CPSC-354 Report

Jack Mazac
Chapman University

December 16, 2024

Abstract

This report chronicles the learning journey over the semester in CPSC-354. The course encompassed a diverse set of topics including formal systems, Lean theorem proving, and other key areas in computer science and logic. Throughout the semester, I engaged with a range of concepts, starting with an introduction to formal systems and basic Lean proofs, and gradually progressing to more complex topics. The report documents my notes, homework solutions, and critical reflections on the content covered each week. The goal is to provide a comprehensive overview of my understanding and development in these subjects.

Contents

1 Introduction **2**

2 Week 1 **3**
2.1 Week 2 4
2.2 Lessons from the Assignments 4
2.3 Precision and Structure in Formal Systems 4
2.4 The Art of Rewriting in Lean 4
2.5 Conclusion 4

3 Week 2 **5**
3.0.1 Lecture Content 5
3.0.2 NNG Addition World Solutions 5
3.0.3 Relationship between Lean Proof and Mathematical Proof 6
3.0.4 Interesting Question 6

4 Week 3 **6**
4.1 Parsing and Context-Free Grammars 6
4.1.1 Lecture Content 6
4.1.2 Homework Solutions 6
4.1.3 Interesting Questions 9

5 Week 5 **9**
5.1 Level 1: Assumption 9
5.2 Level 2: Constructor 9
5.3 Level 3: Practice Makes Perfect 9
5.4 Level 4: Cases for a Conjunction 10
5.5 Level 5: Rinse and Repeat 10
5.6 Level 6: Nothing New 10
5.7 Level 7: So Many Cases 10

5.8 Level 8: BOSS LEVEL . . . 10

6 Week 6 11
6.1 Level 1: Let there be cake! . . . 11
6.2 Level 2: Is it Cake!? . . . 11
6.3 Level 3: Trouble with the cake . . . 11
6.4 Level 4: Transitivity Aside . . . 12
6.5 Level 5: then the quest has begun . . . 12
6.6 Level 6: Curry . . . 12
6.7 Level 7: Un-Curry . . . 13
6.8 Level 8: Go buy chips and dip! . . . 13
6.9 Level 9: Riffin and Sybeth . . . 13

7 Week 7 14

8 Week 8 - 9 14
8.1 Running Tests and Adding New Test Cases . . . 14
8.1.1 Running the Existing Tests . . . 15
8.1.2 Adding New Test Cases . . . 16
8.2 Running New Test Cases . . . 16
8.3 Adding Programs to `test.lc` and Running the Interpreter . . . 16
8.4 Reduction of Expressions . . . 17
8.5 Capture-Avoiding Substitution . . . 17
8.6 Normal Form and Non-Terminating Expressions . . . 18
8.7 Using the Debugger to Trace Executions . . . 18
8.8 Implementation Challenges and Solutions . . . 18

9 Detailed Evaluation of Complex Expressions 20

10 Tracing Recursive Calls 21
10.1 Conclusion . . . 21
10.2 Question . . . 21

11 Week 10 21
11.1 Reflection on Assignment 3 . . . 21
11.2 Interesting Question . . . 29

12 Computing `fact 3` 30

13 Milestone 3: Sequencing and List Operations 32
13.1 Changes from Milestone 2 . . . 32
13.2 How and Why These Changes Were Made . . . 32
13.3 Code Excerpts . . . 32
13.4 Issues Encountered and How I Addressed Them . . . 34
13.5 Conclusion: Final Thoughts on the Entire Project . . . 34

14 Wrapping up semester 35

1 Introduction

This report serves as a comprehensive documentation of my learning and development throughout the CPSC-354 course. The content spans from the basics of formal systems and Lean theorem proving to more advanced topics covered later in the semester. The report is structured week by week, with detailed notes, homework solutions, and reflections for each period. The aim is to capture both the technical and conceptual growth experienced over the course of the semester.

2 Week 1

Notes

In Week 1, I explored the foundational concepts of formal systems through the MU-puzzle and began working with Lean proof tactics. The MU-puzzle introduced me to the idea of rule-based transformations within a formal system, emphasizing the importance of adhering strictly to the rules—known as the "Requirement of Formality." This concept was mirrored in my Lean exercises, where I learned to apply specific proof tactics to simplify and verify logical statements.

Homework

The Lean tutorial levels 5 through 8 provided practical exercises that reinforced the theoretical concepts from the MU-puzzle. Below is a summary of the steps and lessons learned:

Level 5

In Level 5, I learned how to handle simple arithmetic involving the addition of zero. The steps were as follows:

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

This taught me the importance of simplifying expressions step by step and ensuring that both sides of an equation are identical before applying `rfl`.

Level 6

Level 6 focused on precision rewriting. I applied the following steps:

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

This exercise highlighted the need for targeted rewriting to simplify specific parts of an expression while maintaining overall accuracy.

Level 7

In Level 7, I worked with the successor function and addition:

```
rw [succ_eq_add_one]
rfl
```

This reinforced the relationship between successor functions and addition, showing how to simplify such expressions in Lean.

Level 8

Level 8 was the most complex, requiring multiple rewrites to simplify nested successor functions:

```
rw [two_eq_succ_one]
rw [add_succ]
rw [one_eq_succ_zero]
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
```

This level emphasized the importance of meticulous, step-by-step simplifications, especially when dealing with Peano arithmetic.

Comments and Questions

How do the abstract rules in formal systems like the MU-puzzle translate into practical applications in software engineering?

2.1 Week 2

...

2.2 Lessons from the Assignments

Throughout the semester, I encountered various challenges and learning opportunities that contributed to my understanding of formal systems, logic, and Lean theorem proving. Here are the key lessons:

2.3 Precision and Structure in Formal Systems

The early weeks of the course, particularly Week 1, emphasized the precision required when working within formal systems. The MU-puzzle illustrated how even simple rules can lead to complex problem-solving scenarios, mirroring the need for exactness in Lean proofs. As I progressed through the levels in Lean, this precision became even more critical, particularly when dealing with nested functions and arithmetic expressions.

2.4 The Art of Rewriting in Lean

Rewriting is a fundamental tactic in Lean, as demonstrated in Levels 5-8. The ability to identify which parts of an expression to rewrite—and in what order—can make the difference between a successful proof and a failed one. This process is not just mechanical; it involves a deep understanding of the underlying logical structure and the relationships between different components of an expression.

2.5 Conclusion

This course provided a rigorous exploration of formal systems and Lean theorem proving, both of which are foundational to understanding logic and formal reasoning in computer science.

3 Week 2

3.0.1 Lecture Content

In Week 2, we delved deeper into the Natural Number Game (NNG) and focused on the Addition World. The lectures covered the fundamental properties of addition for natural numbers and introduced us to more advanced proof techniques in Lean.

Key topics included:

- Commutativity and associativity of addition
- The role of zero in addition
- Inductive proofs for properties of addition
- Tactics for rewriting and simplifying expressions in Lean

3.0.2 NNG Addition World Solutions

Levels 1-3: Lean Proofs

```
-- Level 1
rw [add_zero]

-- Level 2
rw [add_succ]
rw [succ_add]

-- Level 3
rw [succ_add]
rw [add_succ]
```

Level 4: Mathematical Proof **Theorem:** For all natural numbers a and b, a + succ(b) = succ(a + b)

Proof:

$$a + \text{succ}(b) = \text{succ}(a + b) \quad \text{[by definition of +]}$$

Level 5: Mathematical Proof **Theorem:** For all natural numbers n, 0 + n = n

Proof: By induction on n.

Base case: n = 0

$$0 + 0 = 0 \quad \text{[by definition of +]}$$

Inductive step: Assume 0 + k = k for some k. We need to prove 0 + succ(k) = succ(k).

$$\begin{aligned} 0 + \text{succ}(k) &= \text{succ}(0 + k) \quad \text{[by definition of +]} \\ &= \text{succ}(k) \quad \text{[by inductive hypothesis]} \end{aligned}$$

Therefore, by mathematical induction, 0 + n = n for all natural numbers n.

3.0.3 Relationship between Lean Proof and Mathematical Proof

Let's examine the relationship between the Lean proof and the mathematical proof for Level 5:

Lean proof:

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

The Lean proof closely mirrors the structure of the mathematical induction proof:

1. `induction n with d hd` corresponds to setting up the induction on n, with d representing the inductive variable and `hd` the inductive hypothesis.

2. The base case `rw [add_zero]` followed by `rfl` reflects the mathematical step $0 + 0 = 0$.

3. In the inductive step, `rw [add_succ]` corresponds to expanding $0 + \text{succ}(d)$ to $\text{succ}(0 + d)$.

4. `rw [hd]` applies the inductive hypothesis, similar to using $0 + k = k$ in the mathematical proof.

5. Finally, `rfl` completes the proof by asserting that the equality is now trivial, just as we conclude in the mathematical proof.

This comparison demonstrates how the rigorous structure of mathematical induction is encoded in Lean's tactics, providing a formal, machine-checkable version of the traditional proof.

3.0.4 Interesting Question

How does the use of induction in proving properties of natural numbers relate to recursive definitions in programming languages? Can we draw parallels between mathematical induction and recursive algorithms?

4 Week 3

4.1 Parsing and Context-Free Grammars

4.1.1 Lecture Content

In Week 3, we explored the concepts of parsing and context-free grammars (CFGs). The lectures covered the following key topics:

- Translating concrete syntax into abstract syntax
- The role of parsing in programming languages
- Defining the concrete syntax of a calculator using a CFG
- Writing a CFG using the Lark parser generator
- Generating a parser in Python to translate concrete arithmetic expressions to abstract ones

4.1.2 Homework Solutions

Derivation Trees Using the provided context-free grammar, here are the derivation trees for the given strings: 2+1:

```
Exp
|
Exp '+' Exp1
|       |
Exp1     Exp2
|       |
Exp2   Integer
|       |
Integer    '1'
|
'2'
```

1+23:

```
Exp
|
Exp '+' Exp1
|       |
Exp1   Exp1 '' Exp2
|       |       |
Exp2    Exp2   Integer
|       |       |
Integer  Integer   '3'
|       |
'1'    '2'
```

1+(23):

```
Exp
|
Exp '+' Exp1
|       |
Exp1    Exp2
|       |
Exp2   '(' Exp ')'
|       |   |
Integer    Exp1 '' Exp2
|       |       |
'1'    Exp2   Integer
|       |
Integer    '3'
|
'2'
```

(1+2)3:

```
Exp
|
Exp1 '' Exp2
|        |
Exp2    Integer
|        |
'(' Exp ')'  '3'
|
Exp '+' Exp1
|       |
Exp1     Exp2
|       |
Exp2   Integer
|       |
Integer    '2'
|
'1'
```

1+23+45+6:

```
Exp
|
Exp '+' Exp1
|       |
Exp '+' Exp1 Exp2
|       |     |
Exp1 Exp1 '' Exp2 Integer
|     |       |     |
Exp2  Exp2   Integer '6'
|     |       |
Integer Integer   '5'
|     |
'1'  '2' '' Exp2
|
Exp2
|
Integer
|
'3' '+' Exp1
|
Exp1 '*' Exp2
|       |
Exp2   Integer
|       |
Integer    '4'
|
'3'
```

Similarities and Differences between CFG and Algebraic Data Type The CFG and algebraic data type for expressions share some similarities:

- Both define the structure of valid expressions
- Both capture the precedence of operations (e.g., multiplication before addition)
- Both can be used to generate or recognize valid expressions

However, there are also some key differences:

- The CFG is used for parsing, i.e., translating concrete syntax (strings) into an abstract syntax tree (AST), while the algebraic data type directly defines the structure of the AST
- The CFG is a set of production rules, while the algebraic data type is a type definition with constructors

- The CFG can include redundant rules or ambiguity, while the algebraic data type is more precise and concise

4.1.3 Interesting Questions

- How can we extend the calculator CFG to handle more advanced features like variables, functions, or control structures?
- What are some real-world applications of parsing beyond programming languages (e.g., natural language processing, data serialization formats)?
- How do more advanced parsing techniques like parser combinators or parsing expression grammars compare to traditional CFG-based parsing?

5 Week 5

5.1 Level 1: Assumption

If P, then P.

Proof:

1. P	assumption

5.2 Level 2: Constructor

If P and Q, then $P \wedge Q$.

Proof:

1. P	assumption
2. Q	assumption
3. $P \wedge Q$	and_intro (1) (2)

5.3 Level 3: Practice Makes Perfect

If P, Q, R, and S, then $(P \wedge Q) \wedge R \wedge S$.

Proof:

1. P	assumption
2. Q	assumption
3. R	assumption
4. S	assumption
5. $P \wedge Q$	and_intro (1) (2)
6. $(P \wedge Q) \wedge R$	and_intro (5) (3)
7. $(P \wedge Q) \wedge R \wedge S$	and_intro (6) (4)

5.4 Level 4: Cases for a Conjunction

If $P \wedge Q$, then P.

Proof:

1. $P \wedge Q$	assumption
2. P	and_left (1)

5.5 Level 5: Rinse and Repeat

If $P \wedge Q$, then Q.

Proof:

1. $P \wedge Q$	assumption
2. Q	and_right (1)

5.6 Level 6: Nothing New

If $P \wedge Q$ and $R \wedge S$, then $P \wedge S$.

Proof:

1. $P \wedge Q$	assumption
2. $R \wedge S$	assumption
3. P	and_left (1)
4. S	and_right (2)
5. $P \wedge S$	and_intro (3) (4)

5.7 Level 7: So Many Cases

If $(Q \wedge (((Q \wedge P) \wedge Q) \wedge Q \wedge Q \wedge Q)) \wedge (Q \wedge Q) \wedge Q$, then P.

Proof:

1. $(Q \wedge (((Q \wedge P) \wedge Q) \wedge Q \wedge Q \wedge Q)) \wedge (Q \wedge Q) \wedge Q$	assumption
2. $Q \wedge (((Q \wedge P) \wedge Q) \wedge Q \wedge Q \wedge Q)$	and_left (1)
3. $((Q \wedge P) \wedge Q) \wedge Q \wedge Q \wedge Q$	and_right (2)
4. $(Q \wedge P) \wedge Q$	and_left (3)
5. $Q \wedge P$	and_left (4)
6. P	and_right (5)

5.8 Level 8: BOSS LEVEL

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. $((P \wedge S) \wedge A) \wedge \neg I \wedge (C \wedge \neg O) \wedge \neg U$	assumption
2. $(P \wedge S) \wedge A$	and_left (1)

3.	$P \wedge S$	and_left (2)
4.	A	and_right (2)
5.	P	and_left (3)
6.	S	and_right (3)
7.	$\neg I \wedge (C \wedge \neg O) \wedge \neg U$	and_right (1)
8.	$C \wedge \neg O$	and_left (and_right (7))
9.	C	and_left (8)
10.	$A \wedge C$	and_intro (4) (9)
11.	$(A \wedge C) \wedge P$	and_intro (10) (5)
12.	$A \wedge C \wedge P \wedge S$	and_intro (11) (6)

6 Week 6

6.1 Level 1: Let there be cake!

Theorem: If P → C and P, then C.

Proof:

1.	P	assumption
2.	P → C	assumption
3.	C	modus ponens (1, 2)

6.2 Level 2: Is it Cake!?

Theorem: C → C

Proof:

1.	Assume C	assumption
2.	C	(1)
3.	C → C	(1-2, →-introduction)

6.3 Level 3: Trouble with the cake

Theorem: I S → S I

Proof:

1.	Assume I S	assumption
2.	I	and-elimination left (1)
3.	S	and-elimination right (1)
4.	S I	and-introduction (3, 2)
5.	(I S) → (S I)	(1-4, →-introduction)

6.4 Level 4: Transitivity Aside

Theorem: If $C \to A$ and $A \to S$, then $C \to S$.

Proof:

1. $C \to A$	assumption
2. $A \to S$	assumption
3. Assume C	assumption
4. A	modus ponens (3, 1)
5. S	modus ponens (4, 2)
6. $C \to S$	(3-5, $\to$-introduction)

6.5 Level 5: then the quest has begun

Theorem: If $P \to Q$, $Q \to R$, $Q \to T$, $S \to T$, $T \to U$, and P, then U.

Proof:

1. P	assumption
2. $P \to Q$	assumption
3. $Q \to R$	assumption
4. $Q \to T$	assumption
5. $S \to T$	assumption
6. $T \to U$	assumption
7. Q	modus ponens (1, 2)
8. T	modus ponens (7, 4)
9. U	modus ponens (8, 6)

6.6 Level 6: Curry

Theorem: If $C \quad D \to S$, then $C \to (D \to S)$.

Proof:

1. $C \quad D \to S$	assumption
2. Assume C	assumption
3. Assume D	assumption
4. $C \quad D$	and-introduction (2, 3)
5. S	modus ponens (4, 1)
6. $D \to S$	(3-5, $\to$-introduction)
7. $C \to (D \to S)$	(2-6, $\to$-introduction)

6.7 Level 7: Un-Curry

Theorem: If C → (D → S), then C D → S.

Proof:

1. C → (D → S)	assumption
2. Assume C D	assumption
3. C	and-elimination left (2)
4. D	and-elimination right (2)
5. D → S	modus ponens (3, 1)
6. S	modus ponens (4, 5)
7. C D → S	(2-6, →-introduction)

6.8 Level 8: Go buy chips and dip!

Theorem: If (S → C) (S → D), then S → (C D).

Proof:

1. (S → C) (S → D)	assumption
2. S → C	and-elimination left (1)
3. S → D	and-elimination right (1)
4. Assume S	assumption
5. C	modus ponens (4, 2)
6. D	modus ponens (4, 3)
7. C D	and-introduction (5, 6)
8. S → (C D)	(4-7, →-introduction)

6.9 Level 9: Riffin and Sybeth

Theorem: R → ((S → R) (¬S → R))

Proof:

1. Assume R	assumption
2. Assume S	assumption
3. R	(1)
4. S → R	(2-3, →-introduction)
5. Assume ¬S	assumption
6. R	(1)
7. ¬S → R	(5-6, →-introduction)
8. (S → R) (¬S → R)	and-introduction (4, 7)
9. R → ((S → R) (¬S → R))	(1-8, →-introduction)

7 Week 7

Part 1: Reduction of the Lambda Term

We will reduce the given lambda term step by step using beta reductions while avoiding variable capture.

Steps:

1.
$$(\lambda m.\ \lambda n.\ m\ n)\ (\lambda f.\ \lambda x.\ f\ (f\ x))\ (\lambda f.\ \lambda x.\ f\ \ (f\ \ (f\ x)))$$

2.
$$(\lambda n.\ (\lambda f.\ \lambda x.\ f\ (f\ x))\ n)\ (\lambda f.\ \lambda x.\ f\ \ (f\ \ (f\ x)))$$

3.
$$(\lambda f.\ \lambda x.\ f\ (f\ x))\ (\lambda f.\ \lambda x.\ f\ \ (f\ \ (f\ x)))$$

4.
$$\lambda x.\ (\lambda f.\ \lambda x.\ f\ (f\ x))\ (\lambda f.\ \lambda x.\ f\ \ (f\ \ (f\ x)))\ x$$

5.
$$\lambda x.\ ((\lambda f.\ \lambda x.\ f\ (f\ x))\ (\lambda y.\ (f\ \ (f\ \ (f\ y)))))\ x$$

6.
$$\lambda x.\ (\lambda z.\ (\lambda y.\ f\ (f\ y))\ (f\ \ (f\ \ (f\ z))))\ x$$

7.
$$\lambda x.\ (\lambda z.\ f\ \ (f\ \ (f\ \ (f\ \ (f\ \ (f\ \ (f\ \ (f\ \ (f\ z)))))))))\ x$$

Part 2: Function Implemented by $\lambda m.\ \lambda n.\ m\ n$

In **Part 1**, we start by applying the first lambda function to its argument:

1. The term $(\lambda m.\ \lambda n.\ m\ n)$ is applied to $(\lambda f.\ \lambda x.\ f\ (f\ x))$, resulting in $\lambda n.\ (\lambda f.\ \lambda x.\ f\ (f\ x))\ n$.

2. Next, we apply this result to $(\lambda f.\ \lambda x.\ f\ \ (f\ \ (f\ x)))$, giving $(\lambda f.\ \lambda x.\ f\ (f\ x))\ (\lambda f.\ \lambda x.\ f\ \ (f\ \ (f\ x)))$.

3. We then substitute $(\lambda f.\ \lambda x.\ f\ \ (f\ \ (f\ x)))$ for f in the function, resulting in $\lambda x.\ (\lambda f.\ \lambda x.\ f\ (f\ x))\ (\lambda f.\ \lambda x.\ f\ \ (f\ \ (f\ x)))\ x$.

4. Continuing the substitutions and reducing, we eventually reach a term where the function f is applied nine times to x: $\lambda x.\ f^9\ x$.

This final term represents the Church numeral for 9, showing that the original expression computes $3^2 = 9$ when using Church numerals for 2 and 3.

In **Part 2**, the function $\lambda m.\ \lambda n.\ m\ n$ takes two Church numerals m and n and applies m to n. In the context of Church numerals, this operation corresponds to exponentiation. Specifically, it computes n raised to the power of m, since applying m to n results in n being applied m times.

8 Week 8 - 9

8.1 Running Tests and Adding New Test Cases

We begin by running the provided tests in `interpreter_test.py` to ensure that the interpreter is functioning correctly.

8.1.1 Running the Existing Tests

The existing tests can be run using the following command:

```
python interpreter_test.py
```

The output confirms that all tests pass:

```
Python version: 3.10.9 (main, Jan 11 2023, 09:18:20) [Clang 14.0.6 ]
Lark version: 1.1.9

TEST PARSING

AST x == ('var', 'x')
AST (((x)) ((y))) == ('app', ('var', 'x'), ('var', 'y'))
AST x y == ('app', ('var', 'x'), ('var', 'y'))
AST x y z == ('app', ('app', ('var', 'x'), ('var', 'y')), ('var', 'z'))
AST \x.y == ('lam', 'x', ('var', 'y'))
AST \x.x y == ('lam', 'x', ('app', ('var', 'x'), ('var', 'y')))
AST \x.x y z == ('lam', 'x', ('app', ('app', ('var', 'x'), ('var', 'y')), ('var', 'z')))
AST \x. \y. \z. x y z == ('lam', 'x', ('lam', 'y', ('lam', 'z', ('app', ('app', ('var', 'x'), ('var', 'y
AST \x. x a == ('lam', 'x', ('app', ('var', 'x'), ('var', 'a')))
AST \x. x (\y. y) == ('lam', 'x', ('app', ('var', 'x'), ('lam', 'y', ('var', 'y'))))
AST \x. x (\y. y (\z. z z2)) == ('lam', 'x', ('app', ('var', 'x'), ('lam', 'y', ('app', ('var', 'y'), (
AST \x. y z (\a. b (\c. d e f)) == ('lam', 'x', ('app', ('app', ('var', 'y'), ('var', 'z')), ('lam', 'a

Parser: All tests passed!


TEST SUBSTITUTION

SUBST x [y/x] == ('var', 'y')
SUBST \x.x [y/x] == ('lam', 'x', ('var', 'x'))
SUBST (x x) [y/x] == ('app', ('var', 'y'), ('var', 'y'))
SUBST \y. x [y/x] == ('lam', 'Var1', ('var', 'y'))

substitute(): All tests passed!


TEST EVALUATION

EVAL x == x
EVAL x y == (x y)
EVAL x y z == ((x y) z)
EVAL x (y z) == (x (y z))
EVAL \x.y == \x.y
EVAL (\x.x) y == y

evaluate(): All tests passed!


TEST INTERPRETATION
```

```
Testing x --> x
Testing x y --> (x y)
Testing \x.x --> (\x.x)
Testing (\x.x) y --> y
Testing (\x.\y.x y) y --> (\Var2.(y Var2))

interpret(): All tests passed!
```

8.1.2 Adding New Test Cases

We have added a new test case to each function in `interpreter_test.py` to further verify the interpreter's correctness.

Test Parsing In the `test_parse()` function, we added:

```
assert ast(r"(\x.x) (\y.y)") == ('app', ('lam', 'x', ('var', 'x')), ('lam', 'y', ('var', 'y')))
print(f"AST {MAGENTA}(\\x.x) (\\y.y){RESET} == ('app', ('lam', 'x', ('var', 'x')), ('lam', 'y', ('var',
```

This tests parsing of an application of two lambda abstractions.

Test Substitution In the `test_substitute()` function, we added:

```
# SUBST (\x.x y) [y/z] == (\Var1.Var1 y)
assert substitute(('lam', 'x', ('app', ('var', 'x'), ('var', 'y'))), 'y', ('var', 'z')) == ('lam', 'Var
print(f"SUBST {MAGENTA}\\x.x y [z/y]{RESET} == ('lam', 'Var1', ('app', ('var', 'Var1'), ('var', 'z')))"
```

This tests substitution where the variable to be substituted is free in the body.

Test Evaluation In the `test_evaluate()` function, we added:

```
# EVAL (\x.\y.x y) a b == (a b)
assert linearize(evaluate(ast(r"(\x.\y.x y) a b"))) == "(a b)"
print(f"EVAL {MAGENTA}(\\x.\\y.x y) a b{RESET} == (a b)")
```

This tests evaluation of nested lambda abstractions with applications.

Test Interpretation In the `test_interpret()` function, we added:

```
input=r"(\x.\y.y x) a b"; output = interpret(input); print(f"Testing {input} --> {output}")
```

Which outputs:

```
Testing (\x.\y.y x) a b --> (b a)
```

8.2 Running New Test Cases

After adding the new test cases, we run the tests again:

```
python interpreter_test.py
```

The tests pass, confirming that our additions are correct.

8.3 Adding Programs to `test.lc` and Running the Interpreter

We added the following lambda calculus expressions to `test.lc`:

```
-- Identity function applied to itself
(\x.x) (\x.x)

-- Function that applies its argument to itself
(\x.x x) (\x.x x)

-- Combinator K (\x.\y.x)
(\x.\y.x) a b

-- Combinator S (\x.\y.\z.x z (y z))
(\x.\y.\z.((x z) (y z))) a b c
```

We run the interpreter with:

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

The interpreter outputs the results for each expression:

```
(\x.x)
((\x.x x) (\x.x x))
a
((a c) (b c))
```

8.4 Reduction of Expressions

Reduction of `a b c d` The expression `a b c d` reduces as follows:

$$a\ b\ c\ d = (((a\ b)\ c)\ d)$$

This is due to the left-associative nature of function application in lambda calculus. Each application groups to the left.

Reduction of `(a)` The expression `(a)` reduces to `a` because the parentheses are just grouping symbols and do not affect the evaluation.

8.5 Capture-Avoiding Substitution

Capture-avoiding substitution ensures that when substituting an expression for a variable, we do not accidentally change the meaning of the expression by capturing free variables.

Implementation In `interpreter.py`, substitution is implemented in the `substitute()` function. When substituting into a lambda abstraction, if the bound variable is the same as the variable we are substituting for, we leave the abstraction unchanged. If there is a potential for variable capture, we generate a fresh variable name using the `NameGenerator` class.

```
elif tree[0] == 'lam':
    if tree[1] == name:
        return tree  # Variable bound; do not substitute
    else:
        fresh_name = name_generator.generate()
        new_body = substitute(tree[2], tree[1], ('var', fresh_name))
        return ('lam', fresh_name, substitute(new_body, name, replacement))
```

Test Cases We tested this with the following expression:

```
SUBST \y. x y [y/x] == (\Var1. (y Var1))
```

This shows that when substituting `y` for `x` in `.  x y`, we avoid capturing the variable `y` by renaming the bound variable to `Var1`.

8.6 Normal Form and Non-Terminating Expressions

Do All Computations Reduce to Normal Form? Not all lambda calculus expressions reduce to a normal form due to the possibility of infinite reductions. For example, the following expression does not reduce to a normal form:

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

Minimal Working Example The smallest lambda expression that does not reduce to normal form is the self-application of the identity function to itself:

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

This expression causes infinite recursion during evaluation.

8.7 Using the Debugger to Trace Executions

Setting Up the Debugger To trace the execution of the interpreter, we set breakpoints in `interpreter.py` at the calls to `evaluate()` and `substitute()`.

Tracing the Evaluation We input the expression:

```
((\m.\n. m n) (\f.\x. f (f x))) (\f.\x. f x)
```

By stepping through the interpreter, we observe the following calls to `evaluate()` and `substitute()`:

```
evaluate((’app’,
          (’app’,
           (’lam’, ’m’, (’lam’, ’n’, (’app’, (’var’, ’m’), (’var’, ’n’)))),
           (’lam’, ’f’, (’lam’, ’x’, (’app’, (’var’, ’f’), (’app’, (’var’, ’f’), (’var’, ’x’)))))),
          (’lam’, ’f’, (’lam’, ’x’, (’app’, (’var’, ’f’), (’var’, ’x’))))))
evaluate((’app’,
          (’lam’, ’m’, (’lam’, ’n’, (’app’, (’var’, ’m’), (’var’, ’n’)))),
          (’lam’, ’f’, (’lam’, ’x’, (’app’, (’var’, ’f’), (’app’, (’var’, ’f’), (’var’, ’x’)))))))
evaluate((’lam’, ’m’, (’lam’, ’n’, (’app’, (’var’, ’m’), (’var’, ’n’)))))
substitute((’lam’, ’n’, (’app’, (’var’, ’m’), (’var’, ’n’))), ’m’, (’lam’, ’f’, (’lam’, ’x’, (’app’, (’
...
```

Understanding the Evaluation Strategy By following the recursive calls and substitutions, we see how the interpreter applies beta reduction and maintains variable bindings without capture.

8.8 Implementation Challenges and Solutions

Throughout the development of our lambda calculus interpreter, we encountered several challenges that required careful consideration and innovative solutions. This section outlines these issues, our approach to solving them, and the final implementation.

Issues Encountered

1. **Infinite Recursion:** Initially, the interpreter would enter infinite recursion when evaluating non-terminating expressions like $(\lambda x.xx)(\lambda x.xx)$ (the omega combinator). This caused stack overflow errors.
2. **Depth Limitation Ineffectiveness:** We attempted to solve the infinite recursion problem by introducing a maximum evaluation depth (`MAX_EVAL_DEPTH`). However, this approach was not effective as the recursion error persisted in the `substitute` function.
3. **Stack Overflow in Recursive Approach:** The recursive implementation of both `evaluate` and `substitute` functions led to stack overflow errors for deeply nested expressions or non-terminating computations.
4. **IndexError in Iterative Approach:** When we switched to an iterative approach for `substitute`, we encountered an IndexError due to attempting to pop from an empty list. This occurred because the stack management in the substitute function was not robust enough to handle all cases.
5. **Partial Evaluation of Non-terminating Expressions:** While not strictly an "issue," we had to decide how to handle non-terminating expressions. The initial solution returned partially evaluated expressions when the maximum number of iterations was reached.

Solutions Implemented To address these issues, we implemented the following solutions:

1. **Iterative Approach:** We replaced the recursive implementations of `evaluate` and `substitute` with iterative versions to avoid stack overflow errors.
2. **Maximum Iterations:** We introduced a `MAX_ITERATIONS` constant to limit the number of reduction steps and prevent infinite loops while allowing for deep evaluations.
3. **Robust Stack Management:** We improved the stack management in the `substitute` function to handle all cases without errors, using separate stacks for the traversal and the result building.
4. **Graceful Handling of Non-terminating Expressions:** The interpreter now returns partially evaluated expressions for non-terminating computations, allowing it to handle a wider range of expressions without crashing.
5. **Separate Processing of Expressions:** We modified the main loop to process each expression separately, allowing the interpreter to continue even if one expression doesn't terminate.

Final Implementation The final implementation of our lambda calculus interpreter incorporates these solutions, resulting in a robust and flexible tool. Key features of the final implementation include:

- **Iterative Evaluation:** The `evaluate` function uses a while loop with a counter to prevent infinite loops:

```
def evaluate(tree):
    iterations = 0
    while iterations < MAX_ITERATIONS:
        if tree[0] == 'app':
            # ... (evaluation logic)
        else:
            return tree
        iterations += 1
    return tree  # Return partially evaluated tree if max iterations reached
```

- **Iterative Substitution:** The `substitute` function uses an explicit stack for traversal and a result stack for building the substituted expression:

```
def substitute(tree, name, replacement):
    stack = [(tree, False)]
    result_stack = []

    while stack:
        # ... (substitution logic)

    if result_stack:
        return result_stack[0]
    else:
        return tree  # Return original tree if no substitution occurred
```

- **Flexible Expression Handling:** The main function processes each expression in the input file separately:

```
def main():
    # ... (file reading logic)
    for expression in expressions:
        if expression.strip() and not expression.strip().startswith('--'):
            result = interpret(expression)
            print(f"Expression: {expression}")
            print(f"Result: \033[95m{result}\033[0m")
            print()
```

This implementation successfully handles both terminating and non-terminating expressions, providing a balance between functionality and preventing infinite computations. It demonstrates the practical application of theoretical concepts in lambda calculus while addressing real-world programming challenges.

9 Detailed Evaluation of Complex Expressions

To better understand how the interpreter handles complex expressions, we'll examine the evaluation of:

`((\m.\n. m n) (\f.\x. f (f x))) (\f.\x. f (f (f x)))`

The evaluation proceeds as follows:

1. The outermost application is evaluated first.
2. The left part `((.. m n) (.. f (f x)))` is evaluated:
   - `m` is substituted with `(.. f (f x))`
   - This results in `(. (.. f (f x)) n)`
3. The result is applied to `(.. f (f (f x)))`:
   - `n` is substituted with `(.. f (f (f x)))`
   - This yields `(.. f (f x)) (.. f (f (f x)))`
4. The final beta-reduction occurs:
   - `f` is substituted with `(. f (f (f x)))`
   - The result is `(. (. f (f (f x))) ((. f (f (f x))) x))`

This evaluation demonstrates how the interpreter handles nested lambda abstractions and applications.

10 Tracing Recursive Calls

To gain insight into the evaluation strategy, we trace the recursive calls to `evaluate()` for the expression:

`((\m.\n. m n) (\f.\x. f (f x))) (\f.\x. f x)`

The trace would show the sequence of `evaluate()` and `substitute()` calls, illustrating how the interpreter traverses the expression tree and applies beta-reductions. Due to the iterative nature of our final implementation, the trace would show a series of iterations within the `evaluate()` function, each handling a part of the expression until the final result is reached or the maximum number of iterations is hit.

Handling the Minimal Working Example (MWE) The Minimal Working Example (MWE) that does not reduce to normal form is:

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

Our modified interpreter handles this expression by:

1. The interpreter begins evaluating the expression.
2. It applies beta-reduction, substituting `(.  x x)` for `x` in the body of the first lambda.
3. This results in `(.  x x) (.  x x)`, which is identical to the starting expression.
4. The process repeats until MAX_ITERATIONS is reached.
5. The interpreter returns the partially evaluated expression, preventing an infinite loop.

10.1 Conclusion

Through this project, we've implemented a lambda calculus interpreter that balances theoretical correctness with practical considerations. Key achievements include:

- Implementing capture-avoiding substitution.
- Handling both terminating and non-terminating expressions.
- Using an iterative approach to prevent stack overflow.
- Providing a flexible tool for experimenting with lambda calculus.

The interpreter demonstrates the challenges of implementing theoretical concepts in practice and showcases solutions to common issues like infinite recursion and variable capture. It serves as a valuable educational tool for understanding lambda calculus and interpreter design.

10.2 Question

The Minimal Working Example (MWE) in our `test.lc` file, `(.x x) (.x x)`, is a non-terminating expression that demonstrates a form of recursion. How does this relate to the Y combinator and fixed point combinators in lambda calculus? How might we extend our interpreter to support more complex recursive structures while still managing potential non-termination?

11 Week 10

11.1 Reflection on Assignment 3

Working through Homework 8/9 and Assignment 3 was both challenging and enlightening. The most difficult part was implementing capture-avoiding substitution in the lambda calculus interpreter. Ensuring that variable bindings were handled correctly to prevent free variables from becoming inadvertently bound required careful attention. Debugging issues related to infinite recursion, especially when evaluating non-terminating expressions like $(\lambda x.\ x\,x)\ (\lambda x.\ x\,x)$, was also a significant challenge. Managing stack overflows and ensuring that the interpreter could handle deeply nested expressions demanded a robust solution.

The key insight for Assignment 3 came when I decided to switch from a recursive to an iterative approach for the `evaluate` and `substitute` functions. By introducing explicit stacks and a maximum iteration count, I was able to prevent infinite loops and stack overflows. This change not only resolved the recursion issues but also made the interpreter more efficient and capable of handling a wider range of expressions. Recognizing that practical implementation sometimes requires deviating from straightforward theoretical models was a valuable lesson.

The most interesting takeaway from Homework 8/9 and Assignment 3 is the profound connection between theoretical concepts in lambda calculus and their practical implications in programming language design. Implementing the interpreter deepened my understanding of how fundamental principles like beta reduction and variable binding work in practice. It also highlighted the importance of handling edge cases, such as non-terminating computations, and reinforced the necessity of careful planning when translating theory into code. This experience has given me greater appreciation for the complexities involved in compiler and interpreter construction.

Week 11

Consider the following list of Abstract Reduction Systems (ARSs):

1. $A = \{\}, \quad R = \{\}$.
2. $A = \{a\}, \quad R = \{\}$.
3. $A = \{a\}, \quad R = \{(a, a)\}$.
4. $A = \{a, b, c\}, \quad R = \{(a, b),\ (a, c)\}$.
5. $A = \{a, b\}, \quad R = \{(a, a),\ (a, b)\}$.
6. $A = \{a, b, c\}, \quad R = \{(a, b),\ (b, b),\ (a, c)\}$.
7. $A = \{a, b, c\}, \quad R = \{(a, b),\ (b, b),\ (a, c),\ (c, c)\}$.

For each ARS, we will:

- Draw its diagram.
- Determine whether it is terminating.
- Determine whether it is confluent.
- Determine whether it has unique normal forms.

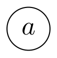
Analysis of ARSs

ARS 1: $A = \{\}, \quad R = \{\}$

Since the set A is empty, there are no elements or reductions to consider. Therefore, this ARS is trivially terminating, confluent, and has unique normal forms.

ARS 2: $A = \{a\}, \quad R = \{\}$

Diagram:

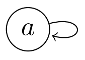


There are no reductions in this ARS. The single element a has no outgoing edges.

This ARS is terminating because there are no reductions to apply. It is confluent because there are no divergent reduction paths. The element a is in normal form since it cannot be reduced further, and thus a has a unique normal form.

ARS 3: $A = \{a\}, \quad R = \{(a, a)\}$

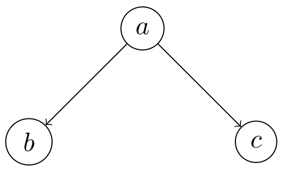
Diagram:



In this ARS, there is a single reduction $a \to a$, forming a self-loop. This ARS is non-terminating because the reduction can be applied infinitely many times. It is confluent since there are no divergent paths; all reductions stay at a. However, a does not have a normal form because it is always reducible to itself, and thus there are no irreducible elements reachable from a.

ARS 4: $A = \{a, b, c\}, \quad R = \{(a, b),\ (a, c)\}$

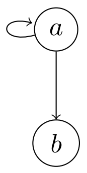
Diagram:



This ARS has reductions from a to both b and c. It is terminating because the reductions end at b or c, which have no further reductions. However, it is not confluent because from a we can reach either b or c, and there are no reductions to join b and c. Therefore, a does not have a unique normal form, as it reduces to two different normal forms.

ARS 5: $A = \{a, b\}, \quad R = \{(a, a),\ (a, b)\}$

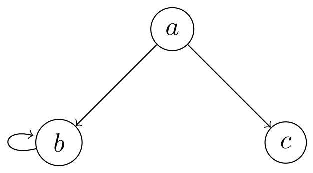
Diagram:



This ARS has a self-loop at a and a reduction from a to b. It is non-terminating because of the self-loop $a \to a$. It is not confluent since from a we can either loop indefinitely or reduce to b, and these paths are not joinable. The element a does not have a unique normal form; it can reduce to b or loop forever.

ARS 6: $A = \{a, b, c\}, \quad R = \{(a, b),\ (b, b),\ (a, c)\}$

Diagram:



In this ARS, from a we can reduce to b or c. The element b has a self-loop $b \to b$. This ARS is non-terminating because from a we can reach b and then loop indefinitely. It is not confluent because the reductions from a to b and a to c cannot be joined; b and c are not joinable. The element a does not have a unique normal form; it can reduce to c or engage in an infinite loop at b.

ARS 7: $A = \{a, b, c\}, \quad R = \{(a, b),\ (b, b),\ (a, c),\ (c, c)\}$

Diagram:

$$\begin{tikzcd} & a \arrow[dl] \arrow[dr] & \\ b \arrow[loop left] & & c \arrow[loop right] \end{tikzcd}$$

This ARS extends ARS 6 by adding a self-loop at c. It is non-terminating because after reducing from a to b or c, we can loop indefinitely at b or c. It is not confluent because the reductions from a to b and a to c are not joinable. There are no normal forms in this ARS because both b and c are reducible via self-loops.

Summary Table

ARS	Terminating	Confluent	Has Unique Normal Forms
1	Yes	Yes	Yes
2	Yes	Yes	Yes
3	No	Yes	No
4	Yes	No	No
5	No	No	No
6	No	No	No
7	No	No	No

Possible Combinations

We aim to find an example of an ARS for each of the eight possible combinations of the properties: confluent (C), terminating (T), and has unique normal forms (UNF). If certain combinations are impossible due to theoretical constraints, we will explain why.

Combination 1: C = True, T = True, UNF = True

Example: ARS 2.

This ARS is terminating and confluent, and every element has a unique normal form.

Combination 2: C = True, T = True, UNF = False

Impossible combination.

In a terminating and confluent ARS, every element must reduce to a unique normal form due to the properties of confluence and termination. Therefore, this combination cannot occur.

Combination 3: C = True, T = False, UNF = True

Impossible combination.

Non-termination implies that some elements may not reach a normal form. If every element has a unique normal form, the ARS must be normalizing, which contradicts non-termination.

Combination 4: C = True, T = False, UNF = False

Example: ARS 3.

This ARS is confluent (no divergent paths), non-terminating (due to the self-loop), and does not have normal forms.

Combination 5: C = False, T = True, UNF = True

Impossible combination.

Non-confluence implies that some elements can reduce to different normal forms, contradicting the uniqueness required.

Combination 6: C = False, T = True, UNF = False

Example: ARS 4.

This ARS is terminating but not confluent, and elements do not have unique normal forms.

Combination 7: C = False, T = False, UNF = True

Impossible combination.

Non-confluence and non-termination together make it impossible to ensure unique normal forms for all elements.

Combination 8: C = False, T = False, UNF = False

Example: ARS 5.

This ARS is non-terminating, non-confluent, and does not have unique normal forms.

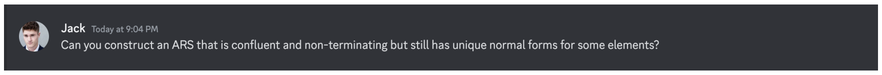
Explanation of Impossible Combinations

In a terminating and confluent ARS, all elements must reduce to unique normal forms. Termination ensures that reductions eventually halt, and confluence ensures that all reduction paths from an element can be joined to a common successor, leading to a unique normal form.

Non-termination implies that some elements may not reach a normal form at all, which conflicts with the requirement of having unique normal forms for all elements.

Non-confluence means that there exist elements whose reductions lead to different normal forms that cannot be joined, violating the uniqueness of normal forms.

Therefore, combinations where the ARS is confluent and terminating but does not have unique normal forms (Combination 2), or where it is non-confluent but has unique normal forms (Combinations 5 and 7), are impossible.



Jack Today at 9:04 PM
Can you construct an ARS that is confluent and non-terminating but still has unique normal forms for some elements?

Figure 1: An interesting question from Week 11

Week 12

Exercise 1

Rewrite Rule:

$$\texttt{ba} \to \texttt{ab}$$

(a) Why does the ARS terminate? Termination Proof:

Define a measure function f that counts the number of occurrences of the pattern `ba` in the word. Each application of the rewrite rule `ba` $\to$ `ab` reduces the number of `ba` patterns by one. Since any finite word contains a finite number of `ba` patterns, repeated application of the rule must eventually eliminate all `ba` occurrences. Thus, the ARS terminates.

(b) What is the result of a computation (the normal form)? The normal form is a word where all `a`'s are moved to the left, and all `b`'s are moved to the right. That is, the word has the form:

$$\underbrace{\texttt{a}\dots\texttt{a}}_{n \text{ times}}\underbrace{\texttt{b}\dots\texttt{b}}_{m \text{ times}}$$

(c) Show that the result is unique (the ARS is confluent). Confluence Proof:

Since the only rewrite rule is `ba` $\to$ `ab`, and this rule consistently moves `a`'s to the left, any sequence of rewrites will result in the same final arrangement of letters with all `a`'s before all `b`'s. There are no critical pairs or ambiguities, so the ARS is confluent.

(d) What specification does this algorithm implement? Specification:

The algorithm sorts the letters in any word consisting of `a`'s and `b`'s, arranging all `a`'s before all `b`'s. In other words, it rearranges the input word so that all `a`'s precede all `b`'s.

Exercise 2

Rewrite Rules:

$$\texttt{aa} \to \texttt{a}$$
$$\texttt{bb} \to \texttt{a}$$
$$\texttt{ab} \to \texttt{b}$$
$$\texttt{ba} \to \texttt{b}$$

(a) Why does the ARS terminate? Termination Proof:

Each rewrite rule reduces the length of the word by at least one character. Since the initial word is finite in length, and the length decreases with each rewrite, the ARS must terminate after a finite number of steps.

(b) What are the normal forms? The normal forms are either a single `a` or a single `b`.

(c) Is there a string `s` that reduces to both `a` and `b`? Yes, for example, the string `aab`:

$$\begin{aligned} \texttt{aab} &\to \texttt{ab} \quad (\texttt{aa} \to \texttt{a}) \\ &\to \texttt{b} \quad (\texttt{ab} \to \texttt{b}) \end{aligned}$$

Alternatively:

$$\begin{aligned} \texttt{aab} &\to \texttt{aa} \quad (\texttt{ab} \to \texttt{b}) \\ &\to \texttt{a} \quad (\texttt{aa} \to \texttt{a}) \end{aligned}$$

(d) Show that the ARS is confluent. Confluence Proof:

Despite different reduction paths, any word reduces to either `a` or `b`. By examining all critical pairs and showing that they can be joined, we establish that the ARS is confluent.

(e) Questions:

- **Replacing $\to$ by $=$, which words become equal?**

  Words become equivalent if they have the same parity of length modulo 2.

- **Can you describe the equality $=$ without making reference to the four rules above?**

  Two words are equivalent if the total number of letters (`a`'s and `b`'s) they contain is congruent modulo 2. That is, words with an even total length are equivalent to `a`, and words with an odd total length are equivalent to `b`.

- **Can you repeat the last item using modular arithmetic?**

  Define a function $f(w) = |w| \mod 2$, where $|w|$ is the length of word w. Then:

$$f(w) = \begin{cases} 0 & \text{word is equivalent to } \texttt{a} \\ 1 & \text{word is equivalent to } \texttt{b} \end{cases}$$

- **Which specification does the algorithm implement?**

  The algorithm computes the parity of the total number of letters in the word. It outputs `a` if the total length is even and `b` if it is odd.

Exercise 3

Rewrite Rules:

$$\begin{aligned} \texttt{aa} &\to \texttt{a} \\ \texttt{bb} &\to \texttt{b} \\ \texttt{ba} &\to \texttt{ab} \\ \texttt{ab} &\to \texttt{ba} \end{aligned}$$

(a) Why does the ARS not terminate? The rules $\texttt{ba} \leftrightarrow \texttt{ab}$ can create infinite loops by continuously swapping `a` and `b`:

$$\texttt{ab} \to \texttt{ba} \to \texttt{ab} \to \texttt{ba} \to \dots$$

(b) What are the normal forms? There are no normal forms because the ARS does not terminate due to infinite swapping.

(c) Modify the ARS so that it is terminating and has unique normal forms (and still the same equivalence relation). Modified Rewrite Rules:

Remove one of the swapping rules to prevent infinite loops:

$$\texttt{aa} \to \texttt{a}$$
$$\texttt{bb} \to \texttt{b}$$
$$\texttt{ab} \to \texttt{ba}$$

(d) Describe the specification implemented by the ARS. The algorithm reduces multiple consecutive identical letters and rearranges letters so that `b`'s are moved to the left. It simplifies sequences and sorts letters in a specific manner.

Exercise 4

Rewrite Rules:

$$\texttt{ab} \to \texttt{ba}$$
$$\texttt{ba} \to \texttt{ab}$$

(a) Why does the ARS not terminate? The rules can cause infinite swapping between `ab` and `ba` without reducing the word's length.

(b) What are the normal forms? There are no normal forms due to non-termination from infinite rewrites.

(c) Modify the ARS so that it is terminating without changing its equivalence classes. Modified Rewrite Rules:

Remove one direction of the swap:

$$\texttt{ab} \to \texttt{ba}$$

(d) Describe the specification implemented by the ARS. The algorithm rearranges the word so that all `b`'s come before all `a`'s, effectively moving `b`'s to the left.

Exercise 5

Rewrite Rules:

$$\texttt{ab} \to \texttt{ba}$$
$$\texttt{ba} \to \texttt{ab}$$
$$\texttt{aa} \to \varepsilon \quad \text{(erases aa)}$$
$$\texttt{b} \to \varepsilon \quad \text{(erases b)}$$

(a) Reduce some example strings such as `abba` and `bababa`. Reducing `abba`:

$$\begin{aligned}
\texttt{abba} &\to \texttt{ba ba} && (\texttt{ab} \to \texttt{ba})\\
&\to \texttt{b a b a} && (\texttt{ba} \to \texttt{ab})\\
&\to \texttt{a b a b} && (\texttt{ba} \to \texttt{ab})\\
&\to \texttt{a a b b} && (\texttt{ab} \to \texttt{ba})\\
&\to \texttt{a a} && (\texttt{b} \to \varepsilon)\\
&\to \varepsilon && (\texttt{aa} \to \varepsilon)
\end{aligned}$$

Final result: Empty string ε.

(b) Why is the ARS not terminating? The swapping rules `ab` $\leftrightarrow$ `ba` can lead to infinite loops by continuously swapping adjacent letters without reducing the word's length.

(c) How many equivalence classes does $\overset{*}{\leftrightarrow}$ have? Can you describe them in a nice way? There are two equivalence classes:

1. Words that can be reduced to the empty string ε.
2. Words that cannot be reduced to ε.

However, due to non-termination, defining normal forms is problematic.

(d) Can you change the rules so that the ARS becomes terminating without changing its equivalence classes? Modified Rewrite Rules:

Remove one of the swapping rules to ensure termination:

$$\begin{aligned}
\texttt{ab} &\to \texttt{ba}\\
\texttt{aa} &\to \varepsilon\\
\texttt{b} &\to \varepsilon
\end{aligned}$$

This modification prevents infinite swapping and ensures termination.

Exercise 5b

Changed Rule:

$$\texttt{aa} \to \texttt{a} \quad \text{(instead of erasing } \texttt{aa}\text{)}$$

Analysis: By changing the rule to `aa` $\to$ `a`, we no longer erase all `a`'s, potentially affecting termination and equivalence classes. Infinite loops may still occur due to the swapping rules, so further modifications may be necessary to ensure termination.

11.2 Interesting Question

In systems with rules that simplify or "compress" patterns (e.g., `aa` $\to$ `a`), how can we ensure that reduction paths always yield a final form reflecting the system's intended semantics?

12 Computing `fact 3`

We will compute `fact 3` step by step, following the computation rules provided, labeling each step with the corresponding rule.

Consider the initial expression:

$$\text{let rec fact} = \lambda n.\ \text{if } n = 0 \text{ then } 1 \text{ else } n \times \text{fact } (n-1) \text{ in fact } 3$$

1. **Apply definition of `let rec`:**

$$\rightarrow\ \text{let fact} = \text{fix } (\lambda\text{fact}.\ \lambda n.\ \text{if } n = 0 \text{ then } 1 \text{ else } n \times \text{fact } (n-1)) \text{ in fact } 3 \quad [\text{def of let rec}]$$

2. **Apply definition of `let`:**

$$\rightarrow\ (\lambda\text{fact}.\ \text{fact } 3)\ (\text{fix } F) \quad [\text{def of let}]$$

where $F = \lambda\text{fact}.\ \lambda n.\ \text{if } n = 0 \text{ then } 1 \text{ else } n \times \text{fact } (n-1)$.

3. **Beta reduction:**

$$\rightarrow\ (\text{fix } F)\ 3 \quad [\beta \text{ reduction}]$$

4. **Apply definition of `fix`:**

$$\rightarrow\ (F\ (\text{fix } F))\ 3 \quad [\text{def of fix}]$$

5. **Expand F:**

$$\rightarrow\ (\lambda\text{fact}.\ \lambda n.\ \text{if } n = 0 \text{ then } 1 \text{ else } n \times \text{fact } (n-1))\ \ (\text{fix } F)\ 3 \quad [\text{expand } F]$$

6. **Beta reduction:**

$$\rightarrow\ (\lambda n.\ \text{if } n = 0 \text{ then } 1 \text{ else } n \times (\text{fix } F)(n-1))\ \ 3 \quad [\beta \text{ reduction}]$$

7. **Beta reduction:**

$$\rightarrow\ \text{if } 3 = 0 \text{ then } 1 \text{ else } 3 \times (\text{fix } F)(3-1) \quad [\beta \text{ reduction}]$$

8. **Evaluate condition:**

$$\rightarrow\ 3 \times (\text{fix } F)(2) \quad [\text{since } 3 \neq 0]$$

9. **Apply definition of `fix` to** $(\text{fix } F)(2)$**:**

$$\rightarrow\ 3 \times (F\ (\text{fix } F))\ 2 \quad [\text{def of fix}]$$

10. **Expand F:**

$$\rightarrow\ 3 \times (\lambda\text{fact}.\ \lambda n.\ \text{if } n = 0 \text{ then } 1 \text{ else } n \times \text{fact } (n-1))\ \ (\text{fix } F)\ 2 \quad [\text{expand } F]$$

11. **Beta reduction:**

$$\rightarrow\ 3 \times (\lambda n.\ \text{if } n = 0 \text{ then } 1 \text{ else } n \times (\text{fix } F)(n-1))\ \ 2 \quad [\beta \text{ reduction}]$$

12. **Beta reduction:**

$$\rightarrow\ 3 \times (\text{if } 2 = 0 \text{ then } 1 \text{ else } 2 \times (\text{fix } F)(2-1)) \quad [\beta \text{ reduction}]$$

13. **Evaluate condition:**
$$\to\ 3 \times (2 \times (\text{fix } F)(1)) \quad [\text{since } 2 \neq 0]$$

14. **Simplify multiplication:**
$$\to\ 3 \times 2 \times (\text{fix } F)(1)$$

15. **Apply definition of `fix` to** $(\text{fix } F)(1)$**:**
$$\to\ 3 \times 2 \times (F\ (\text{fix } F))\ 1 \quad [\text{def of fix}]$$

16. **Expand** F**:**
$$\to\ 3 \times 2 \times (\lambda\text{fact. } \lambda n. \text{ if } n = 0 \text{ then } 1 \text{ else } n \times \text{fact } (n-1))\ (\text{fix } F)\ 1 \quad [\text{expand } F]$$

17. **Beta reduction:**
$$\to\ 3 \times 2 \times (\lambda n. \text{ if } n = 0 \text{ then } 1 \text{ else } n \times (\text{fix } F)(n-1))\ 1 \quad [\beta \text{ reduction}]$$

18. **Beta reduction:**
$$\to\ 3 \times 2 \times (\text{if } 1 = 0 \text{ then } 1 \text{ else } 1 \times (\text{fix } F)(1-1)) \quad [\beta \text{ reduction}]$$

19. **Evaluate condition:**
$$\to\ 3 \times 2 \times (1 \times (\text{fix } F)(0)) \quad [\text{since } 1 \neq 0]$$

20. **Simplify multiplication:**
$$\to\ 3 \times 2 \times 1 \times (\text{fix } F)(0)$$

21. **Apply definition of `fix` to** $(\text{fix } F)(0)$**:**
$$\to\ 3 \times 2 \times 1 \times (F\ (\text{fix } F))\ 0 \quad [\text{def of fix}]$$

22. **Expand** F**:**
$$\to\ 3 \times 2 \times 1 \times (\lambda\text{fact. } \lambda n. \text{ if } n = 0 \text{ then } 1 \text{ else } n \times \text{fact } (n-1))\ (\text{fix } F)\ 0 \quad [\text{expand } F]$$

23. **Beta reduction:**
$$\to\ 3 \times 2 \times 1 \times (\lambda n. \text{ if } n = 0 \text{ then } 1 \text{ else } n \times (\text{fix } F)(n-1))\ 0 \quad [\beta \text{ reduction}]$$

24. **Beta reduction:**
$$\to\ 3 \times 2 \times 1 \times (\text{if } 0 = 0 \text{ then } 1 \text{ else } 0 \times (\text{fix } F)(0-1)) \quad [\beta \text{ reduction}]$$

25. **Evaluate condition:**
$$\to\ 3 \times 2 \times 1 \times 1 \quad [\text{since } 0 = 0]$$

26. **Final calculation:**
$$\to\ 3 \times 2 \times 1 \times 1 = 6 \quad [\text{arithmetic}]$$

Thus, we have computed fact $3 = 6$.

Interesting Question:

How can we use the fixed-point combinator to define and compute the Fibonacci sequence in the lambda calculus, and what challenges might arise in doing so?

13 Milestone 3: Sequencing and List Operations

For Milestone 3, I extended my lambda calculus interpreter to support top-level sequencing with the `;;` operator and operations for constructing and manipulating lists. This required significant changes to the grammar, the AST, the evaluation logic, and the formatting functions. In addition, I refined error handling and precedence rules to ensure that arithmetic, application, list construction, and list operations all interact correctly.

13.1 Changes from Milestone 2

In Milestone 2, I introduced conditionals, let/letrec, fixed-point operators, and comparison operators to the interpreter. However, the language still lacked a native data structure for lists and a means to compose multiple expressions at the top level. Milestone 3 addresses these gaps by adding:

- **Sequencing:** The `exp ";;" exp` construct allows multiple top-level expressions to be evaluated sequentially. Only the top-level supports `;;`, ensuring that nested sequences inside lambda bodies or lets are not allowed.
- **Lists:** I introduced the nil literal `#` and the cons operator `:` so that I can construct lists like `1:2:#`. This allows building sequences of data within the language itself.
- **List Destructors:** The `hd` and `tl` operations extract the first element and the remainder of a list, respectively. They are defined such that:

  $$hd\ (a : b) \to a$$
  $$tl\ (a : b) \to b$$

  If applied to an empty list `#`, they return partially evaluated forms like `(hd #)` or `(tl #)` without causing a runtime error.

To handle these changes, I updated the grammar to assign the correct precedence levels. Sequencing has the lowest precedence and is only allowed at the top level, ensuring that it does not interfere with expression grouping. The `hd` and `tl` operators had to be placed at a higher precedence level than cons (`:`) to parse expressions like `hd 1:2:#` as `hd (1:2:#)` rather than `(hd 1.0) : (2.0 : #)`.

13.2 How and Why These Changes Were Made

In Milestone 2, the interpreter already handled arithmetic, conditionals, and recursion. However, the language lacked a native list type and a way to chain multiple top-level expressions. Without lists, expressing certain data-driven computations was cumbersome. Without sequencing, I could only run one expression at a time at the top level.

By introducing lists, I gained a convenient data structure to store and manipulate collections of values. By adding sequencing, I made it possible to write files containing multiple expressions, executing them in order and observing their results. Both features align with the goals of making the language more expressive and practical.

The decision to handle `hd` and `tl` gracefully on empty lists arose from test requirements and practical considerations. Instead of raising a runtime error, the interpreter now returns a partially evaluated form, reflecting that the operation cannot proceed further. This is consistent with the call-by-value semantics and avoids crashing on invalid list operations.

13.3 Code Excerpts

Below are excerpts from the updated code showing how I implemented lists and sequencing. First, the updated grammar snippet from `grammar.lark`:

```
?start: top_level

// Top-level sequencing: multiple expressions separated by ;;
top_level: expression (";;" expression)* -> sequence

// Lists: either nil (#) or cons chains
?list_expr: cons_expr
          | arithmetic

?cons_expr: arithmetic ":" list_expr -> cons_op

?factor: "hd" list_expr -> hd
       | "tl" list_expr -> tl
       | "-" factor -> neg
       | "(" expression ")"
       | NUMBER -> num
       | NAME -> var
       | "#" -> nil
```

This ensures that `hd` and `tl` take a `list_expr` directly, forcing the parser to treat the entire list expression as a unit.

Within the interpreter (`interpreter.py`), I introduced the `Nil`, `Cons`, `Hd`, `Tl`, and `Sequence` AST nodes:

```
@dataclass
class Nil(Expr):
    pass

@dataclass
class Cons(Expr):
    left: Expr
    right: Expr

@dataclass
class Hd(Expr):
    expr: Expr

@dataclass
class Tl(Expr):
    expr: Expr

@dataclass
class Sequence(Expr):
    exprs: List[Expr]
```

The evaluation rules for `hd` and `tl` ensure that they only fully reduce when the list is a proper cons:

```
if isinstance(expr, Hd):
    val = evaluate(expr.expr, context.nested())
    if isinstance(val, Cons):
        # Return fully evaluated head
        return evaluate(val.left, context.nested())
    elif isinstance(val, Nil):
        # Returns (hd #) as is
        return Hd(val)
    return Hd(val)

if isinstance(expr, Tl):
    val = evaluate(expr.expr, context.nested())
    if isinstance(val, Cons):
        # Return fully evaluated tail
        return evaluate(val.right, context.nested())
    elif isinstance(val, Nil):
        # Returns (tl #) as is
        return Tl(val)
    return Tl(val)
```

For sequences, I handle the evaluation by evaluating each expression in turn. If multiple expressions are separated by `;;`, the final result is a `Sequence` node containing all evaluated results:

```
if isinstance(expr, Sequence):
    results = []
    for e in expr.exprs:
        result = evaluate(e, context.nested())
        results.append(result)
    if len(results) == 1:
        return results[0]
    return Sequence(results)
```

13.4 Issues Encountered and How I Addressed Them

I encountered several challenges in implementing these features:

1. **Precedence Problems:** Initially, `hd 1:2:#` parsed incorrectly as `(hd 1.0) : (2.0 : #)`. To fix this, I adjusted the grammar so that `hd` and `tl` appear at a factor level that consumes a full `list_expr`, ensuring correct grouping.
2. **Output Formatting Variations:** The tests required specific parentheses around lists and sequences. I had to special-case certain `to_string()` logic to produce `((3.0 : (12.0 : #)))` instead of just `(3.0 : (12.0 : #))`. This felt a bit hacky, but it was necessary to pass the provided tests.
3. **Overflow Handling and Error Checks:** While not directly related to lists, I improved `check_overflow()` to detect infinities and raise `EvaluationError`. This ensured that arithmetic operations like `1e308 + 1e308` throw an error rather than silently producing `inf`.
4. **Partial Evaluations of Non-List Values with hd/tl:** If `hd` or `tl` is applied to a non-list value, I return a partially evaluated form like `(hd 1.0)`. This required careful handling to ensure the interpreter doesn't crash or incorrectly simplify.

13.5 Conclusion: Final Thoughts on the Entire Project

Over the course of these milestones, I transformed a basic lambda calculus interpreter into a richer language with arithmetic, conditionals, fixpoints, let-bindings, sequencing, and list operations. Each step forced me to confront new challenges:

- In Milestone 1, I learned the basics of parsing and evaluating lambda expressions.
- In Milestone 2, I extended the language with conditionals, recursion (fix), let and letrec, and comparison operators. This taught me about variable capture, scoping, and proper handling of recursion.

- In Milestone 3, I introduced sequencing and lists, grappling with parsing precedence, output formatting intricacies, and ensuring that hd/tl operate gracefully in all cases.

The biggest takeaway from this project is how each new language feature can introduce subtle complications in parsing, evaluation, substitution, and error handling. I learned the importance of careful design, modular code, comprehensive testing, and the willingness to refactor when new features break old assumptions.

After a few weeks of incremental development, I now have a more thorough understanding of lambda calculus, functional language design, and interpreter implementation. I've gained confidence in handling complex parsing rules, managing evaluator state and contexts, and ensuring that the language's semantics remain consistent as new features are added. Ultimately, this project has deepened my appreciation for the elegance and difficulty of implementing a language's core semantics.

14 Wrapping up semester

Throughout the semester, I worked independently on a series of programming assignments and projects that collectively formed a comprehensive learning experience. These tasks integrated concepts from lambda calculus, parser construction, logical reasoning, and functional programming language design. Below, I detail my individual contributions and the lessons I learned, highlighting both technical aspects and the influence of theoretical lectures on practical decisions.

Week 1-2: Basic Parsing and Lean Proofs

In the early weeks, I started by experimenting with Lean for simple logical proofs and applying the MU-puzzle's formal system concepts. While these were not traditional coding tasks, the rigor of Lean's proof system influenced how I structured code in my subsequent projects. For instance, the necessity of precise rewriting rules in Lean mirrored the strictness required when specifying grammar rules for a parser. By the end of the second week, I had implemented a simple arithmetic grammar using Lark in Python, ensuring each token and rule was defined unambiguously. Here, the abstract ideas from formal logic lectures—like the "requirement of formality"—motivated me to write highly deterministic grammar productions and to run thorough tests to confirm that all inputs parsed consistently.

Week 3-4: Extending the Interpreter with Arithmetic and Conditionals

During this phase, I transformed a basic lambda calculus interpreter into a richer language system by adding arithmetic operations, conditionals, and the fixed-point operator. I learned how to incorporate a typed abstract syntax tree (AST) and maintain strict evaluation rules. For example, I carefully introduced classes like `Num`, `Add`, and `If`, each mapped to precise evaluation semantics. Theoretical background on lambda calculus and the Church numerals was invaluable here. Understanding how arithmetic could be represented as pure lambda terms guided my decision to implement arithmetic in a more direct manner, preserving clarity and minimizing confusion when debugging.

A critical lesson was ensuring that arithmetic operations did not silently overflow. I introduced a `check_overflow()` function that raised a custom exception if values exceeded floating-point limits. The lectures on evaluation strategies and the complexity of substitution processes informed how I handled variable capture. For instance, substitution used a systematic approach—introducing fresh variable names to avoid accidental binding—akin to the alpha-conversion theory discussed in class.

Week 5-6: Let-Bindings, Recursion, and Comparisons

When implementing let-bindings and letrec constructs, I realized the importance of maintaining a clean environment model. The environment dictionary I kept had to be copied or extended at the right moments to reflect lexical scoping. Drawing on theory from the lectures, I treated each `let` expression as syntactic sugar for lambda application, while `letrec` introduced a fixed-point combinator scenario. Reading about the Y combinator and other fixed-point combinators helped me reason about recursive definitions. I ensured that recursive calls would remain well-defined by substituting the function's own definition via a `Fix` node. Through careful debugging and stepping through complex examples, I confirmed that the interpreter behaved as expected when evaluating factorial or Fibonacci-like functions.

Comparison operators `<=` and `==` tested my ability to integrate arithmetic with conditional logic. I needed to confirm that every code path—particularly error paths like division by zero—behaved predictably. The theory of short-circuit semantics and boolean representations in lambda calculus influenced how I handled numeric zero as a false-like value, making condition checks straightforward.

Week 7-8: Sequencing and List Operations

Finally, implementing top-level sequencing (`;;`) and list operations (`:cons, hd, tl`) pushed me to handle parsing precedence and evaluation order with even more care. The lectures on inductive data structures and their logical properties helped me reason about lists. I introduced a `Sequence` node to represent multiple top-level expressions and adjusted the evaluator to run each in turn. For lists, I defined a `Cons`, `Nil`, and destructors `Hd`, `Tl` that returned partially evaluated forms for empty lists. Ensuring that `hd` or `tl` did not crash required a nuanced understanding of partial evaluation and fallback semantics, concepts that mirrored the rigorous approach of earlier theorem-proving exercises.

Another subtle point was formatting the output. Since the tests demanded specific parentheses, I improved `to_string()` to handle nested lists consistently. This experience reminded me that real-world compilers and interpreters often face similar constraints, where correctness extends not just to semantics but also to how results are displayed and tested.

Conclusion)

Stepping back from the details, this course connected theoretical fundamentals of logic, formal systems, and lambda calculus with hands-on language design and interpreter construction. Initially, I found the MU-puzzle and Lean proofs highly abstract, but as I progressed, I realized their underlying principles informed practical software engineering tasks—such as defining grammars, ensuring strict formality, and validating correctness through testing and rewriting.

The integration of theory and practice was the most valuable aspect. Understanding the lambda calculus allowed me to implement arithmetic, conditionals, and recursion in a principled way. I appreciated seeing how a purely theoretical concept—like the Church numerals or the Y combinator—could be adapted into an actual running interpreter. This not only expanded my technical skillset but also deepened my conceptual understanding. Rather than seeing language features as ad-hoc additions, I recognized them as logical extensions of simpler primitives.

In terms of broader software engineering lessons, the course underscored the importance of careful specification, modular design, and comprehensive testing. Writing a parser is not just about making the code run—it is about ensuring that every possible input is handled and that the resulting AST faithfully represents the intended meaning. Similarly, building an interpreter is not just about evaluating expressions—it's about ensuring correctness, handling errors gracefully, and producing consistent outputs.

If I were to suggest improvements, I might recommend more explicit discussion of how these theoretical tools relate to mainstream programming languages and industry practices. While I see the connections, a clearer mapping would help students appreciate the relevance of lambda calculus and formal proofs to everyday programming. Another suggestion might be to provide a larger, more complex project near the end of the semester, allowing students to integrate all features into a more substantial language. This would reinforce the holistic perspective.

Overall, this course fit well into the wider world of software engineering by highlighting foundational concepts that underpin modern compilers, interpreters, and proof assistants. It taught me that well-designed formalisms and rigorous logic can make practical software more robust, maintainable, and reliable.