

CPSC-354 Report

Zack Klopukh
Chapman University

September 30, 2024

Abstract

Documentation of my notes, questions, and homework throughout my CPSC-354 class can be found here.

Contents

1	Introduction	1
2	Week by Week	1
2.1	Week 1	1
2.2	Week 2	2
2.3	Week 3	4
2.4	Week 4	5
2.5	Week 5	7
2.6	8
3	Lessons from the Assignments	8
4	Conclusion	8

1 Introduction

2 Week by Week

2.1 Week 1

Notes

Using proofs is important for reasoning on how computers can operate.

Homework

5) Goal: $a + (b + 0) + (c + 0) = a + b + c$

```
rw [add_zero b, add_zero c]
```

$a + b + c = a + b + c$

```
rfl
```

Here we use `add_zero` to rewrite and remove the extra zeros so that both sides are equal by reflexivity.

6) Goal: $a + (b + 0) + (c + 0) = a + b + c$

```
rw [add_zero b, add_zero c]
```

$a + b + c = a + b + c$

rfl A nearly identical solution as number 5.

```
7) Goal: succ n = succ (n + 0)  rw [add_zero]
rfl
Here we used add_zero to rewrite n+0 to n which was all it took to show equality through reflexivity.
```

```
8) Goal: 2 + 2 = 4
  rw [four_eq_succ_three]
2 + 2 = succ 3
  rw [three_eq_succ_two]
2 + 2 = succ (succ 2)
  rw [two_eq_succ_one]
succ 1 + succ 1 = succ (succ (succ 1))
  rw [one_eq_succ_zero]
succ (succ 0) + succ (succ 0) = succ (succ (succ (succ 0)))
  rw [add_succ]
succ (succ (succ 0) + succ 0) = succ (succ (succ (succ 0)))
  rw [add_succ]
succ (succ (succ (succ 0) + 0)) = succ (succ (succ (succ 0)))
  rw [add_zero]
succ (succ (succ (succ 0))) = succ (succ (succ (succ 0)))
rfl
```

This is the longest proof in the homework and involves multiple rewritings to show. The process I took was turning both 2's and the 4 into their forms as successors of 0. From that point I added successors to the left side and by doing so I rewrite the two separate 2's into one large successor. Finally by using add zero, I removed one of the two left terms completely making it a single term on the left and right, both representative of 4.

Comments and Questions

This introduction world taught me how precise proofs need to be. I am beginning to understand how important it is to a machine that proofs are completely concrete, to the point where $2+2=4$ becomes a tricky problem. As I go on to take these rewrites for granted, I will with the understanding that under the hood the computer does not go off intuition, but precise and exact steps.

Question of the week: How can systems of algorithmic reasoning as shown in the tutorial world, based in discrete mathematics, be encoded in modern programming languages?

2.2 Week 2

Homework

Question 1: Prove the theorem 'zero.add': For all natural numbers n , we have $0 + n = n$.

Proof:

```
theorem zero_add : n : ℕ, 0 + n = n :=

  intro n,
  induction n with d hd,
  rw add_zero
  rw add_succ
  rw hd
  rfl
```

This uses induction to prove the zero_add property. It is a simple way to introduce inductive reasoning.

Question 2: Prove the theorem: For all natural numbers a, b , we have $\text{succ}(a) + b = \text{succ}(a + b)$.

Proof:

```

induction b with a b
rw [add_zero]
rw [add_zero]
rfl
rw [add_succ]
rw [b]
rw [add_succ]
rfl

```

This proof uses induction on b to show that adding $\text{succ}(a)$ to b results in $\text{succ}(a + b)$. Inductive reasoning is key to establishing this property.

Question 3: Prove the theorem: On the set of natural numbers, addition is commutative. In other words, if a and b are arbitrary natural numbers, then $a + b = b + a$.

Proof:

```

induction a with b hd
rw [add_zero]
rw [zero_add]
rfl
rw [add_succ]
rw [succ_add]
rw [succ_eq_add_one]
rw [succ_eq_add_one]
rw [hd]
rfl

```

This proof establishes the commutativity of addition on natural numbers by using induction on a . Each step involves rewrites that simplify the equation until both sides match.

Question 4: Prove the theorem: On the set of natural numbers, addition is associative. In other words, if a , b , and c are arbitrary natural numbers, we have $(a + b) + c = a + (b + c)$.

Proof:

```

induction a with b hd
rw [zero_add]
rw [add_comm]
rw [zero_add]
rfl
rw [succ_add]
rw [succ_add]
rw [succ_add]
rw [succ_eq_add_one]
rw [succ_eq_add_one]
rw [hd]
rfl

```

This proof uses induction on a to establish the associativity of addition on the natural numbers. Each step simplifies the expression to demonstrate that the addition operation is associative.

Question 5: Prove the theorem: If a , b , and c are arbitrary natural numbers, we have $(a + b) + c = (a + c) + b$.

Proof:

```

induction a with b hd
rw [zero_add]

```

```

rw [zero_add]
rw [add_comm]
rfl
rw [add_assoc]
nth_rewrite 1 [add_assoc]
nth_rewrite 2 [add_comm]
rfl

```

This proof uses induction on a to show that the expression $(a + b) + c$ can be rearranged to $(a + c) + b$. Various rewrite steps and the commutativity and associativity of addition are used to achieve the result.

Comments and Questions

In this homework, I used lean to complete many proofs I've done in Discrete Mathematics including commutativity and associativity. These are proofs that I did previously use induction to solve, but representing that through lean to a computer makes the challenge extra formal, as there can be no jumps in logic. It's another week where the homework makes me thoughtful on how computers function at a lower level.

What challenges or additional considerations might arise when using a more complicated set like the set of all real numbers compared to the set of natural numbers?

2.3 Week 3

Homework

This week, I used a Large Language Model (LLM) to delve deeply into a research question that piqued my interest. Beyond obtaining basic answers, I further explored the advancements this topic has facilitated in various fields, as these developments are of particular interest to me.

What role does syntax play in the usability of programming languages?

For a comprehensive report detailing my research and findings, please refer to the following link:

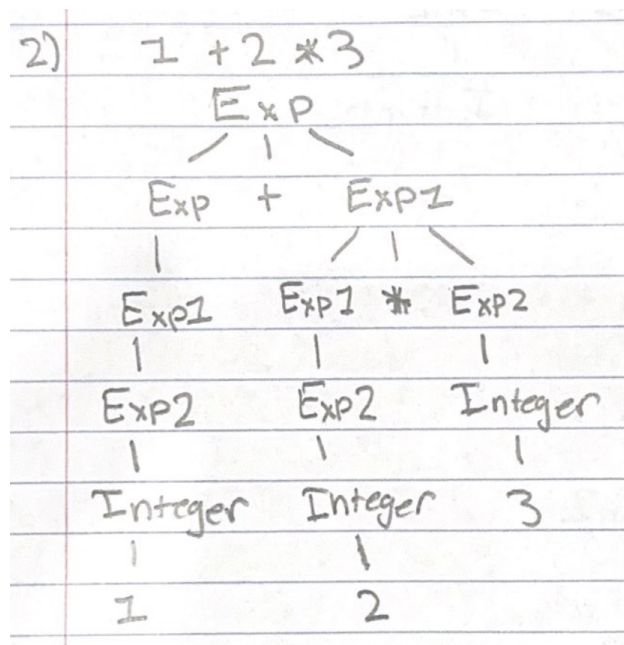
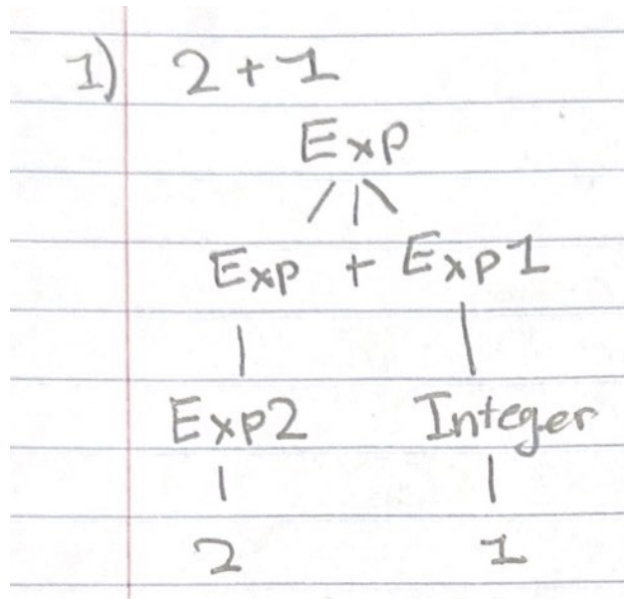
<https://github.com/zackklopukh/LLMReport/blob/main/README.md>

Discord Posting

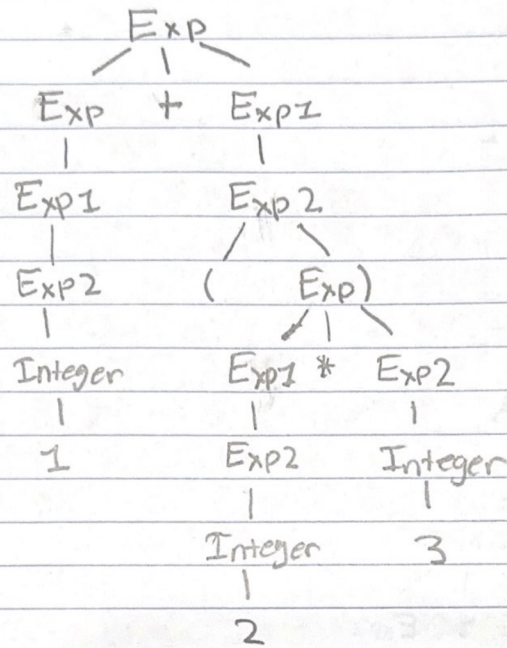
In my literature review with GPT-4o, I examined the profound influence of user-friendly interfaces on programming languages. Initially, programming was confined to those with deep technical knowledge due to complex syntax and low-level languages. However, the evolution of programming syntax to more readable and high-level languages significantly broadened access to programming, inviting individuals from diverse fields to contribute and innovate. The shift from assembly languages to high-level languages such as FORTRAN, COBOL, and later Python and JavaScript revolutionized the field. This transition facilitated increased creativity and productivity, as developers could focus more on solving problems rather than managing low-level details. The development of high-level languages enabled rapid software development, democratizing programming and spurring advancements across various domains. In the report with GPT-4o, I explored how modern hybrid systems combine interpreted and compiled code to enhance performance and flexibility. Techniques like Just-In-Time (JIT) compilation and the integration of high-level languages with systems like WebAssembly illustrate the ongoing evolution of programming. These advancements reflect the growing synergy between user-friendly interfaces and robust performance, continuing to shape the future of programming. <https://github.com/zackklopukh/LLMReport/blob/main/README.md>

2.4 Week 4

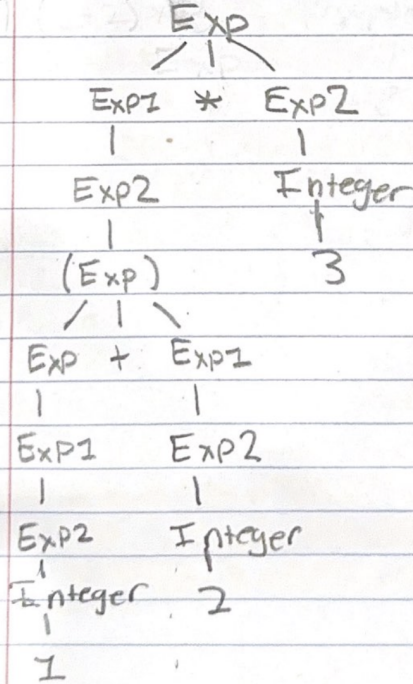
Homework

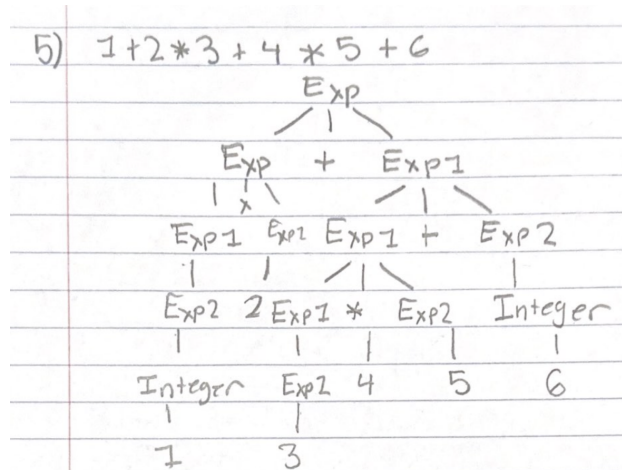


3) $1 + (2 + 3)$



4) $(1 + 2) * 3$





Comments and Questions

How do different parsing strategies compare when evaluating factors such as speed or memory usage?

Comparing a few different parsing strategies. Top-down recursive parsing may be slower if backtracking is required but it is very data efficient. Bottom-up can be faster especially for deterministic grammars although in some cases could use more memory in the case of more complex grammar. Packrat parsing is also very fast as it doesn't require any backtracking, but storing the intermediate results will require extra memory.

2.5 Week 5

Homework

1. Exhibit evidence that you're planning a party.
exact todo.list
2. Fill a box, label correctly.
exact and.intro p s
3. Three x and.intro.
exact and.intro (and.intro a i) (and.intro o u)
4. Exhibit evidence that Pippin is coming to the party.
exact and.left vm
5. Both P and Q entails just Q as well!
exact h.right
6. and.intro, and.left, and.right.
exact and.intro (and.left h1) (and.right h2)
7. Navigate the tree.
exact h.left.left.right
8. Take apart and build evidence.
exact (h.left.right, h.right.right.left.left, h.left.left.left, h.left.left.right)

Comments and Questions

How do the conjunction and disjunction propositions seen in LLG influence the design of type systems in developed programming languages?

GPT: The conjunction (AND) and disjunction (OR) propositions from LLG significantly influence the design of type systems in programming languages by enabling type composition and enhancing type safety. Conjunctions correspond to product types (like tuples), allowing for complex data structures, while disjunctions relate to sum types (like variants), enabling flexible return types. These logical constructs facilitate pattern matching and support formal verification, ensuring that programs adhere to their specifications and reducing runtime errors. Overall, they provide a robust foundation for creating expressive and reliable type systems in modern programming languages.

2.6 ...

...

3 Lessons from the Assignments

Write this section during the semester. This is approximately a quarter of a page per week and the material should come from the work you do anyway. Just keep your eyes open for interesting lessons.

Make sure that you use L^AT_EX to structure your writing (eg by using subsections).

4 Conclusion

Soon to be.

References

[BLA] Author, [Title](#), Publisher, Year.