CPSC-354 Report

Chaz Gillette

October 31, 2024

Contents

1 Week 1: Introduction to Lean and Natural Number Game Tutorial

In week one, we worked with Lean and reviewed our discrete math knowledge in the Number Game Tutorial. Problems and solutions are listed below.

Homework Solutions: Week 1

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 b]

rfl
```

Level 7: succ_eq_add_one Theorem

```
Theorem succ_eq_add_one: 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: $2 + 2 = 4$

```
2 + 2 = 4.
```

```
nth_rewrite 2 [two_eq_succ_one]  -- only change the second ‘2‘ to ‘succ
    ↪ 1‘.

rw [add_succ]

rw [one_eq_succ_zero]

rw [add_succ, add_zero]  -- two rewrites at once

rw [three_eq_succ_two]  -- change ‘succ 2‘ to ‘3‘

rw [four_eq_succ_three]

rfl
```

Detailed Explanation: Level 7 Proof

I chose to explain the proof for level seven because this is where we make the breakthrough with addition. Our goal is to prove that the successor of a is equal to $a + 1$. So, in step one, we want to rewrite one as the successor of zero. That gives us $\text{succ } a = a + \text{succ } 0$. The next step is `add_succ` so that $\text{succ } a = \text{succ } (a + 0)$. After that, we can remove the zero by `rw [add_zero]`, which will leave us with $\text{succ } a = \text{succ } a$, which is thus proven true with the reflexive property.

Lessons from the Assignments

Lesson from Week 1

Week one was our review and introduction to the math side of what we'll be learning this semester. We started by revisiting the basic rules of discrete math. This meant getting back into the flow of writing out our proofs using the rules that we have access to. With only natural numbers to start, we started looking at successors again and eventually proving our way toward addition.

For me, this was a needed refresher because it's been a moment since I took discrete math, and I'm unfamiliar with writing my proofs as code, which is a learning curve for me. I'm a very pen-to-paper mathematician, so thinking about math at the same time I'm trying to recall syntax for code is a challenge for me. That said, we went through eight levels of proofs, and I was able to begin to get the hang of it.

I'm looking forward to bridging the gap between my math knowledge and how I involve it when I code. Sometimes I feel like I have the education to understand the concepts, but I struggle to apply them when I'm coding. The speed in which I type out code is not as quick as how I think about what I'd like to apply. This first assignment was a nice intro to opening my eyes as to what it might look like to get faster at that and also write technical reports in a coding environment as well. By shifting everything I do—the code, the math, the reporting—into an IDE, I know that I'll be able to get more comfortable working in that environment.

2 Week 2: Finishing the NNG Addition World

In week two, we focused on completing the Natural Number Game (NNG) Addition World, which helped solidify our understanding of addition in Lean. The problems and solutions for Levels 1-5 are listed below.

Homework Solutions: Week 2

Level 1: zero_add

```
theorem zero_add (n : nat) : 0 + n = n := by
induction n with d hd
rw [add_zero]
rfl
rw [add_succ]
rw [hd]
rfl
```

Level 2: succ_add

```
theorem succ_add (a b : nat) : succ a + b = succ (a + b) := by
induction b with d hd
rw [add_zero]
rw [add_zero]
rfl
rw [add_succ]
rw [hd]
rw [add_succ]
rfl
```

Level 3: add_comm

```
theorem add_comm (a b : nat) : a + b = b + a := by
induction b with d hd
rw [add_zero]
rw [zero_add]
rfl
rw [add_succ]
rw [hd]
rw [succ_add]
rfl
```

Level 4: add_assoc

```
theorem add_assoc (a b c : nat) : a + b + c = a + (b + c) := by
induction c with d hd
rw [add_zero]
rw [add_zero]
rfl
rw [add_succ]
rw [hd]
rw [add_succ]
rfl
```

Level 5: add_right_comm

```
theorem add_right_comm (a b c : nat) : a + b + c = a + c + b := by
induction c with d hd
rw [add_zero]
rw [add_zero]
rfl
rw [add_succ]
rw [hd]
rw [add_succ]
rw [add_comm b d]
rfl
```

Mathematical Proof for Level 5: add_right_comm

The goal is to prove the right commutativity of addition, meaning for all natural numbers a, b, and c, the equation $a + b + c = a + c + b$ holds. This is done by using induction on c.

Base Case: When $c = 0$, we need to prove:

$$a + b + 0 = a + 0 + b$$

Using the identity property of addition, we know that $a + 0 = a$ and $a + b + 0 = a + b$. Thus, both sides simplify to:

$$a + b = a + b$$

which is true.

Inductive Step: Assume that $a + b + c = a + c + b$ holds for some c. Now, we must show that it holds for $c + 1$:

$$a + b + (c + 1) = a + (c + 1) + b$$

Using the associative and commutative properties, and the inductive hypothesis, we can manipulate the expression to prove the equality.

Detailed Explanation: Level 5 Proof (add_right_comm)

In this proof, the goal is to show that for all natural numbers a, b, and c, the equation $a+b+c=a+c+b$ holds. This is the right commutativity of addition. The proof is done by induction on c.

1. **Base Case ($c = 0$):** We need to prove $a + b + 0 = a + 0 + b$. Since adding zero doesn't change the value, both sides simplify to $a + b$, which are equal.

2. **Inductive Step:** Assume the statement holds for some c. That is, $a + b + c = a + c + b$. We need to show $a + b + (c + 1) = a + (c + 1) + b$.

- Starting with the left side:

$$a + b + (c + 1) = (a + b + c) + 1$$

- By the inductive hypothesis:

$$(a + c + b) + 1 = a + c + (b + 1)$$

- Since addition is associative:

$$a + (c + (b + 1)) = a + ((c + 1) + b)$$

- Therefore, the right-hand side matches, proving the statement.

Lessons from the Assignments

Lesson from Week 2

In week two, we delved deeper into the relationship between mathematical proofs and Lean code proofs. We learned how to perform proofs recursively, using induction. This requires establishing a base case and then proving that if the statement holds for n, it also holds for $n + 1$. By doing so, we can prove statements for all natural numbers.

I learned how to translate mathematical induction into Lean code, which strengthened my understanding of both programming and mathematical proof techniques. This experience highlighted the importance of rigorous logical reasoning in coding proofs.

3 Week 3: Using LLMs for Literature Review

In Week 3, I explored the topic of **Quantum Programming Languages** using an LLM to guide my investigation. I also began to code a computer in Python without the help of any libraries.

Link to the Full Literature Review

The full literature review, including the questions and answers from the LLM, can be found here.

Discord Post

My Discord name is Chaz Gillette, and below is a copy of my Discord post summarizing the literature review:

> "What are some of the key differences between classical logic and constructive logic that we should be mindful of when working through the Lean tutorials?"

Reviews I Voted For

I voted for the following two reviews after reading them:

1. Review 1
2. Review 2

Lessons Learned

In week 3, I learned a lot about what makes programming for quantum computing different; mostly, this has to do with the non-binary nature of quantum computing compared to current computers. Outside of my report, I learned how to break down order of operations when coding. Using functions to call other functions within parentheses, I began to tackle mathematical equations like a real calculator would.

4 Week 4: Introduction to Parsing and Context-Free Grammars

In Week 4, we delved into the concepts of parsing and context-free grammars, exploring how they are used to translate concrete syntax into abstract syntax. This is a crucial step in understanding how programming languages are processed and interpreted.

Key Concepts

- **Concrete syntax**: Represents a program as a string (e.g., "1 + 2 * 3")
- **Abstract syntax**: Represents a program as a tree structure
- **Parsing**: The process of transforming concrete syntax into abstract syntax
- **Context-free grammar**: A set of rules that define the structure of a language

Context-Free Grammar for Arithmetic Expressions

We studied the following context-free grammar for arithmetic expressions:

```
Exp -> Exp '+' Exp1
Exp1 -> Exp1 '*' Exp2
Exp2 -> Integer
Exp2 -> '(' Exp ')'
Exp -> Exp1
Exp1 -> Exp2
```

Homework Solutions: Week 4

For the homework, we were asked to parse various expressions using the given context-free grammar. The step-by-step derivations for each problem are figures 1-5 after the conclusion.

The parsing process demonstrates how the grammar rules are applied to derive the final expression, showing the hierarchical structure of the arithmetic operations.

Lessons Learned

In Week 4, I gained several important insights:

1. The importance of context-free grammars in defining the structure of programming languages
2. How parsing bridges the gap between concrete syntax (what we write) and abstract syntax (how the computer interprets it)
3. The hierarchical nature of expressions and how this is captured in abstract syntax trees
4. The role of parsing in compiler design and language processing

This week's content has deepened my understanding of how programming languages are structured and interpreted, providing a foundation for more advanced topics in language design and implementation.

5 Week 5: Logic Game Tutorial and Lecture Content

In Week 5, we focused on the content covered in the lectures and completed the Lean logic game's tutorial world. Below are the solutions for Levels 1 through 8, along with a proof for Level 8 written in mathematical logic.

Solutions: Lean Logic Game Tutorial

1. **Level 1:**

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

2. **Level 2:**

```
example (P S : Prop)(p: P)(s : S) : P ∧ S := by
exact ⟨p, s⟩
```

3. **Level 3:**

```
example (A I O U : Prop)(a : A)(i : I)(o : O)(u : U) : (A ∧ I) ∧ O ∧ U := by
exact ⟨⟨a, i⟩, o, u⟩
```

4. **Level 4:**

```
example (P S : Prop)(vm: P ∧ S) : P := by
exact vm.left
```

5. **Level 5:**

```
example (P Q : Prop)(h: P ∧ Q) : Q := by
exact h.right
```

6. **Level 6:**

```
example (A I O U : Prop)(h1 : A ∧ I)(h2 : O ∧ U) : A ∧ U := by
exact ⟨h1.left, h2.right⟩
```

7. **Level 7:**

```
example (C L : Prop)(h: (L ∧ (((L ∧ C) ∧ L) ∧ L ∧ L ∧ L)) ∧ (L ∧ L) ∧ L) : C := by
exact h.left.right.left.left.right
```

8. **Level 8:**

```
example (A C I O P S U : Prop)
(h: ((P ∧ S) ∧ A) ∧ ¬I ∧ (C ∧ ¬O) ∧ ¬U) : A ∧ C ∧ P ∧ S := by
exact ⟨h.left.right, h.right.right.left, h.left.left.left, h.left.left.right⟩
```

Level 8: Formal Proof in Mathematical Logic

We want to show: If $((P \wedge S) \wedge A) \wedge \neg I \wedge (C \wedge \neg O) \wedge \neg U$, then $A \wedge C \wedge P \wedge S$.

Proof:

1. From $((P \wedge S) \wedge A)$, we have: - $P \wedge S$ (left side) - A (right side)
2. From $P \wedge S$, we have: - P - S
3. From $C \wedge \neg O$, we have C.
4. Therefore, combining A, C, P, and S, we get $A \wedge C \wedge P \wedge S$.

Discussion Question on Discord

During this week's assignment, I asked the following question on Discord:

> "How scalable is Lean if applied to large-scale software verification projects?"

6 Week 6: Implication in Lean Logic

In Week 6, we delved into the concept of implication using the Lean Logic game tutorial. This week focused on understanding and applying implication in logical propositions.

Key Concepts

- Implication in propositional logic
- Lambda functions in Lean
- Conjunction ($\wedge$) and its relationship with implication
- Function composition in logical proofs

Lean Logic Game Tutorial Solutions

We completed levels 1-9 of the Lean Logic game tutorial, focusing on implication. Here are the solutions:

1. **Level 1:**

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

2. **Level 2:**

```
example (C: Prop) : C → C := by
exact λ h : C, h
```

3. **Level 3:**

```
example (I S: Prop) : I ∧ S → S ∧ I := by
exact λ h : I ∧ S, ⟨h.right, h.left⟩
```

4. **Level 4:**

```
example (C A S: Prop) (h1 : C → A) (h2 : A → S) : C → S := by
exact λ c : C, h2 (h1 c)
```

5. **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) : U := by
exact h5 (h3 (h1 p))
```

6. **Level 6:**

```
example (C D S: Prop) (h : C ∧ D → S) : C → D → S := by
exact λ c d, h ⟨c, d⟩
```

7. **Level 7:**

```
example (C D S: Prop) (h : C → D → S) : C ∧ D → S := by
exact λ h1 : C ∧ D, h h1.left h1.right
```

8. **Level 8:**

```
example (C D S : Prop) (h : (S → C) ∧ (S → D)) : S → C ∧ D := by
exact λ s : S, ⟨h.left s, h.right s⟩
```

9. **Level 9:**

```
example (R S : Prop) : R → (S → R) ∧ (¬S → R) := by
exact λ r : R, ⟨λ _ , r, λ _ , r⟩
```

Reflections and Insights

We saw how lambda functions can be used to construct implications and how logical statements can be built and proven using basic principles of propositional logic.

The most intersting was the interaction between conjunction ($\wedge$) and implication ($\rightarrow$), as seen in levels 6 and 7. These exercises showed how we can convert between statements involving conjunctions and nested implications.

Discussion Question on Discord

During this week's assignment, I asked the following question on Discord:

> "Which proofs would generally be harder for AI, implication or inductive?"

7 Week 7: Lambda Calculus and Church Numerals

In Week 7, we explored lambda calculus and the representation of natural numbers using Church numerals.

Key Concepts

- Lambda calculus and term reduction
- Church numerals for natural numbers
- Arithmetic operations using lambda expressions

Lambda Term Reduction

We reduced the following lambda term:

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

1. **Initial Expression:**

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

2. **First Reduction:** Apply $\lambda m.\ \lambda n.\ m\ n$ to $\lambda f.\ \lambda x.\ f(f\ x)$:

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

3. **Second Reduction:** Apply the resulting function to $\lambda f.\ \lambda x.\ f(f(f\ x))$:

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

4. **Third Reduction:** Expand the abstraction over x:

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

5. **Fourth Reduction:** Simplify the inner application:

$$\lambda x.\ (\lambda x.\ x(x(x\ x)))\ ((\lambda x.\ x(x(x\ x)))\ (x(x(x\ x))))$$

6. **Final Reduction:** Simplify to obtain the Church numeral 6:

$$\lambda x.\ x(x(x(x(x(x\ x)))))$$

Function Implemented by $(\lambda m.\ \lambda n.\ m\ n)$

The function $(\lambda m.\ \lambda n.\ m\ n)$ implements **multiplication** on natural numbers using Church numerals. Applying one numeral to another composes their functions, resulting in the product of the two numbers.

Discussion Question

This weeks discord question is:

> "What challenges do other operations like subtraction and division using lamda calculus and Church numerals present compared to addition and multiplication?"

8 Week 8: Implementing a Lambda Calculus Interpreter

In Week 8, we explored the implementation of a simple lambda calculus interpreter and delved into key concepts such as expression reduction, capture-avoiding substitution, and normal forms.

Key Concepts

- Lambda calculus expression parsing and evaluation
- Left-associativity of function application
- Capture-avoiding substitution in lambda expressions
- Normal forms and non-terminating computations

Adding Lambda Expressions to `test.lc` and Running the Interpreter

We added various lambda expressions from lectures on Lambda Calculus and Church Encodings to a file named `test.lc`. This allowed us to test the interpreter and observe how it evaluates different expressions. Here are some of the expressions we included:

```
-- Identity function applied to a value
(\x. x) a

-- Boolean True applied to two values
(\t. \f. t) a b

-- Boolean False applied to two values
(\t. \f. f) a b
```

```
-- Church numeral for one
(\f. \x. f x)

-- Successor function applied to one
(\n. \f. \x. f (n f x)) (\f. \x. f x)

-- Addition of one and two
(\m. \n. \f. \x. m f (n f x)) (\f. \x. f x) (\f. \x. f (f x))
```

We ran the interpreter using the command:

```
python interpreter.py test.lc
```

The interpreter processed each expression, performing beta reductions and outputting the results. For example, applying the identity function to `a` resulted in `a`, and adding Church numerals for one and two yielded the Church numeral for three.

Left-Associativity of Application and Parentheses Reduction

We explored why the expression `a b c d` reduces to `(((a b) c) d)` and why `(a)` reduces to `a`. In lambda calculus, function application is **left-associative**, meaning that multiple applications are grouped from the left. Thus:

$$a\ b\ c\ d \quad \equiv \quad (((a\ b)\ c)\ d)$$

Parentheses in lambda calculus are used for grouping and do not alter the meaning of the expression. Therefore, `(a)` simply reduces to `a`.

Capture-Avoiding Substitution and Interpreter Limitations

Capture-avoiding substitution is a crucial mechanism in lambda calculus interpreters to ensure that free variables in substituted expressions do not become inadvertently bound, which can alter the intended meaning of expressions.

Issues in the Interpreter Implementation

While our interpreter attempts to implement capture-avoiding substitution, we discovered that it does not handle variable capture correctly in certain cases. Specifically, the interpreter:

- Fails to properly rename bound variables during substitution, leading to variable capture.
- Stops reduction prematurely when it encounters variables that should be further reduced.
- Does not consistently generate fresh variable names to avoid conflicts.

Example Demonstrating the Issue

Consider evaluating the expression:

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

When we run this expression through the interpreter, we observe that it gets stuck and does not fully reduce to the expected normal form. Here's what happens step by step:

1. **Initial Expression:**

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

2. **First Beta Reduction:** Substitute m with $\lambda f.\ \lambda x.\ f(f\ x)$ in $\lambda n.\ m\ n$:

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

   The interpreter should generate a fresh variable to avoid capture but fails to do so properly.

3. **Second Beta Reduction:** Apply the function to $\lambda f.\ \lambda x.\ f(f(f\ x))$:

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

   The variable n is replaced with $\lambda f.\ \lambda x.\ f(f(f\ x))$, but the interpreter does not handle the substitution correctly.

4. **Incorrect Variable Capture:** The interpreter incorrectly allows the bound variable f in $\lambda f.\ \lambda x.\ f(f\ x)$ to capture the free variable f from the argument, leading to an incorrect expression.

5. **Premature Termination:** Due to the improper handling of variable names, the interpreter stops reducing the expression further, resulting in an incomplete evaluation.

Analysis of the Interpreter's Substitution Function

The substitution function in the interpreter is intended to perform capture-avoiding substitution by generating fresh variable names. However, it has shortcomings:

```python
def substitute(tree, name, replacement):
    if tree[0] == 'var':
        if tree[1] == name:
            return replacement
        else:
            return tree
    elif tree[0] == 'lam':
        if tree[1] == name:
```

```
            return tree
        else:
            fresh_name = name_generator.generate()
            return ('lam', fresh_name, substitute(
                substitute(tree[2], tree[1], ('var', fresh_name)),
                name, replacement))
    elif tree[0] == 'app':
        return ('app', substitute(tree[1], name, replacement),
                       substitute(tree[2], name, replacement))
    else:
        raise Exception('Unknown tree', tree)
```

Issues Identified:

- The function generates fresh variable names but does not consistently apply them throughout the expression.
- When substituting within a lambda abstraction, it may not correctly replace all instances of the bound variable, leading to variable capture.
- The interpreter may leave unevaluated variables in the expression, causing it to stop reducing prematurely.

Specific Problem in Our Interpreter

In our interpreter, when evaluating the expression $((\lambda m.\ \lambda n.\ m\ n)\ (\lambda f.\ \lambda x.\ f(f\ x)))\ (\lambda f.\ \lambda x.\ f(f(f\ x)))$, the substitution function fails to avoid capturing the free variable f during the substitution of n. This happens because:

- The bound variable f in the inner lambda abstractions conflicts with the free variable f in the replacement expression.
- The interpreter does not generate a fresh variable name for f, leading to incorrect binding.
- As a result, the variable f gets incorrectly bound, and the expression cannot be reduced further.

Consequences of the Interpreter's Limitations

Due to the incorrect implementation of capture-avoiding substitution:

- The interpreter fails to evaluate certain expressions that require careful handling of variable scopes.
- Expressions that should reduce to normal form remain partially evaluated.
- The interpreter does not accurately reflect the semantics of lambda calculus, limiting its usefulness for complex expressions.

Reflections and Insights

This exercise highlighted the importance of correctly implementing capture-avoiding substitution in lambda calculus interpreters. Mismanagement of variable scopes can lead to incorrect evaluations and hinder the reduction of expressions to normal form.

This weeks discord question is:

> "What are the implications of untyped vs. typed lambda calculus on interpreter design?"

9 Week 9: Evaluating Expressions and Tracing the Interpreter

In Week 9, we focused on evaluating specific lambda expressions using the interpreter and tracing its evaluation strategy, particularly the recursive calls to `evaluate()` and `substitute()` functions.

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

We evaluated the expression:

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

Following the interpreter's steps precisely, we performed each substitution line by line.

Step-by-Step Evaluation

1. **Initial Expression:**

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

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

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

   The variable m is replaced with $\lambda f.\ \lambda x.\ f(f\ x)$.

3. **Second Application:** Apply the result to $\lambda f.\ \lambda x.\ f(f(f\ x))$.

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

   The variable n is replaced with $\lambda f.\ \lambda x.\ f(f(f\ x))$.

4. **Third Application:** Apply $\lambda f.\ \lambda x.\ f(f\ x)$ to $\lambda f.\ \lambda x.\ f(f(f\ x))$.

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

Tracing Recursive Calls in the Interpreter

To understand the evaluation strategy, we traced the recursive calls to `evaluate()` and `substitute()` functions using the expression:

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

Trace of Recursive Calls

We recorded the calls to `evaluate()` and `substitute()`, including line numbers from the interpreter code.

1. **Top-Level Evaluation:** `evaluate()` is called with the entire expression.
2. **Evaluating the Left Application:** `evaluate()` is called on the left part $(\lambda m.\ \lambda n.\ m\ n)\ (\lambda f.\ \lambda x.\ f(f\ x))$.
3. **Beta Reduction:** Since the left part is a lambda abstraction, we perform beta reduction by calling `substitute()` to replace m with $\lambda f.\ \lambda x.\ f(f\ x)$.
4. **Capture-Avoiding Substitution:** `substitute()` generates fresh variable names (e.g., `Var1`) to avoid variable capture during substitution.
5. **Evaluating the Resulting Expression:** `evaluate()` is called on the substituted expression $\lambda n.\ (\lambda f.\ \lambda x.\ f(f\ x))\ n$.
6. **Second Beta Reduction:** We perform beta reduction again by substituting n with $\lambda f.\ \lambda x.\ f\ x$, generating fresh variables as needed.
7. **Recursive Calls:** The interpreter continues making recursive calls to `evaluate()` and `substitute()`, building up the final expression.
8. **Final Result:** $\lambda \text{Var5}.\ (\lambda f.\ (\lambda x.\ f\ x))\ ((\lambda f.\ (\lambda x.\ f\ x))\ \text{Var5})$

Interpreting the Trace

The trace shows how the interpreter:

- Uses fresh variable names to prevent variable capture.
- Recursively evaluates sub-expressions.
- Stops prematurely after giving new variables, doesn't continue with the reduction

Reflections and Insights

This week's exercises deepened our understanding of how lambda calculus expressions are evaluated in practice.

Discussion Question

For this week's discussion, we considered:

> "What are the implications of evaluation strategies (e.g., normal order vs. applicative order) on the termination and performance of lambda calculus interpreters?"

—

Conclusion

In week one, we reviewed our discrete math knowledge and began coding proofs. Week two we saw the relationship between mathematical proofs and Lean code proofs, and then we began to solve problems recursively. In week three, I took a dive into quantum computing languages in my literature review while also working on coding a calculator using a recursive approach to solve parentheses. In week four, we explored parsing and context-free grammars, learning how to translate concrete syntax into abstract syntax, which is crucial for understanding how programming languages are processed. Week 5 used Lean as a way of proving logic puzzles. Week 6 built upon our logical foundations by focusing on implication in Lean Logic, proving logical statements. For week 7 we began to use lamda calculus to begin creating functions that could perform addition and multiplication. For weeks 8 and 9 we tackled a broken lambda calculus interpreter by identifying it's capture avoiding issues via the debugger.

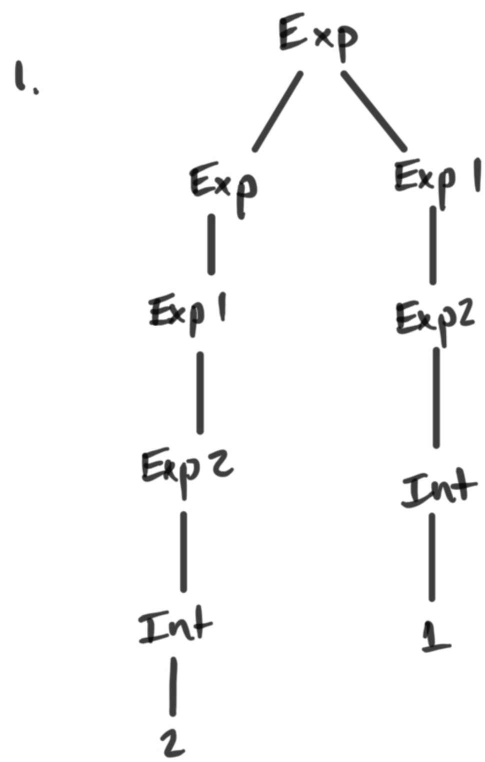

Figure 1: Derivation Tree for Expression 1

2. $1+2\times3$

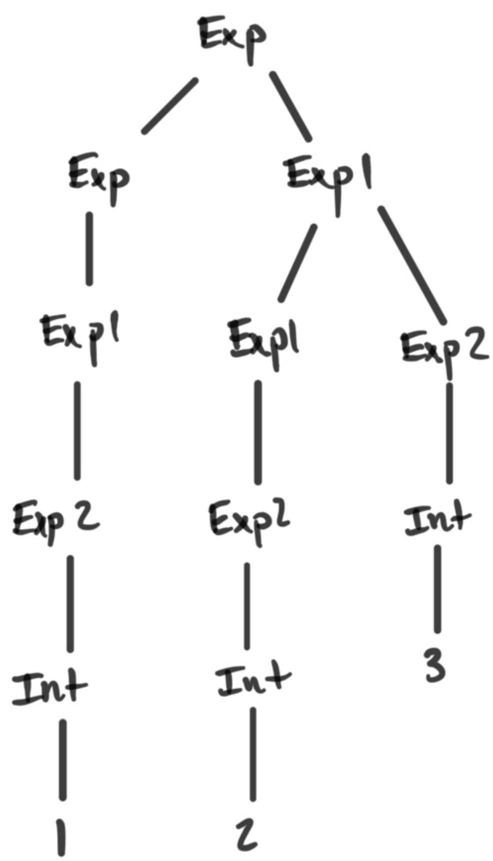

Figure 2: Derivation Tree for Expression 2

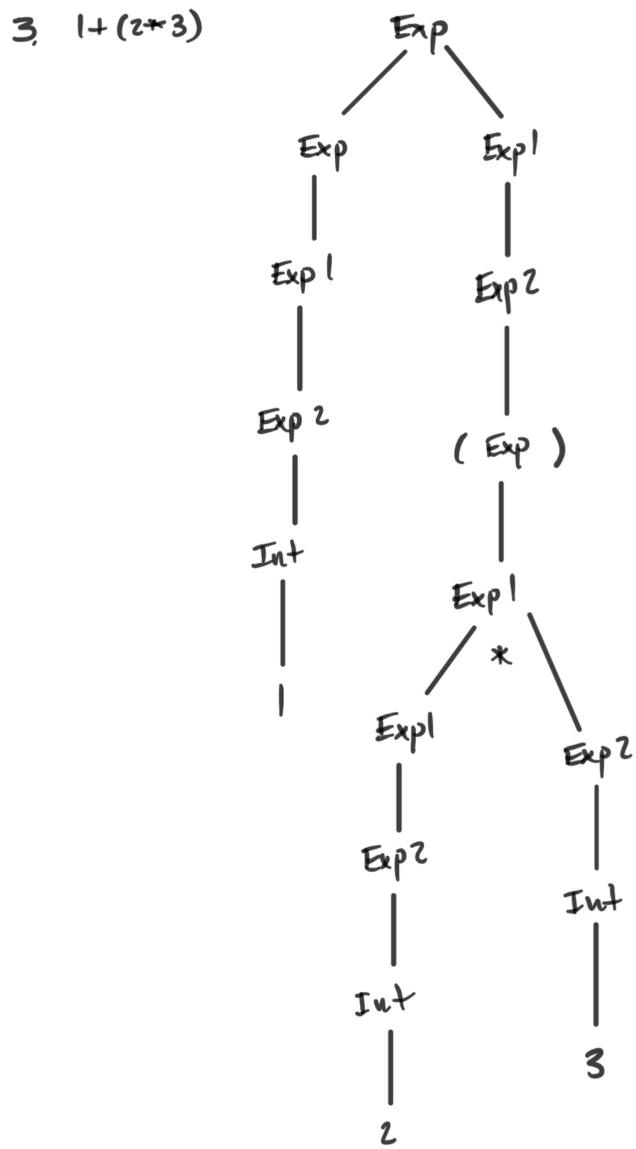

Figure 3: Derivation Tree for Expression 3

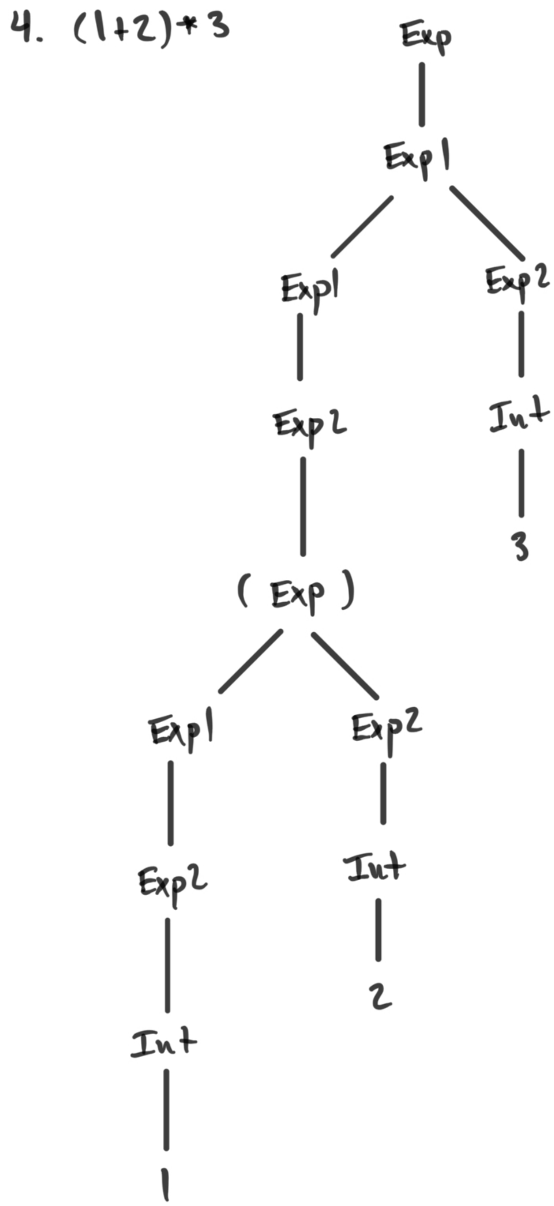

Figure 4: Derivation Tree for Expression 4

5.

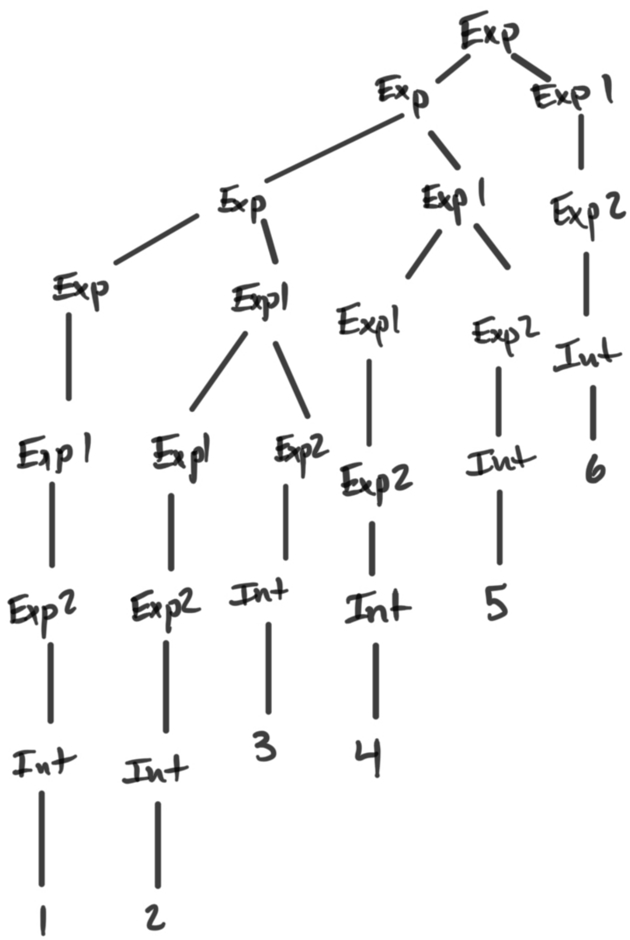

Figure 5: Derivation Tree for Expression 5