat problem helped me to approach topics from a more logical viewpoint and formalize concepts to be more understandable as computation.

I think that learning lambda calculus was a very useful part of the course. This can be applied to software engineering through recursion, and functions. Another thing that I found really interesting were the riddles, like the towers of hanoi, the MU puzzle, and the prisoner hat problem. Although there was not any actual coding while solving these problems, it felt like we approached the problems in a way that made me feel like I would definitely be able to code the solutions by the end.

In class, I really appreciated all of the extra help for all of the problems we were working through. Getting time to work on them in class and ask questions was very useful to me and definitely improved my learning.

I do wish that in class we could be directly shown more of how what we are learning fits into software engineering, and how we might use these topics while solving coding questions. There were some times during the lecture where I was a bit confused, and got lost, but after asking questions I was eventually able to get back on track.

Overall, this course was very useful. Learning lambda calculus, and how to solve riddles helped me to think more logically, and will therefore improve my ability to solve problems through writing code. I wish we were able to see more direct applications to software engineering in class, but I still think that many applications become clear the more you think about it.

References

[BLA] Author, Title, Publisher, Year.