CPSC 354 Report

Andrew Eppich
Chapman University

November 3, 2024

Abstract

Contents

1 Introduction

2 Homework 1

2.1 Question 5

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

2.1.1 Proof Explanation

For this question, the Lean proof is related to the corresponding proof in mathematics because we know that we can use the additive identity property, which says that $x + 0 = x$. By using this, we can simplify $b + 0$ and $c + 0$ easily to get $a + b + c = a + b + c$, which we can determine is the same by the reflexivity property, which states that if $a = b$, then a and b are identical. Therefore, $a + b + c$ is identical to $a + b + c$.

2.2 Question 6

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

2.3 Question 7

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

2.4 Question 8

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

2.5 Discord Question

I was wondering if the computers use of discrete math extends to all program computations or just math computations

3 Homework 2

3.1 Question 1

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

3.2 Question 2

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

3.3 Question 3

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

```
rw [succ_add]
rfl
```

3.4 Question 4

```
induction a with d hd
rw [zero_add]
rw [zero_add]
rfl
rw [succ_add]
rw [succ_add]
rw [succ_add]
rw [hd]
rfl
```

3.4.1 Explaination

The lean proof relates to the proof in mathematics because it uses induction to solve the problem. Then the Lean proof is solved by solving the equation of the successors. Just like in mathematics it uses simple rules to change the positioning of the parenthesis so each side is exactly the same. This is exactly like how the mathematical proof would be written.

3.5 Question 5

```
induction a with d hd
rw [zero_add]
rw [zero_add]
rw [add_comm]
rfl
rw [add_comm]
rw [add_comm]
rw [succ_add]
rw [succ_add]
rw [succ_add]
rw [succ_add]
rw [hd]
rfl
```

3.6 Discord Question

I was wondering how discrete math and the recursive algorithms we talked about fit into a programming language and how it actually works

4 Homework 3

4.1 Discord Post

Discord Name: Andrew Eppich. In my literature review with ChatGPT, I explored interpreted vs. compiled programming languages. I found that interpreted languages are changed from user to machine code line by line, which is inefficient. Compiled languages are compiled from user to machine code all at once and then run which makes it faster and easier to spot errors. From there I explored interpreted languages and their role in machine learning as well as their history in machine learning. I first found that compiled was much more efficient than interpreted. I then found out that interpreted is mainly used for machine learning. It is mainly used because of the extensive amount of libraries used with interpreted languages, especially Python. Some of those libraries include NumPy, pandas, scikit-learn, TensorFlow, and Matplotlib. These libraries are crucial for machine learning because they are associated with data processing and deep learning. I then took a look into the history of programming languages with machine learning. I found that at first compiled languages were used from the 1950s-1980s. In 1991, Python was developed which became the standard for machine learning in the early 2000s. Python became the main language for machine learning from 2010 and on because of its libraries TensorFlow and PyTorch. https://github.com/AndrewEppich/LLM-Literature-Review/blob/main/README.md

4.2 Reports Voted For

https://github.com/zackklopukh/LLMReport

https://github.com/maxler0y/354_HW3

5 Homework 4

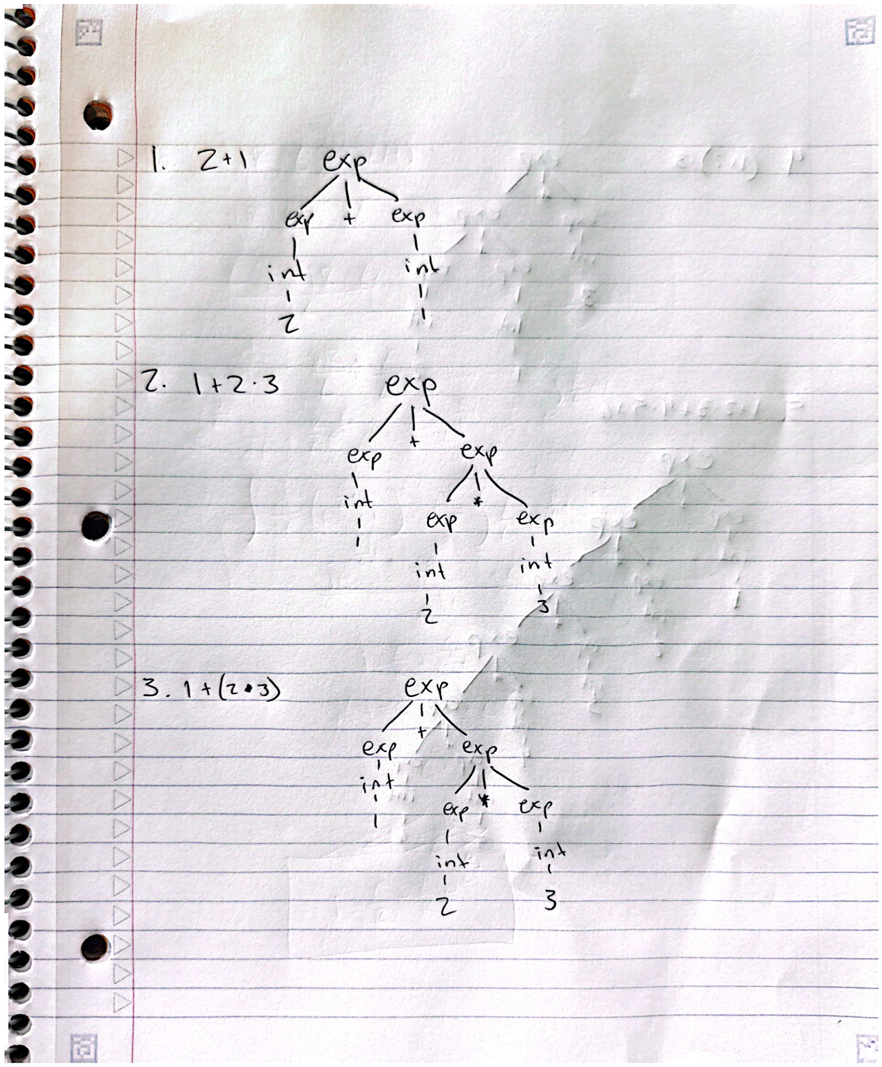

Figure 1: Homework 4 - Page 1

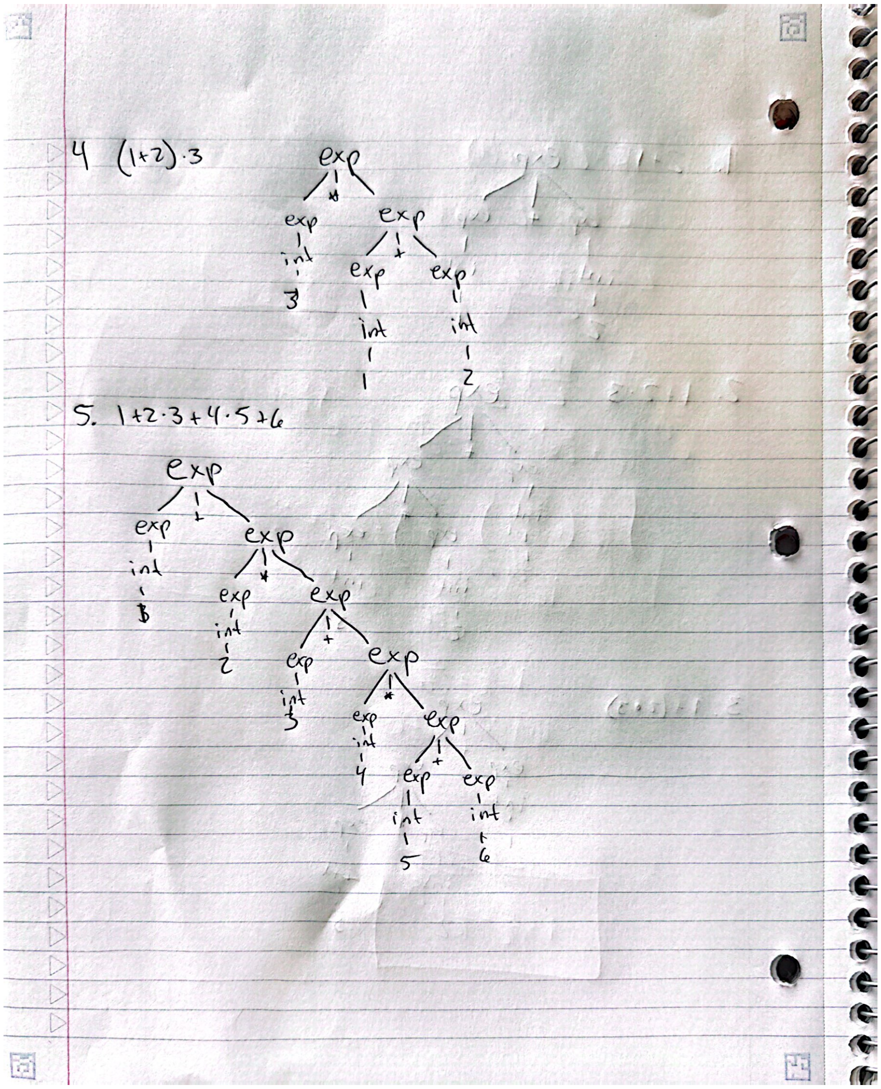

Figure 2: Homework 4 - Page 2

5.1 Discord Question

How complex do parsing algorithms get the farther down the level of programming languages you go?

6 Homework 5

6.1 Question 1

```
exact todo_list
```

6.2 Question 2

```
exact and_intro p s
```

6.3 Question 3

```
exact <a,i>, <o,u>
```

6.4 Question 4

```
have p := vm.left
exact p
```

6.5 Question 5

```
exact and_right h
```

6.6 Question 6

```
have a:= and_left h1
have b := and_right h2
exact < a, b>
```

6.7 Question 7

```
have h1 := h.right
have h2 := h.left
have h3 := h2.right
exact h.left.right.left.left.right
```

6.8 Question 8

```
have h1 := and_left h
have h2 := and_right h
have h3 := and_right h2
have h4 := and_left h3
have h5 := and_left h4
have h6 := and_right h1
have h7 := and_left h1
have h8 := and_left h7
have h9 := and_right h7
exact < h6, h5, h8, h9>
```

6.8.1 Proof Explaination

I took the left and right sides and set them equal to different variables. Next I took those broken down parts and continued to break them down into different variables until I had a single value for a single variable. Then I added the desired variables together to acheive the desired equation

6.9 Discord Question

When thinking about the final Lean problem on the homework, I solved it by breaking up each term over and over again with and_left and and_right until there were single terms. It was like a tree forming a new branch and leaves every time the equation was broken up. My question is, how is the logic of AND and and_left and and_right used in programming languages and other applications? And is this just the logic that is used to parse a BST?

7 Homework 6

7.1 Lecture Explaination

```
In lecture this week we started learning about lambda caclulus. We learned about the syntax of P
    -> Q. We learned about the rule of how to eliminate so we can prove P -> Q by saying P is
    evidence of B so h \( A \) :B. We also learned that we can use lambda to create a function of
    terms that is able to be broken apart. We alsp learned the transitive property that says \( P
    -> Q \) -> \( Q -> R \) -> P -> R.
```

7.2 Question 1

```
exact bakery_service p
```

7.3 Question 2

```
  have h1 : C -> C := fun C \mapsto C
  exact h1
```

7.4 Question 3

```
    exact fun h : I \mapsto and_intro (and_right h) h.left
```

7.5 Question 4

```
exact fun c => h2 (h1 c)
```

7.6 Question 5

```
have q : Q := h1 p
have t : T := h3 q
exact h5 t
```

7.7 Question 6

```
exact fun c => fun d => h <c, d>
```

7.8 Question 7

```
exact fun h1 => h h1.1 h1.2
```

7.9 Question 8

```
exact fun s => < h.1 s, h.2 s>
```

7.10 Question 9

```
exact fun r => < fun s => r, fun ns => r>
```

7.11 Discord Question

are lambda calculus proofs a part of the compiler? if so, how do they interact with the code that is being compiled?

8 Homework 7

8.1 Question 1

Beta Reduction 1:

$$((\lambda m.\lambda n.m\, n)\,(\lambda f.\lambda x.f(f\, x)))\,(\lambda f.\lambda x.f(f(f\, x)))$$

Beta Reduction 2:

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

Beta Reduction 3:

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

Beta Reduction 4:

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

Beta Reduction 5:

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

Beta Reduction 6:

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

Beta Reduction 7:

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

8.2 Question 2

The function takes two Church numerals m and n and applies m to n. Applying m to n is like multiplying two numbers so it implements multiplication

8.3 Discord Questions

how complex does lambda calculus recursion get in computer systems?

9 Homework 8-9

9.1 Question 2

a b c d reduces to that because in the code it parses the expression and characterizes it as app which is in the lark grammar defined as exp1 exp2 so it takes a and b and puts parenthesis around it. Then it goes to the next expression whcih is /(a + b /) and then gets the next term which is c so it puts parenthesis around /(a + b /) and c. Then it does this process again with d. then if you have /(a/) it will just return a because in lark grammar it is characterized as var. In the code if the expression is var it returns just the term in the parenthesis so it just returns a

9.2 Question 3

Capture avoiding subsittuion works because when when substituting variables, if there is a free variable in the expression then substituting variables in the equation can cause the expression to change its meaning. Capture-avoiding substitutions avoid capturing free variables in the expression when changing a variable which can cause the whole expression to change its meaning. in the code it checks bound variable matches the substitution target, if it does it leaves it unchanged if not it generates a new variable

9.3 Question 4

We do not always get the expected results. Well-defined expressions typically return the expected results, but if there is an error handling an edge case, it would not return the expected results. Not all computations reduce to normal form. Expressions like $(\lambda x.\, x\, x)$ and $(\lambda x.\, x\, x)$ create an infinite loop of self-application that will never be stable.

9.4 Question 5

This is the smallest lambda function that doesn't reduce to normal form: $(\lambda x.\, x\, x)\,(\lambda x.\, x\, x)$. This reduces to itself, so there is no normal form.

9.5 Quesdtion 7

Initial Expression:

$$((\lambda m.\, \lambda n.\, m\, n)\,(\lambda f.\, \lambda x.\, f\,(f\,x)))\,(\lambda f.\, \lambda x.\, f\,(f\,(f\,x)))$$

First Application: Apply $(\lambda m.\, \lambda n.\, m\, n)$ to $(\lambda f.\, \lambda x.\, f\,(f\,x))$:

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

Second Application: Now apply $(\lambda n.\, (\lambda f.\, \lambda x.\, f\, (f\, x))\, n)$ to $(\lambda f.\, \lambda x.\, f\, (f\, (f\, x)))$, substituting n with $(\lambda f.\, \lambda x.\, f\, (f\, (f\, x)))$:

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

Evaluating the Final Application: Substitute $(\lambda f.\, \lambda x.\, f\, (f\, (f\, x)))$ for f in $(\lambda f.\, \lambda x.\, f\, (f\, x))$:

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

Result: The final expression (in normal form) is:

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

9.6 Question 8

```
12: evaluate(((\m.\n. m n) (\f.\x. f (f x))) (\f.\x. f x))
    39: evaluate((\m.\n. m n) (\f.\x. f (f x)))
        12: evaluate(\m.\n. m n) = \m.\n. m n
        12: evaluate(\f.\x. f (f x)) = \f.\x. f (f x)
        51: substitute(\n. m n, m, (\f.\x. f (f x)))
            51: substitute((\f.\x. f (f x)) n, n, (\f.\x. f x))
        12: evaluate((\n. (\f.\x. f (f x)) n))
            51: substitute((\f.\x. f (f x)) n, n, (\f.\x. f x))
    39: evaluate((\f.\x. f (f x)) (\f.\x. f x))
        51: substitute((\x. f (f x)), f, (\f.\x. f x))
            51: substitute((\x. ((\f.\x. f x) ((\f.\x. f x) x))), f, (\f.\x. f x))
```

9.7 Discord Questions

Week 8: What does it mean in terms of a programming language output when something does not reduce to normal form

Week 9: how often is tracing used in the debugger in industry

10 Homework 10

1. I found that figuring out the debugger and how to use the print statements like print(linearize(x)) was the hardest aspect because it took a while for me to get it to work

2. I came up with the key insight because I was struggling to figure out how to make sure the MWE returned the correct result. I figured out that if the expressions were identical then the result would be the same as the expressions. So i added a part that checked for that

3. The most interesting takeaway from this was learning how to use code to evaluate the lambda expressions

10.1 Discord Question

week 10: what aspects if coding prompts substitutions to be made to evaluate it