CPSC-354 Report

Maxwell Rovenger
Chapman University

December 16, 2024

Abstract

Contents

1 Introduction

In this report, I aim to establish a comprehensive understanding of the fundamentals of programming languages through the application of Lean logic, recursion, LLMs, and parsers. This is achieved by developing a solid foundation in discrete mathematics, utilizing it to create a grammar file with Lark, and exploring how these principles act as a framework for the structure and functionality of programming languages. The report

delves into the application of discrete mathematical concepts to define syntax and semantics, demonstrates the implementation of grammar using Lark for parsing, and highlights the integration of recursion and logic in constructing efficient solutions. By bridging theoretical knowledge with practical application, this report seeks to provide a holistic perspective on the fundamentals of programming languages.

2 Week by Week

2.1 Week 1

Notes

Lectures introduced Lean, a programming language to help prove discrete mathematics proofs. Through using the Natural Numbers Game, we saw how Lean operates and how it works it can, with functions acting as steps, prove theorems similar to how we did in discrete math with pen and paper and induction.

Homework

Solved problems in Natural Numbers Game using Lean to help recap teachings from Discrete Mathematics. Specifically dealing with successors and predecessors and how they can be used to change certain sides to equal the other.

Level 5/8:

$$\begin{aligned} a + (b + 0) + (c + 0) &= a + b + c && \text{by add\_zero} \\ a + b + (c + 0) &= a + b + c && \text{by add\_zero} \\ a + b + c &= a + b + c && \text{by reflexivity (rfl)} \end{aligned}$$

Level 6/8:

$$\begin{aligned} a + (b + 0) + (c + 0) &= a + b + c && \text{by add\_zero on c} \\ a + (b + 0) + c &= a + b + c && \text{by add\_zero on b} \\ a + b + c &= a + b + c && \text{by reflexivity (rfl)} \end{aligned}$$

Level 7/8:

$$\begin{aligned} \operatorname{succ} n &= n + 1 && \text{by one\_eq\_succ\_zero} \\ &= n + \operatorname{succ} 0 && \text{by add\_succ} \\ &= \operatorname{succ} (n + 0) && \text{by add\_zero} \\ &= \operatorname{succ} n && \text{by reflexivity (rfl)} \end{aligned}$$

Level 8/8:

$$\begin{aligned} 2 + 2 &= 4 && \text{by four\_eq\_succ\_three} \\ 2 + 2 &= \operatorname{succ} 3 && \text{by three\_eq\_succ\_two} \\ 2 + 2 &= \operatorname{succ} (\operatorname{succ} 2) && \text{by two\_eq\_succ\_one} \\ \operatorname{succ} 1 + \operatorname{succ} 1 &= \operatorname{succ} (\operatorname{succ} (\operatorname{succ} 1)) && \text{by add\_succ} \\ \operatorname{succ} (\operatorname{succ} 1 + 1) &= \operatorname{succ} (\operatorname{succ} (\operatorname{succ} 1)) && \text{by succ\_eq\_add\_one} \\ \operatorname{succ} 1 + 1 + 1 &= \operatorname{succ} (\operatorname{succ} (\operatorname{succ} 1)) && \text{by succ\_eq\_add\_one} \\ 1 + 1 + 1 + 1 &= \operatorname{succ} (\operatorname{succ} (1 + 1)) && \text{by succ\_eq\_add\_one} \\ 1 + 1 + 1 + 1 &= \operatorname{succ} (1 + 1) + 1 && \text{by succ\_eq\_add\_one} \\ 1 + 1 + 1 + 1 &= 1 + 1 + 1 + 1 && \text{by reflexivity (rfl)} \end{aligned}$$

For level 5/8 specifically, we can see that the Lean proof, `rw[add_zero]`, corresponds to Proof Algorithm 1: Addition, in that any variable added to zero will ultimately equal just the variable. For example: $a + 0 = a$.

From this homework, I learned how to use the Lean proof and saw how each function operated exactly like algorithms and proofs I had used in Discrete Mathematics.

Comments and Questions

Although this week generally served as just an introduction to the curriculum and a recap of discrete mathematics, I am curious as to whether or not there was a better example to be shown of how exactly a computer uses discrete mathematics. At this point, I am knowledgeable of how discrete mathematics operates and how computers can use operators to conduct mathematical operations, but I have yet to see a direct example of a computer "thinking" through a math calculation.

If the way computers have been taught mathematics is based purely on successors and predecessors, does that make other operations like multiplication, division, and exponents far more taxing on a CPU since they have to calculate an incredible amount of successors, or do the operators used to cause multiplication and exponentiation ignore that by just creating duplicates and adding them together?

2.2 Week 2

Notes

Recursion, in coding, allows for more simplistic code that is easier to read, scale, and apply.

Regarding the allowing of typos in coding, code should be non-ambigious because coding should be universal and should work the same within the same virtual machine regardless of how you access it. We see examples of this in GitHub's copilot since it only makes suggestions that are non-ambigious and can work on any machine, as long as they have the correct virtual machine.

Lean contains tactics, Ex: rw, and theorems, Ex: one_eq_succ_zero. Tactics are commands while theorems are logical propositions.

Homework

Level 1/5:

$$\begin{aligned} 0 + n &= n && \text{by induction on } n \\ 0 + 0 &= 0 && \text{by add\_zero} \\ 0 &= 0 && \text{by reflexivity (rfl)} \\ 0 + \operatorname{succ} d &= \operatorname{succ} d && \text{by add\_succ} \\ \operatorname{succ}(0 + d) &= \operatorname{succ} d && \text{by induction hypothesis (hd)} \\ \operatorname{succ} d &= \operatorname{succ} d && \text{by reflexivity (rfl)} \end{aligned}$$

Level 2/5:

$$\begin{aligned} \operatorname{succ} a + b &= \operatorname{succ}(a + b) && \text{by induction on } b \\ \operatorname{succ} a + 0 &= \operatorname{succ}(a + 0) && \text{by add\_zero} \\ \operatorname{succ} a &= \operatorname{succ} a && \text{by reflexivity (rfl)} \\ \operatorname{succ} a + \operatorname{succ} b &= \operatorname{succ}(a + \operatorname{succ} b) && \text{by add\_succ} \\ \operatorname{succ}(\operatorname{succ} a + b) &= \operatorname{succ}(a + \operatorname{succ} b) && \text{by induction hypothesis (n\_ih)} \\ \operatorname{succ}(\operatorname{succ}(a + b)) &= \operatorname{succ}(\operatorname{succ}(a + b)) && \text{by reflexivity (rfl)} \end{aligned}$$

Level 3/5:

$$\begin{aligned}
a + b &= b + a && \text{by induction on } b \\
a + 0 &= 0 + a && \text{by add_zero} \\
a &= 0 + a && \text{by zero_add} \\
a &= a && \text{by reflexivity (rfl)} \\
a + \operatorname{succ} b &= \operatorname{succ} b + a && \text{by add_succ} \\
\operatorname{succ}(a + b) &= \operatorname{succ} b + a && \text{by induction hypothesis (hb)} \\
\operatorname{succ}(b + a) &= \operatorname{succ} b + a && \text{by succ_add} \\
\operatorname{succ}(b + a) &= \operatorname{succ}(b + a) && \text{by reflexivity (rfl)}
\end{aligned}$$

Level 4/5:

$$\begin{aligned}
a + b + c &= a + (b + c) && \text{by induction on } c \\
a + b + 0 &= a + (b + 0) && \text{by add_zero} \\
a + b &= a + b && \text{by reflexivity (rfl)} \\
a + b + \operatorname{succ} c &= a + (b + \operatorname{succ} c) && \text{by add_succ} \\
\operatorname{succ}(a + b + c) &= a + (b + \operatorname{succ} c) && \text{by add_succ} \\
\operatorname{succ}(a + b + c) &= a + \operatorname{succ}(b + c) && \text{by add_succ} \\
\operatorname{succ}(a + b + c) &= \operatorname{succ}(a + (b + c)) && \text{by induction hypothesis (hc)} \\
\operatorname{succ}(a + (b + c)) &= \operatorname{succ}(a + (b + c)) && \text{by reflexivity (rfl)}
\end{aligned}$$

Level 5/5:

$$\begin{aligned}
a + b + c &= a + c + b && \text{by induction on } c \\
a + b + 0 &= a + 0 + b && \text{by add_zero and add_zero} \\
a + b &= a + b && \text{by reflexivity (rfl)} \\
a + b + \operatorname{succ} c &= a + \operatorname{succ} c + b && \text{by add_succ} \\
\operatorname{succ}(a + b + c) &= a + \operatorname{succ} c + b && \text{by add_succ} \\
\operatorname{succ}(a + b + c) &= \operatorname{succ}(a + c) + b && \text{by succ_add} \\
\operatorname{succ}(a + b + c) &= \operatorname{succ}(a + c + b) && \text{by induction hypothesis (hc)} \\
\operatorname{succ}(a + c + b) &= \operatorname{succ}(a + c + b) && \text{by reflexivity (rfl)}
\end{aligned}$$

Comments and Questions

Personally, I felt that when it comes to AI autocorrecting code, instead of being lenient towards mistyped code, AI could just simply correct the typos. This would allow for a smoother coding process wihle also creating non-ambigious code.

After playing around with the Tower of Hanoi, it is apparent that recursion is invaluable in programming. We often see it as a way to neatly write code but it can also be massive time saver as apparent with the Tower of Hanoi where instead of having to, through trial and error finding out the solution, we can just use a recursive equation and plug in our values. My question then is, in what other fields can we apply recursion, as it is becoming more apparent to me that recursion can be a form of thinking rather than a field specific method?

Lean Proof

$$\begin{aligned} a+b+c &= a+(b+c) \\ a+b+0 &= a+(b+0) == defof+ \\ a+b &= a+(b+0) == defof+ \\ a+b &= a+b \end{aligned}$$

2.3 Week 3

Notes

It is impossible for a computer to prove all mathematical proofs since there are statements in arithmetic that are unclear if they are true or not and thus cannot be proved. This is due to there being proofs in mathematics that are built based upon assumption and although there are no problems found so far, they cannot be proved through ordinary means like what the computer would opt to do.

Homework

LLM Literature Review README

Discord Posting

For my literature review, I used ChatGPT4 to ask questions about recursion within computer systems. More specifically, if it decreases memory usage or provides any processing speed based benefits rather than just readability or integration based benefits. Through my conversation, I learned that due to creating more and more stack frames for each iteration, recursion can actually increase memory usage. However, there is an alternative, that being tail recursion. It was developed in 1970s however it either does not have support or only has limited support for popular languages like Python and JavaScript. Thus, it was concluded that recursive functions are actually less memory efficient than iterative solutions. This surprised me as through recursion, humans are able to abstract a lot of our thinking and are able to quickly find out what the next iterative output is. This contrasts computers which require more processing power to perform recursive methods. This creates an inverse relationship where the better the human understanding, the worse the computer understanding.

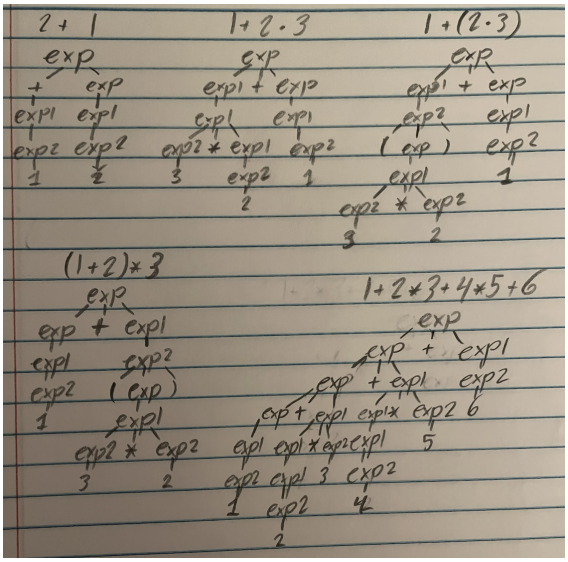
2.4 Week 4

Notes

An abstract syntax tree is the result of parsing and helps display the order and relationship between operators in a mathematical problem. the abstract syntax tree is the most intermediate, as it is more drawn out than both concrete syntax and a concrete syntax tree.

We can define well formed expressions through formal grammar. Since formal grammar helps to create definitions for what becomes a well-formed expression.

Homework



Questions and Comments

If my understanding is correct, all formal languages are allowed to have their own alphabets and grammar that must be defined. However, we have found ourselves in a position where the majority of, if not all, programming langauges will use the same characters or symbols for operators and values. Thus, my question is if there is a protocol enforced that mandates the formal language used in a programming languages or has this all occurred just for ease of use?

2.5 Week 5

Notes

In this week's lecture, we learned about the logic behind AND and how computer's process it. This shows that a proof in logic can be represented as a program in a programming language and vice versa.

Similarly, in the natural numbers game, we saw the application of these theorems which allowed us to prove theorems in Lean. So if a refresher is needed, going through the "A Lean Intro To Logic" is a good idea.

Homework

Level 1/8:

todo_list : P

exact todo_list

Level 2/8:

Goal : $P \wedge S$

exact and_intro p s

Level 3/8:

Goal : $(A \wedge I) \wedge O \wedge U$

exact and_intro (and_intro a i) (and_intro o u)

Level 4/8:

Assumptions :
−vm : $P \wedge S$
have p := vm.left
Assumptions :
−vm : $P \wedge S$
−p : P
exact p

Level 5/8:

Assumptions :
−h : $P \wedge Q$
have q := and_right h
Assumptions :
−h : $P \wedge Q$
−q : Q
exact q

Level 6/8:

Assumptions :
−h1 : $A \wedge I$
−h2 : $O \wedge U$
have A := and_left h1
Assumptions :
−h1 : $A \wedge I$
−h2 : $O \wedge U$
−A : A
have U := and_right h2
Assumptions :
−h1 : $A \wedge I$
−h2 : $O \wedge U$
−A : A
−U : U
exact and_intro A U

Level 7/8:

Assumptions :
$-\text{h} : (L \wedge ((L \wedge C) \wedge L) \wedge L \wedge L \wedge L) \wedge (L \wedge L) \wedge L$
have h1 := and_left h
Assumptions :
$-\text{h} : (L \wedge ((L \wedge C) \wedge L) \wedge L \wedge L \wedge L) \wedge (L \wedge L) \wedge L$
$-\text{h1} : L \wedge ((L \wedge C) \wedge L) \wedge L \wedge L \wedge L$
have h2 := and_right h1
Assumptions :
$-\text{h} : (L \wedge ((L \wedge C) \wedge L) \wedge L \wedge L \wedge L) \wedge (L \wedge L) \wedge L$
$-\text{h1} : L \wedge ((L \wedge C) \wedge L) \wedge L \wedge L \wedge L$
$-\text{h2} : ((L \wedge C) \wedge L) \wedge L \wedge L \wedge L$
have h3 := and_left h2
Assumptions :
$-\text{h} : (L \wedge ((L \wedge C) \wedge L) \wedge L \wedge L \wedge L) \wedge (L \wedge L) \wedge L$
$-\text{h1} : L \wedge ((L \wedge C) \wedge L) \wedge L \wedge L \wedge L$
$-\text{h2} : ((L \wedge C) \wedge L) \wedge L \wedge L \wedge L$
$-\text{h3} : (L \wedge C) \wedge L$
have h4 := and_left h3
Assumptions :
$-\text{h} : (L \wedge ((L \wedge C) \wedge L) \wedge L \wedge L \wedge L) \wedge (L \wedge L) \wedge L$
$-\text{h1} : L \wedge ((L \wedge C) \wedge L) \wedge L \wedge L \wedge L$
$-\text{h2} : ((L \wedge C) \wedge L) \wedge L \wedge L \wedge L$
$-\text{h3} : (L \wedge C) \wedge L$
$-\text{h4} : L \wedge C$
have h5 := and_right h4
exact h5

Level 8/8:

$$\begin{aligned}
&\text{Assumptions :}\\
-\text{h} &: ((P \wedge S) \wedge A) \wedge \neg I \wedge (C \wedge \neg O) \wedge \neg U\\
&\text{have h1 := and_left h}\\
&\text{Assumptions :}\\
-\text{h} &: ((P \wedge S) \wedge A) \wedge \neg I \wedge (C \wedge \neg O) \wedge \neg U\\
-\text{h1} &: (P \wedge S) \wedge A\\
&\text{have h2 := and_left h1}\\
&\text{Assumptions :}\\
-\text{h} &: ((P \wedge S) \wedge A) \wedge \neg I \wedge (C \wedge \neg O) \wedge \neg U\\
-\text{h1} &: (P \wedge S) \wedge A\\
-\text{h2} &: P \wedge S\\
&\text{have h3 := and_right h1}\\
&\text{Assumptions :}\\
-\text{h} &: ((P \wedge S) \wedge A) \wedge \neg I \wedge (C \wedge \neg O) \wedge \neg U\\
-\text{h1} &: (P \wedge S) \wedge A\\
-\text{h2} &: P \wedge S\\
-\text{h3} &: A\\
&\text{exact and_intro h3 (and_intro h2 h1)}
\end{aligned}$$

level 8/8 in Mathematical Logic

(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)	$\neg I \wedge (C \wedge \neg O) \wedge \neg U$	and_right (1)
(6)	$(C \wedge \neg O) \wedge \neg U$	and_right (5)
(7)	$C \wedge \neg O$	and_left (6)
(8)	C	and_left (7)
(9)	P	and_left (3)
(10)	S	and_right (3)

Comments and Questions

Following our exposure to proving the new theorem regarding "AND" in Lean, I took note of the recursive nature of this course. Specifically, how all topics in this course build up on each other. This lead me to asking the question: How does recursion play into isomorphism and is the scalability of recursive functions a necessity for theorems and logic within programming?

2.6 Week 6

Notes

Implication is actually just a function that has to be proved through theorems and proofs rather than just visual confirmation.

Precedence: An integral part of programming languages, decides the order of operations in a mathematical problem. This is important as it can change the outcome of a problem if not done correctly.

- Appraisal: Processed left-to-right
- λ Abstraction: Processed right-to-left

Homework

Level 1/9:

`Assumptions:`
$-bakery\_service : P \to C$
`exact bakery_service p`

Level 2/9:

$haveh_1 : C \to C := func : C => c$
`Assumptions:`
$-h_1 : C \to C$
`exact h_1`

Level 3/9:

`Objects:`
`- I S : Prop`
`Goal:`
$-I \wedge S \to S \wedge I$
$exact\lambda h : I \wedge S \to and_intro(and\_righth)h.left$

Level 4/9:

`Assumptions:`
$-h1 : C \to A$
$-h2 : A \to S$
$exacth1 \gg h2$

Level 5/9:

`Assumptions:`
`-p : P`
$-h1 : P \to Q$
$-h2 : Q \to R$
$-h3 : Q \to T$
$-h4 : S \to T$
$-h5 : T \to U$
$exact(h1 \gg h3 \gg h5)p$

Level 6/9:

`Assumptions:`
$-h : C \wedge D \to S$
$exactfunc => fund => h\langle c, d\rangle$

Level 7/9:

`Assumptions:`
$-h : C \to D \to S$
$exactfun\langle c, d\rangle => hcd$

Level 8/9:

`Assumptions:`
$-(S \to C) \wedge (S \to D)$
$exactfuns => \langle h.1s, h.2s\rangle$

Level 9/9:

```
Objects:
  - R S : Prop
Goal:
```
$-R \to (S \to R) \wedge (\neg S \to R)$
$exactfunr => \langle fun => r, fun => r\rangle$

Comments and Questions

Although closing laptops allows for better lecture retention, it heavily damages the ability to take notes, decreasing the value of the report's "notes" subsection.

In class, we commonly hear the term "context-free grammar", is it not the case where all grammar within computers should be context free considering it is the foundation for mathematics which always operates the same? What are some examples of grammar that is context specific?

2.7 Week 7

Notes

λ-calc: Useful helpers

1. Projects:
   - (1a) Take 2 args & return 1st lambax.lambday.x (lambdax.lambday.y)
   - (2a)
2. Erase: Take 3 args & return 1st & 3rd: lambdax.lambday.lambdaz.xz
3. Duplicate: take 1 arg & duplicate lambdax.xx

How to encode logic in λ-calc:

1. Specifications:
   - if-then-else true M N = M
   - if-then-else false M N = N
   - true := λx.λy.x
   - false := λx.λy.y

2. Not:
   - not true = false
   - not false = true

- not M = if-then-else M false true
- not := λx.x false true

Homework

1) Reducing the lambda term:

We are given the term:

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

Step 1: Apply the first function

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

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

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

Step 2: Apply the second function

Now apply $\lambda n.(\lambda f.\lambda x.f(f\, x))\, n$ to $(\lambda f.\lambda x.f(f(f\, x)))$:

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

Step 3: Apply the inner function

Finally, apply $(\lambda f.\lambda x.f(f\, x))$ to $(\lambda f.\lambda x.f(f(f\, x)))$:

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

This simplifies to:

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

Final result:

The fully reduced form is:

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

2) Explanation of the function $\lambda m.\lambda n.m\, n$:

The lambda term $\lambda m.\lambda n.m\, n$ represents a higher-order function that takes two arguments. The first argument, m, is a function, and the second argument, n, is a value. The function applies m to n.

In the context of natural numbers, this function can be interpreted as a basic operation of applying a function m to an argument n, commonly seen in **Church encoding**. Specifically, it applies m to n, where m could represent a function that operates on natural numbers encoded in the lambda calculus style, and n is the number to which m is applied.

Thus, $\lambda m.\lambda n.m\,n$ essentially represents a function that applies m to n, and in the case of Church numerals, this can represent various operations on numbers, such as adding or multiplying, depending on the nature of m.

Comments and Questions

Lambda calculus seems to be extremely low level and be the defining foundation for a lot of programming calculations. With this said, how is it not its own programming language that is in use today for the sake of the fastest possible runtime, sort of similar to how assembly is the bare bones equivalent of programming.

2.8 Week 8/9

Notes

Extensions of lambda calculus
- Basic logic: boolean logic, if-construct
- Basic arithmetic (numerals, add, exp,...)
Want:
- let l = local (names)s
- recursion

Introduce let: Example from VSCode:
- let plus = $\lambda m.\lambda n.xf.\lambda x.mf(ufx)$
in let two = $\lambda f.\lambda x.f(fx)$
in let one = $\lambda f.\lambda x.fx$
in plus one two

Concrete syntax: let x = e, in ez
Goal: define this λ-calc (Church encoding) but can also add this as syntectic sugar.
add to CFG: $c \to c \to$ let id = e in e
abstract syntax:

Y-combinator

$Y := fix := \lambda f.(\lambda x.f(xx))(\lambda x.f(xx))$

Claim: (fix F) is an fp of F, ie, F $\to$ F (fix F).
Proof: fixF $\to (\lambda x.F(xx))(\lambda x.F(xx))$

Homework

Exercise 2:
- a b c d $\to$ (((a b) c) d)
- This is because the grammar identifies that there are no parentheses and thus seeks to establish the order of operations.
- (a) $\to$ a
- This is because the grammar identifies that there are parentheses and then seeks to simplify the equation.

Exercise 3:
- Capture-avoiding substitution works by ensuring that when substituting a variable x in an expression, no free variables get mistakenly bound by a new scope. If a substitution would result in a name conflict (i.e., x being replaced in a context where it is already bound), an α-conversion (renaming of bound variables) is performed to avoid variable capture.

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

print("\nCapture-Avoiding Substitution: Test passed!\n")

print("\nevaluate(): All tests passed!\n")
```

- Implemented by:
1. Base case: If the term is a variable and matches the one being substituted, replace it with the expression.
2. Lambda abstraction: When encountering a lambda abstraction $\lambda y.e$, check if the bound variable y conflicts with x . If so, rename y to a fresh variable before substituting.
3. Recursion: Perform the substitution recursively in the body of the lambda or function application, applying α-conversion where necessary to avoid capture.

Exercise 4: - When dealing with capture avoiding substitution, the expected result is not always returned. This is caused by events where there is a scope change that is unanticipated which results in an unexpected outcome. - Not all computations return to normal form since there are computations that have forms where no reductions can occur.

Exercise 5: - The smallest λ-expression that does not reduce to normal form is an omega combinator.
- $\omega = \lambda x.xx$

Exercise 6: - Used debugger

Exercise 7:

- ((m.n. mn) (f.x. f(fx))) (f.x. f(f(fx)))
- Gets interpreted to
- (Var5.((f.(x.(f (f (f x))))) ((f.(x.(f (f (f x))))) Var5)))

Exercise 8:

```
12: eval (((\m.(\n.(m n))) (\f.(\x.(f (f x))))) (\f.(\x.(f x))))
    39: eval ((\m.(\n.(m n))) (\f.(\x.(f (f x)))))
        25: eval (\f.(\x.(f (f x))))
        19: eval (\f.(\x.(f x)))
    51: substitute ((\m.(\n.(m n))), (\f.(\x.(f (f x)))), m)
        45: eval ((\f.(\x.(f (f x)))) (\f.(\x.(f x))))
            29: eval (\f.(\x.(f x)))
            55: substitute ((\f.(\x.(f (f x)))), (\f.(\x.(f x))), f)
                38: eval (\x.(f (f x)))
                    40: substitute ((\x.(f (f x))), x, x)
                        22: eval (f (f x))
                            14: eval (f x)
```

Comments and Questions

Throughout the course, we've explored countless algorithms and calculations that work recursively. What makes it so difficult to revert any expression, that has been evaluated using a recursive algorithm, back to its original state?

In the course, we've seen that lambda expressions can be reduced into normal form, what exactly is the benefit of having theses expressions displayed in normal form?

2.9 Week 10

Notes

2n → n/2
2n+1 → 3n+1

c(n) :=
- n/2 n even
- 3n+1 n odd

Conjecture: stabilizes at 1, no matter the initial n.

Rewriting:

Purpose: Formalism, to capture and analyze the process of transforming syntax (aka strings).

Examples:

- High school algebra
- CFGs
- Turing machines
- D/NFA
- MIU save
- Cellular Automata

Definition: an abstract rewrite (or reduction) system ARS is a pair (A, R) with:

- A: set of expressions
- R: set of rewrite rules
- R ¡= A x A a binary relation on A

Computational behavior:
- Confluence and Termination

Homework

1) What did you find most challenging when working through Homework 8/9 and Assignment 3?

When working on Homework 8/9 and Assignment 3, I found it very challenging to both understand and make adjustments to the existing repo which we used for the evaluations. It slowed down my progress by a considerable amount as I had to spend a lot of time understanding the codebase before I could make any changes to it. After finally gaining understanding, I was able to make progress with the exercises and try out the debugger.

2) How did you come up with the key insight for Assignment 3?

I came up with the key insight for Assignment 3 by first understanding the problem and then breaking it down into smaller parts. This process was also enforced by the debugger as it provides the same style of problems solving where we can transform the program into bits and pieces by using the debugger's break points.

3) What is your most interesting takeaway from Homework 8/9 and Assignment 3?

The most interesting takeaway from both assignments, personally, was the fact that the smallest lambda expression that does not reduce to normal form is an omega combinator. This was interesting to me as it showed that not all computations can return to normal form which further supports the format in computation where there are exceptions for almost all rules.

Comments and Questions

Although Langton's Ant has an emergent behavior, is it not the case where when we can visually confirm that the behavior occurs everytime, are we not able to just test the extremes and then be able to verify that this emergent behavior is just its normal behavior and true in all scenarios?

2.10 Week 11

Notes

(2) Terminating if: does ntot admit an infinite computation (a1,a2,a3,...) also called; strictly normalizing.

(3) (weakly) normalizing if every element has some normal form.

Application: Termination analysis of programs → undecidable in general, but can analyze/classify using rewriting theory.

Confluent: If there is no diamond shape in the diagram, it is not confluent.

Unique NF: If we ask if there is a unique normal form, we're asking if all elements have at least one unique normal form.

Properties of ARS: Termination, Normalization, Confluence

ARS: $ba \to ab$
- Is it terminating? (No infinite set)

We can tell that it is terminating if there is a decrease of length in the generating rules.

Not every computation terminates: (a,a,a,a,...)

Termination does imply noramlization.

Homework

1. $A = \emptyset$ and $R = \emptyset$

- **Terminating**: Yes, trivially terminating because there are no elements or relations.
- **Confluent**: Yes, trivially confluent.
- **Unique Normal Forms**: Yes, trivially has unique normal forms because there are no elements or rewrites.

2. $A = \{a\}$ and $R = \emptyset$

- **Terminating**: Yes, trivially terminating because there are no relations to rewrite a.
- **Confluent**: Yes, trivially confluent.
- **Unique Normal Forms**: Yes, a is its own normal form since there are no rewrites.

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

- **Terminating**: No, $a \to a$ is a loop, so there is an infinite rewrite sequence.
- **Confluent**: Yes, trivially confluent because there is only one element with a single rewriting path.
- **Unique Normal Forms**: No, there is no unique normal form since a can be rewritten infinitely to itself.

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

- **Terminating**: Yes, terminating because each element can be rewritten only a finite number of times.
- **Confluent**: No, not confluent because a can be rewritten to either b or c, leading to different results.
- **Unique Normal Forms**: No, there is no unique normal form as a can reach either b or c.

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

- **Terminating**: No, because $a \to a$ creates an infinite loop.
- **Confluent**: Yes, trivially confluent as there are only two elements and no branching rewrites.
- **Unique Normal Forms**: No, a does not have a unique normal form due to the loop.

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

- **Terminating**: No, because $b \to b$ is a loop, allowing infinite rewrites.
- **Confluent**: No, not confluent because a can rewrite to either b or c, leading to different results.
- **Unique Normal Forms**: No, since a can reach different results (either b or c).

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

- **Terminating**: No, both $b \to b$ and $c \to c$ create loops.
- **Confluent**: No, not confluent because a can rewrite to either b or c, leading to different results.
- **Unique Normal Forms**: No, since a can reach different results (either b or c).

Images for the above ARSs

1. Confluent: True. Terminating: True. Has unique normal forms: True.
2. Confluent: True. Terminating: True. Has unique normal forms: False.
3. Confluent: True. Terminating: False. Has unique normal forms: True.
4. Confluent: True. Terminating: False. Has unique normal forms: False.
5. Confluent: False. Terminating: True. Has unique normal forms: True.
6. Confluent: False. Terminating: True. Has unique normal forms: False.
7. Confluent: False. Terminating: False. Has unique normal forms: True.
8. Confluent: False. Terminating: False. Has unique normal forms: False.

1. $A \xrightarrow{*} B$

2. $A\downarrow \xrightarrow{*} B \qquad A \xrightarrow{*} C$

3. $A \longrightarrow B \qquad A \longrightarrow C \qquad B \xrightarrow{*} D \qquad C \xrightarrow{*} D$

4. $A \xrightarrow{*} B \qquad A \xrightarrow{*} C$

5. $A \longrightarrow A\downarrow$

6. $A \longrightarrow A \qquad A \xrightarrow{*} B$

7. $A \longrightarrow A\downarrow \qquad A \xrightarrow{*} B$

8. $A \xrightarrow{*} B \qquad B \longrightarrow B \qquad A \xrightarrow{*} C \qquad C \longrightarrow C$

Comments and Questions

After working with ARSs for this week, are we able to implement any recursive functions that have the ability to solve all of problems that we have encountered so far or are there too many exceptions for that to be the case?

2.11 Week 12

Notes

Properties of ARS

1. Terminating: No infinite computations.
2. Confluence: No diamond shapes which follows the diamond property.
3. Unique normal form: Every x has a unique normal form. Meaning, there exists a unique normal form for every element.
4. Normalizing: Every element has a normal form. Not necessarily uniquely.

Example:

- Bubble sort (ba $\rightarrow$ ab)
- Calculator: evaluator for context free grammar of arithmetic
- λ-calculus, Beta-reduction

Church-Rosser property: If $y \xleftrightarrow{*} z => y \downarrow z$

Confluence: $y \xleftarrow{*} x \xrightarrow{*} z => y \downarrow z$

Homework

Exercise 1:

$ab \to ba$

- The ARS terminates because once all "ab"s are converted into "ba", they can't be converted back, and thus the algorithm terminates.
- "ab" becomes "ba" or "abab" becomes "baba"
- The ARS is confluent because for any input, regardless of which "ab" is chosen to be converted to "ba", we end up with a single, predictable normal form which provides a unique result.
- The ARS implements a sorting specification.

Exercise 2:

$aa \to a$
$bb \to a$
$ab \to b$
$ba \to b$

- This ARS terminates because the rules reduce the length of the string which will eventually reach a normal form.
- The normal forms are either "a" or "b" since it depends entirely on the original input string.
- There is no string that can reduce to both "a" and "b".
- The ARS is not confluent since any input will result in either "a" or "b" as the normal form which means that they do not have unique result.
- Strings that share the same final normal form would become equal.
- A string is equivalent to a if it contains an even number of "b"s, or it is equivalent to be if it contains an odd number of "b"s.
- Create a counter for the number of "b"s in the string. If the result of mod 2 to the counter is 0, it is even, and thus "a". However, if the counter is 1, it is odd and the result is "b".
- This algorithm implements a parity checker for the number of "b"s in the string.

Exercise 3:

$aa \to a$
$bb \to a$
$ba \to ba$
$ab \to ab$

- This ARS does not terminate since any input "ab" or "ba" will infinitely loop between the two states since they can be converted back and forth.
- Strings that don't contain "ab" or "ba" will have a normal form of "a" or "b". However, strings that include "ab" or "ba" will be forced to cycle between the two and not have a normal form.
- Remove the rules $ab \to ab$ and $ba \to ba$ and keep the other two.
- The specification implemented is simplification of strings based on indempotence.

Exercise 4:

$ab \rightarrow ba$
$ba \rightarrow ab$

- This ARS does not terminate since the rewrite rules only support an infinite cycle/loop.
- There are no normal forms since the strings will infinitely cycle between "ab" and "ba".
- $aa \rightarrow a$ $bb \rightarrow a$ $ba \rightarrow ba$ $ab \rightarrow ab$
- The specification implemented is dominance check.

Exercise 5:

$ab \rightarrow ba$
$ba \rightarrow ab$
$aa \rightarrow$
$b \rightarrow$

- $abba \rightarrow aa \rightarrow$
  $bababa \rightarrow aaa \rightarrow a$
- The ARS is not terminating because when the ARS is caught in the cycle, it will never use the third or fourth rule, causing an infinite cycle with no termination.
- There are two classes. 1. Strings, whose counter of "a" mod 2 = 0, that are reduced to an empty string with a normal form of "". 2. Strings, whose counter of "a" mod 2 = 1, that are reduced to a normal form of "a".
- Remove the first two rules which breaks the infinite loop. This maintains the same parity and ensures that the same normal form outputs are maintained.
- 1. Does this string reduce to an empty string? 2. Does this string reduce to an "a"?

Exercise 5b:

$ab \rightarrow ba$
$ba \rightarrow ab$
$aa \rightarrow a$
$b \rightarrow$

- $abba \rightarrow aa \rightarrow a$
  $bababa \rightarrow aaa \rightarrow aa \rightarrow a$
- The ARS is not terminating because when the ARS is caught in the cycle, it will never use the third or fourth rule, causing an infinite cycle with no termination.
- There are two classes. 1. Strings, that contain no "a"s, that are reduced to an empty string with a normal form of "". 2. Strings, that contain any number of "a"s, that are reduced to a normal form of "a".
- Remove the first two rules which breaks the infinite loop. This maintains the same parity and ensures that the same normal form outputs are maintained.
- 1. Does this string reduce to an empty string? 2. Does this string reduce to an "a"?

Comments and Questions

When dealing with the normalization of ARSs, is it possible to prevent infinite loops through a designating the order in which rules must be applied or the number of times a rule can be applied?

2.12 Week 13

Notes

There is a riddle where 10 people with hats, either black or white, are staring in the same direction. This means that the person in the back can see the 9 people in front of them, the person in the 9th position can see the 8 people in front of them, and so on. A person can only guess one color and make no other means of communication. One mistake is allowed.

The solution is to have a person say "white", if there is an even number of white hats in front of them. Furthermore, if there is an odd number of white hats, they should say "black".

With the addition of the new grammar lark rules in milestone 2, we can begin to see the foundations of a programming language.

Homework

1. let rec fact = λn. if $n = 0$ then 1 else $n * \text{fact}(n-1)$ in fact 3
   $\rightarrow$ (definition of let rec)
2. let fact = (fix (λfact. λn. if $n = 0$ then 1 else $n * \text{fact}(n-1)$)) in fact 3
   $\rightarrow$ (definition of let)
3. (λfact. fact 3)(fix (λfact. λn. if $n = 0$ then 1 else $n * \text{fact}(n-1)$))
   $\rightarrow$ (beta reduction: substitute fact)
4. (fix (λfact. λn. if $n = 0$ then 1 else $n * \text{fact}(n-1)$)) 3
   $\rightarrow$ (definition of fix)
5. (λfact. λn. if $n = 0$ then 1 else $n * \text{fact}(n-1)$)(fix (λfact. λn. if $n = 0$ then 1 else $n * \text{fact}(n-1)$)) 3
   $\rightarrow$ (beta reduction: substitute fact)
6. (λn. if $n = 0$ then 1 else $n *$ (fix (λfact. λn. if $n = 0$ then 1 else $n * \text{fact}(n-1)$))$(n-1)$) 3
   $\rightarrow$ (beta reduction: substitute n)
7. if $3 = 0$ then 1 else $3 *$ (fix (λfact. λn. if $n = 0$ then 1 else $n * \text{fact}(n-1)$))$(3-1)$
   $\rightarrow$ (evaluate if)
8. $3 *$ (fix (λfact. λn. if $n = 0$ then 1 else $n * \text{fact}(n-1)$)) $(3-1)$
   $\rightarrow$ (arithmetic: $3-1$)
9. $3 *$ (fix (λfact. λn. if $n = 0$ then 1 else $n * \text{fact}(n-1)$)) 2
   $\rightarrow$ (definition of fix)
10. $3 *$ ((λfact. λn. if $n = 0$ then 1 else $n * \text{fact}(n-1)$)(fix (λfact. λn. if $n = 0$ then 1 else $n * \text{fact}(n-1)$))) 2
    $\rightarrow$ (beta reduction: substitute fact)
11. $3 *$ (λn. if $n = 0$ then 1 else $n *$ (fix (λfact. λn. if $n = 0$ then 1 else $n * \text{fact}(n-1)$))$(n-1)$) 2
    $\rightarrow$ (beta reduction: substitute n)
12. $3 *$ (if $2 = 0$ then 1 else $2 *$ (fix (λfact. λn. if $n = 0$ then 1 else $n * \text{fact}(n-1)$))$(2-1)$)
    $\rightarrow$ (evaluate if)
13. $3 * (2 *$ (fix (λfact. λn. if $n = 0$ then 1 else $n * \text{fact}(n-1)$))$(2-1)$)
    $\rightarrow$ (arithmetic: $2-1$)
14. $3 * (2 *$ (fix (λfact. λn. if $n = 0$ then 1 else $n * \text{fact}(n-1)$)) 1)
    $\rightarrow$ (repeat steps for $n = 1$ and $n = 0$ as above)
    $\rightarrow$ (final arithmetic)
15. $3 * (2 * 1) = 6$

Comments and Questions

Are there any forms of recursion that we can apply towards the hat riddle that would make it simpler to solve? Even accounting for the event where there are multiple colors?

3 Lessons from the Assignments

Milestone 1

From the moment of its creation, I assumed a leadership role for this project by immediately setting up a group chat to facilitate communication and coordination among team members. Next, I created the GitHub repository and established clear guidelines on how we would utilize branches to manage our work.

Additionally, I separated our progress with each milestone within branches to act as an easy to use version control. This way, if our future milestones had taken a bad turn, we could just reset with the progress made in the previous milestone. This structure ensured that we could collaborate efficiently while minimizing the risk of conflicts or errors.

Beyond these organizational efforts, during the first milestone I worked closely with my teammates to implement the required modifications to the `evaluate` and `substitute` functions for the new grammar. These changes were integral to incorporating the grammar Lark file that enabled addition, multiplication, and subtraction operations. This experience deepened my understanding of how interpreters for programming languages function. Specifically, I learned how to transform equations using a grammar Lark file and manage the order of operations, ensuring that multiplication took precedence over addition. These lessons not only enhanced my technical skills but also provided valuable insights into the inner workings of interpreters.

Something I found especially noteworthy was the appearance of recursion, a topic which we had recently spoken of plenty. After viewing the code for the Tower of Hanoi solution, it became apparanent that the interpreter's task was extremely similar in almost every facet. For example, just like how the disks of the Tower of Hanoi had to follow a similar pattern, so did our interpreter. An example of this is how our interpreter needed to perform normalization to every equation before beginning to perform an operator on its components.

Milestone 2

For the second milestone, my primary contribution was to integrate the new grammar into the existing grammar Lark file. Once again, I updated the `evaluate` and `substitute` functions to accommodate these changes. This phase of the project introduced new challenges, particularly in the realm of comparison operations. Through my work on this milestone, I gained a solid understanding of how comparisons are conducted within an interpreter and how the interpreter can compare values. More specifically, how interpreters can do so by utilizing the same mathematical thinking that was used in discrete mathematics.

Unfortunately, milestone 2 also highlighted areas where I needed additional support. Specifically, I struggled to grasp the implementation of the logic behind how parentheses works within our grammar lark file. The additional layer of priority and complete formation of PEMDAS led to confusion on my end. Recognizing my limitations, I turned to my teammates for guidance. Their explanations and contributions helped bridge my knowledge gap and ensured that we could successfully integrate parentheses into the project. I felt that this experience was a valuable learning experience as it took a strong amount of introspection to discover such a knowledge gap and allowed us to move onto the next milestone.

Milestone 3

In the final milestone, I continued to build on my strengths by integrating the new grammar into the grammar Lark file. However, the complexity of this milestone proved to be significantly greater than the previous ones. The use of lists within the programming language was particularly challenging for me, as it required a low-level understanding that I found difficult to grasp. Despite my efforts, I was unable to derive as much learning from this milestone as I had hoped.

One of the key challenges during this phase was balancing the demands of milestone 3 with the impending final report. The deadlines combined with finals forced us to prioritize the final report during class time, which limited the time we could dedicate to fully understanding the grammar in milestone 3. Reflecting on this experience, I believe that both additional in-class time for group work and prioritizing a fleshed out group wide understanding before beginning each milestone would have been beneficial. These changes could have greatly changed the troublesome and difficult experience of milestone 3.

4 Seminars

4.1 Seminar Speaker: Dr. Ansel Teng

Summary

Dr. Ansel Teng's speaker event touched on the applications of AI in world the nonprofit organization fundraising. His works are dedicated towards a SaaS platform which utilizes both donor data and AI models to predict which people are most likely to donate, donate the most, and how to best contact them. Additionally, the SaaS also has lots of filtering which greatly increases the effectiveness of the software.

Question

Would current donors decrease or cease donations after the realization that they are having all of their information extracted after giving a donation? And would the benefits outweigh the cost of that drawback?

4.2 "Creating Your Own Programming Language" by Computerphile

Summary

I watched Computerphile's video "Creating Your Own Programming Language" which is about the process of designing a basic programming language using Python. It begins by constructing an interpreter capable of evaluating expressions with Reverse Polish Notation (RPN), employing a stack-based approach for efficient computation. As the language develops, features like variables are introduced, allowing for value assignment and reuse, along with support for multi-line code. The implementation is further enhanced with logic structures, such as loops and conditionals, which enable the interpreter to handle more complex tasks. An important section was the creation of a factorial function using a while loop, showcasing the versatility provided by combining loops and branching logic. The video also emphasizes practical techniques, such as Python's split method for tokenization and the use of dictionaries for variable management. By the end, the language evolves into a functional tool capable of solving meaningful problems.

Question

How would this programming language be created if not for the use of the C++/Python? How do you create a programming language without a programming language?

4.3 Stratefied Design Lens by Eric Normand

Summary

I listened to Eric Normand's podcast on the stratified design lens, which emphasizes that effective software development isn't about rigidly following rules but about making thoughtful design decisions by clearly separating different layers within an application. This stratified approach helps prevent a system from becoming brittle, as it allows each layer to evolve independently. Furthermore, this It encourages thoughtful organization by placing logic where it truly belongs, making your software more adaptable and maintainable.

Question

My Question: Is it still possible to separate the layers in a clear cut manner even when the model creates both complex rules and flexibility at the same time?

4.4 "Programming Loops vs Recursion" by Computerphile

Summary

I watched the "Programming Loops vs Recursion" video by Computerphile, which delved into the history and progression of loops in programming. The video highlighted the shift from early assembly languages to the emergence of high-level languages like FORTRAN. As programming languages evolved, nested loops were introduced, greatly enhancing the power of loops. This concept of nesting loops laid the groundwork for understanding the importance of recursion in programming. One notable takeaway was that nested loops enabled more complex calculations, making it possible to handle multi-dimensional problems like 2D arrays. However, even with the power of nested loops, there were still limitations, leading to the need for recursion to tackle more complex tasks, such as solving Ackermann's function. The video also touched on how recursion is fundamental to compiler development, which relates to our class where we created lambda-calculus compilers.

Question

Would using earlier assembly languages to teach recursion help give students a better idea of how recursion works due to the elimination of any notable abstraction?

4.5 "Coding a Web Server in 25 Lines" by Computerphile

Summary

In the Computerphile's video, "Coding a Web Server in 25 Lines," Dr. Laurence Tratt built a basic web server in Rust with just 25 lines of code. He starts off by creating a "listener" that only responds to his IP address to avoid any need to implement server security. Next up, he establishes the basic communication for a server, where when there is a request, the server listens, and there is a response. Lastly, he cuts a lot of corners and has the browser do the majority of the work for the response and is able to create a simple text response for the page with just one line.

Question

Is this simplicity in web server design achievable with other languages that are dynamically typed such as Python?

5 Conclusion

From my perspective, in the wider world of software engineering, this course acts as a bridge between discrete math and any introductory python/java/c++ course. What I mean by this is that programming language seems to describe exactly how we're able to utilize the basic calculations done in discrete math to create a programming language that uses it as building blocks. In total, it provides a solid foundation to understand exactly how computers are able to perform calculations using the most basic of operations. Additionally, this course seeks to provide students an understanding of prompt engineering and how we can use LLMs on a larger scale than just asking it a simple question; similar to how we use google.

Personally, the most interesting topic in this course was our use of grammar lark files. It was interesting to see how we could use a simple grammar file to define how the mathematical operators would work in a language. I appreciated the pacing of the unit since we first started off with implementing addition and multiplication and then moved onto division and parentheses. I also found stressing over elements such as parentheses or left-to-right priority which I had not stopped to consider the implementation of before. I believe through this unit I developed the ability to think more critically about how all of the parts work together in the lowest levels of abstraction within software. Lastly, to see the grammar come together at the end and being able to operate within the rules of PEMDAS perfectly gave a sense of accomplishment and progress regarding the course which helped build understanding and motivation towards learning more.

As for improvements in the course, I would suggest a more high-level description of each unit during their introduction so that the students have a better idea of the unit's goal. I believe that creating a basic and easy to understand goal is the first step in developing an interest in the learning of a topic. Otherwise units can just become a system where students just go through the motions without any overarching understanding of the importance of the concepts. Moving on, I understand that since LLMs are in their infancy that it is difficult to implement them. I found almost all of the coding projects to serve little purpose since it mostly involved prompt engineering randomly until all of the tests cleared. I think it would be better to use assignments that feature near complete code snippets. This way professors can minimize the amount of busy work and just require students to complete a few lines of code that solidify understanding. Since, at the end of the day the goal is understanding and not completing the grammar since it serves no purpose.