

CPSC-354 Report

Brandon Hughes
Chapman University

October 13, 2024

Abstract

Contents

1	Introduction	2
2	Week by Week	2
2.1	Week 1	2
2.1.1	Notes	2
2.1.2	Homework	2
2.1.3	Comments and Questions	3
2.2	Week 2	4
2.2.1	Notes	4
2.2.2	Homework	4
2.2.3	Comments and Questions	6
2.3	Week 3	6
2.3.1	Homework	6
2.4	Week 4	7
2.4.1	Notes	7
2.4.2	Homework	7
2.4.3	Comments and Questions	8
2.5	Week 5	9
2.5.1	Notes	9
2.5.2	Homework	9
2.5.3	Comments and Questions	10
2.6	Week 6	10
2.6.1	Notes	10
2.6.2	Homework	10
2.6.3	Comments and Questions	11
2.7	Week 7	11
2.7.1	Notes	11
2.7.2	Homework	12
2.7.3	Comments and Questions	12
3	Lessons from the Assignments	12
4	Conclusion	12

1 Introduction

2 Week by Week

2.1 Week 1

2.1.1 Notes

During this week, there was a review of Git and being introduced to Latex and Lean. Some helpful commands include git add, commit, status, and push. Through the website "https://sudorealm.com/blog/how-to-write-latex-documents-with-visual-studio-code-on-mac", we set up latex to be able to complete the weekly report.

2.1.2 Homework

1. Finish the Natural Number Game Tutorial World.
 - a) Show the completed work for levels 5 through 8.
 - b) For one level, explain in detail how the Lean proof is related to its corresponding proof in mathematics.

1a. Show the completed work for levels 5 through 8.

Level 5: Prove that $a+(b+0)+(c+0)=a+b+c$.

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

Level 6: Prove that $a+(b+0)+(c+0)=a+b+c$.

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

Level 7: Prove that for all natural numbers a , we have $\text{succ}(a)=a+1$.

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

Level 8: Prove that $2+2=4$.

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

1b. For one level, explain in detail how the Lean proof is related to its corresponding proof in mathematics. For level 7, we had to prove the theorem of the $\text{succ}(n)$ is also equal to $n+1$.

Lean Proof:

```
Start: succ(n) = n + 1
1: rw[one_eq_succ_zero]
Result: succ n = n + succ 0
2: rw[add_succ]
Result: succ n = succ (n + 0)
```

```
3: rw[add_zero]
End: succ n = succ n
```

Thus, proving reflexivity.

Proof by Mathematics: By using the induction we are able to prove the theorem of the $\text{succ}(a)$ is equal to $a+1$.

Base Case:

Consider $n = 0$,

$$S(0) = 0 + 1$$

$$0 + 1 = 1$$

Thus, $0 + 1 = S(0)$, which holds true.

Inductive hypothesis:

Assume for some natural number k that $k + 1 = S(k)$.

Inductive Step:

We need to show that $k + 1 + 1 = S(k + 1)$.

$(k + 1) + 1 = S(k + 1)$, by adding parenthesis

$S(k) + 1 = S(k + 1)$, by using the inductive hypothesis.

$S(k + 1) = S(k + 1)$, by using addition of successors.

Thus, proving reflexivity.

Through these steps, we can see that the end goal of proving reflexivity on both the Lean proof and its corresponding proof in mathematics. The similarities come from the lean proof and the inductive step however, as they are similar steps in being able to prove the theorem. The lean proof is more straight forward because instead of proving it through a basis, inductive hypothesis, and inductive step, you only have to prove it through rewriting the equation so that both sides are equal. Which is done through the inductive step of the mathematics proof.

2.1.3 Comments and Questions

When looking at Formal Systems from the textbook, we are given this example of an impossible puzzle to solve. The MU problem, where you are given a set of rules and have to obtain MU from MI, however, its impossible because you can never end up without having an odd number of I's inside the string. When it comes to Formal systems, and solving them a lot of times people will look towards actually doing compared to trying to assess the logic behind this however computers, mostly AI, generally start at the logic. How might combining human intuition and AI's logical reasoning lead to more effective problem-solving strategies? Would we be able to solve problems quicker, or would AI's logical reasoning overtake the human trial and error method?

2.2 Week 2

2.2.1 Notes

During this week, we learned in class that both Math and Lean can be seen as languages. Math is a specification language while Lean is a programming language. A specification language is used to define the requirements and properties of a system. A Math proof can be written into Lean proof very easily since they use and follow the same rules when it comes to solving theorems and problems. The difference between the two proofs however is that in Math we typically reason forwards from the problem to the answer, while in Lean we reason backwards from the answer to the problem. We could also do it the opposite way but it would become more challenging. Another idea that we learned in class is that a recursive data type, which could also be called an algebraic data type, and induction are of similar processes as you define what a number is inside of a number. An example of recursion can be seen in the Tower of Hanoi as you are solving the previous tree in the next tree. Tower of Hanoi are also similar to binary search trees since the amount of nodes in a amount of 'n' level of a tree, if you have a balanced tree is the same amount of moves it takes to solve when you have 'n' amount of disks. Lastly, we also learned that when you write a recursive program it creates a stack behind the scenes to solve all the problems, it will always go to the one on top rather than starting back at the start of the problem. If you don't have a stack you could also write it on a rewriting machine.

2.2.2 Homework

1. Finish the Natural Number Game Addition World.

a) Show the completed work for levels 1 through 5.

b) For level 4 or 5, explain in some detail how the Lean proof is related to its corresponding proof in mathematics

1a.

Level 1:

$Sd = Sd$	<code>rfl</code>
$S(0 + d) = Sd$	<code>rw[hd]</code>
$0 + Sd = Sd$	<code>rw[add_succ]</code>
$0 = 0$	<code>rfl</code>
$0 + 0 = 0$	<code>rw[add_zero]</code>
$0 + n = n$	<code>induction n with d hd</code>

Level 2:

$SS(a + d) = SS(a + d)$	<code>rfl</code>
$S(Sa + d) = SS(a + d)$	<code>hd</code>
$S(Sa + d) = S(a + Sd)$	<code>rw[add_succ]</code>
$Sa + Sd = S(a + Sd)$	<code>rw[add_succ]</code>
$Sa = Sa$	<code>rfl</code>
$Sa = S(a + 0)$	<code>rw[add_zero]</code>
$Sa + 0 = S(a + 0)$	<code>rw[add_zero]</code>
$Sa + b = S(a + b)$	<code>induction b with d hd</code>

Level 3:

$S(d + a) = S(d + a)$	rfl
$S(a + d) = S(d + a)$	rw[hd]
$S(a + d) = Sd + a$	rw[succ.add]
$a + Sd = Sd + a$	rw[add.succ]
$a = a$	rfl
$a = 0 + a$	rw[zero.add]
$a + 0 = 0 + a$	rw[add.zero]
$a + b = b + a$	induction b with d hd

Level 4:

$S(a + (b + d)) = S(a + (b + d))$	rfl
$S(a + b + d) = S(a + (b + d))$	rw[hd]
$S(a + b + d) = a + S(b + d)$	rw[add.succ]
$S(a + b + d) = a + (b + Sd)$	rw[add.succ]
$a + b + Sd = a + (b + Sd)$	rw[add.succ]
$a + b = a + b$	rfl
$a + b = a + (b + 0)$	rw[add.zero]
$a + b + 0 = a + (b + 0)$	rw[add.zero]
$a + b + c = a + (b + c)$	induction c with d hd

Level 5:

$S(a + d + b) = S(a + d + b)$	rfl
$S(a + b + d) = S(a + d + b)$	rw[hd]
$S(a + d + b) = S(a + d) + b$	rw[succ.add]
$S(a + d + b) = a + Sd + b$	rw[add.succ]
$a + b + Sd = a + Sd + b$	rw[add.succ]
$a + b = a + b$	rfl
$a + b = a + 0 + b$	rw[add.zero]
$a + b + 0 = a + 0 + b$	rw[add.zero]
$a + b + c = a + c + b$	induction c with d hd

1b.

For Level 4, we are proving the associativity of addition. On the set of natural numbers, addition is associative. In other words, if a,b and c are arbitrary natural numbers, we have $(a+b)+c=a+(b+c)$. In Math and Lean, we have to do a proof by induction on c.

Math Proof:

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

Base Case: $(a + b) + 0 = a + (b + 0)$

$$\begin{aligned}(a + b) + 0 &= a + (b + 0) \\ a + b &= a + (b + 0) && \text{def of } + \\ a + b &= a + b && \text{def of } +\end{aligned}$$

Induction Step: $(a + b) + Sd = a + (b + Sd)$

Induction Hypothesis: $(a + b) + d = a + (b + d)$

$$\begin{aligned}(a + b) + Sd &= a + (b + Sd) \\ S((a + b) + d) &= a + (b + Sd) && \text{def of } + \\ S((a + b) + d) &= a + S(b + d) && \text{def of } + \\ S((a + b) + d) &= S(a + (b + d)) && \text{def of } + \\ S(a + (b + d)) &= S(a + (b + d)) && \text{Induction Hypothesis}\end{aligned}$$

The Lean proof written above is the exact same as the same steps in the Math proof just backwards. Instead of having `add_zero` and `add_succ`, we have the definition of addition as that can be proven to add both successors and zero to numbers.

2.2.3 Comments and Questions

In the beginning of the reading, it takes about how recursion is different from paradox or infinite regress, since it never defines something in terms of itself, but always in terms of simpler versions of itself. Is the only difference between a paradox and a recursively solvable problem be that it has an exit statement at its very simplest version or are there more differences? If some paradoxes were proposed recursively, would we be able to break down some harder problems into simpler version to prove if they are unsolvable logically?

2.3 Week 3

2.3.1 Homework

For this homework assignment, I used ChatGPT to explore language interoperability, which refers to how different programming languages interact with one another to create a single system or application. In modern software development and application creation, it is common for multiple programming languages to be used together to meet different kinds of functions and performance requirements. As a result, being able to integrate between these languages flawlessly is essential for building efficient and reliable systems. This report goes over some of the current methods used to ensure compatibility between programming languages

and investigates ways to enhance these systems further. While many languages already possess some interoperability with one another, challenges arise when trying to bridge the gaps between the languages that were not designed to work together. These challenges can lead to issues such as performance inefficiencies, data type mismatches, and increase complexity in code maintenance. By examining these difficulties, potential improvements can be seen to create more solutions for the future. Additionally, advancements in tools, frameworks, and compiler technology could further enhance interoperability efforts. By improving in ways in which programming languages collaborate, developers will be able to build more versatile, efficient, and scalable systems. Addressing these challenges and making enhancements will lead to more seamless development processes and better applications.

https://github.com/Brandon-Hughes/LLM_Literature_Review

2.4 Week 4

2.4.1 Notes

Concrete syntax considers a program as a string in the form of a sequence of characters. This is the syntax that we are used to seeing and interacting with on a day to day basis. Abstract syntax considers a program as a tree-like structure that represents the program's structure and organization. An Abstract Syntax Tree (AST) is a tree representation of the abstract syntactic structure of source code written in a programming language. ASTs can write type checkers, interpreters, and compilers via recursion on these algebraic data types, but translating strings into trees is very difficult. The main idea is that you can translate a concrete syntax (string) into a concrete syntax tree, but translating a concrete syntax tree (string) into an abstract syntax tree (tree) is very difficult. Parsing is about putting the parentheses in the correct position. A context-free grammar is a set of rules that describe how to form strings in a language. The purpose of a context-free grammar is to define a set of strings or a language, in which a particular string in the language can be derived from the start symbol. The symbols that will appear in the strings (terminals), are those that are enclosed in single quotes. Other symbols may never appear in the parsed string, but only control which strings can be derived. These are called non-terminals.

2.4.2 Homework

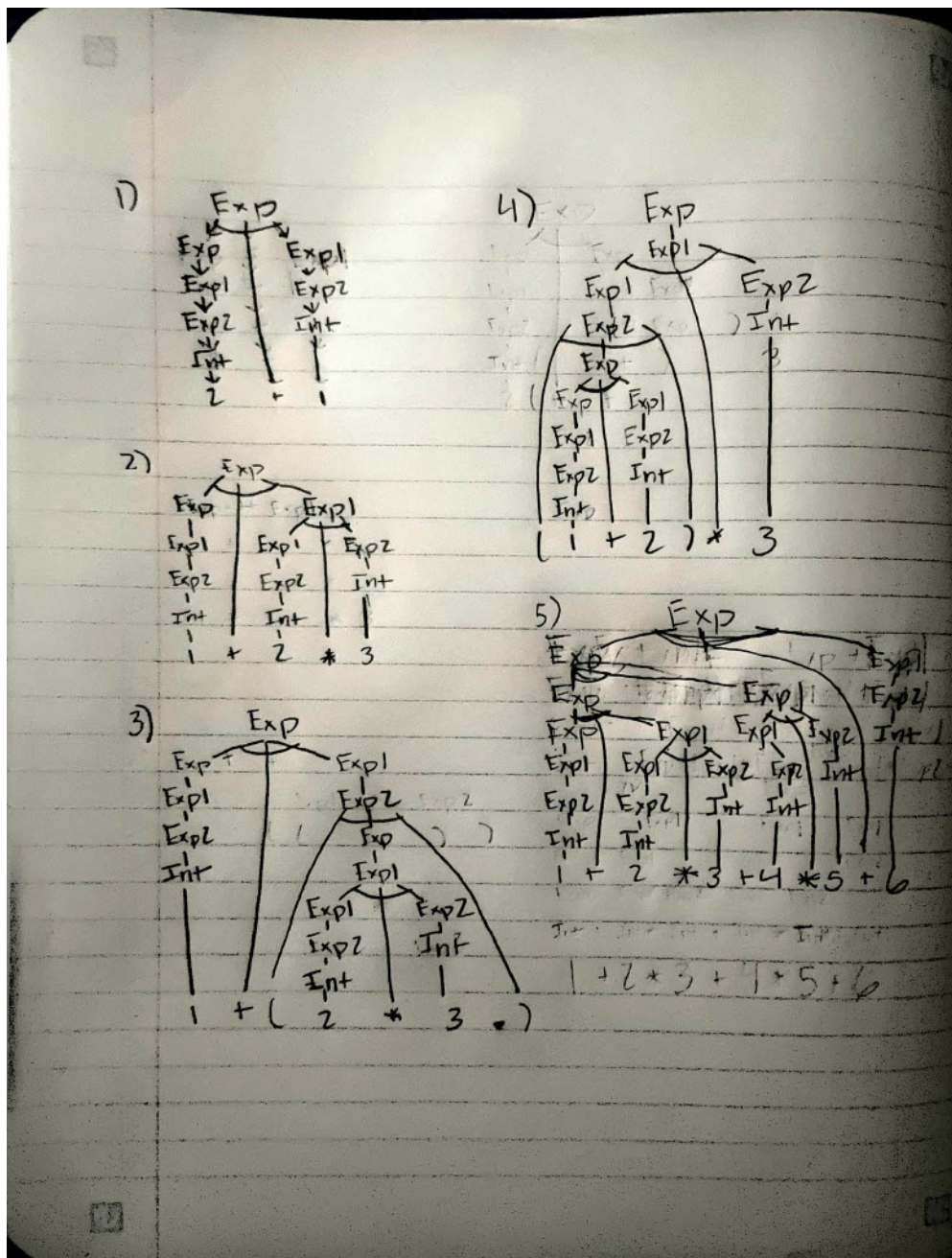
Using the context-free grammar:

$$\begin{aligned}Exp &\rightarrow Exp + Exp1 \\Exp1 &\rightarrow Exp1Exp2 \\Exp2 &\rightarrow Integer \\Exp2 &\rightarrow (Exp) \\Exp &\rightarrow Exp1 \\Exp1 &\rightarrow Exp2\end{aligned}$$

Generate the abstract syntax tree for the expressions.

1. $2+1$
2. $1+23$
3. $1+23$
4. $1 + 23$
5. $1+23+45+6$

The Abstract Syntax Trees for Problems 1 through 5 are shown below:



As we can see from the image above, each of the problems can be solved by using the rules of the context-free grammar with the start symbol Exp .

2.4.3 Comments and Questions

During this week we've explored the idea of applying context-free grammar in the basics of Math and we are going to start applying that into programming languages. Besides the usage in math and programming languages, are we able to see or use context-free grammar in other examples or in our lives?

2.5 Week 5

2.5.1 Notes

By practicing the application of context-free grammars (CFG) in a programming language, a calculator was created that took an input and broke it down into an Abstract Syntax Tree (AST) using a CFG. By breaking the expression into a AST, the problem was solved by reading through the AST to produce a correct result for the expression. This application served to practice parsing expressions into smaller pieces that are more easily read by programming languages, as well as provide an introduction to context-free grammars.

The second portion of this class week focused on an introduction and tutorial of constructive logic. Through the Lean logic game, students were introduced to how to write math expressions as logic-driven trees, similar to how a CFG might break an expression into components. While the process is similar, it differs by working with logical inferences instead of numbers. The keyword "exact" is used to give the game's answer, which serves as the concluding statement. The symbol \wedge denotes a logical "and," meaning the logic must follow both A and B. The logical "and" can be broken apart using the keywords ".left" and ".right" to access either the left or right item of the \wedge , respectively.

2.5.2 Homework

1. Finish the Lean Logic Game Tutorial World.
 - a) Show the completed work for levels 1 through 8.
 - b) For level 8, write down the proof in mathematical logic.

1a.

Level 1: If todo_list is P, then P.

exact todo_list

Level 2: If p is P and s is S, then $P \wedge S$.

exact ⟨p,s⟩

Level 3: If a is A, i is I, o is O, and u is U, then $A \wedge I \wedge O \wedge U$.

exact ⟨a,⟨i,⟨o,u⟩⟩⟩

Level 4: If vm is $P \wedge S$, then P.

exact vm.left

Level 5: If h is $P \wedge Q$, then Q.

exact vm.right

Level 6: If h1 is $A \wedge I$ and h2 is $O \wedge U$, then $A \wedge U$.

exact ⟨h1.left,h2.right⟩

Level 7: If h is $(L \wedge ((L \wedge C) \wedge L) \wedge L \wedge L \wedge L)$, then C.

exact h.left.right.left.left.right

1b.

Level 8: If $((P \wedge S) \wedge A) \wedge \neg I \wedge (C \wedge \neg O) \wedge \neg U$ then $A \wedge C \wedge P \wedge S$

(1)	assumption1 : $((P \wedge S) \wedge A) \wedge \neg I \wedge (C \wedge \neg O) \wedge \neg U$	
(2)	A	and_left and_right on (1)
(3)	C	and_right and_right and_left and_left on (1)
(4)	P	and_left and_left and_left on (1)
(5)	S	and_left and_left and_right on (1)
(6)	$A \wedge C \wedge P \wedge S$	and_intro on (2) (3) (4) (5)

2.5.3 Comments and Questions

In the creation of a context-free grammar, we can see that by establishing a hierarchy through constructs like exp, exp1, exp2, etc., we can control the order of operations. How can programmers determine how deep the hierarchy should go to ensure the grammar is well-structured and the language doesn't break?

2.6 Week 6

2.6.1 Notes

Lambda calculus is a programming language comprised of only variables and functions. Two main portions build lambda calculus, the syntax and the semantics. The syntax is made of three constructions abstraction, application, and variables. Abstraction is taking the program and using a placeholder variable so that it can change what you use. It's the creation of general-purpose functions that can take in many different inputs but will produce a structured output. Application is how we apply functions to one another. Lastly, variables are just variables. An important part of Lambda calculus is dropping the parenthesis for easier readability. You apply the left and move to the right to solve for lambda in the same parenthesis. The next portion of Lambda calculus is the semantics or how to execute it. By executing, we are either inputting numbers into variables or simplifying them into their lowest form. When having multiple inputs and variables within the same expression, apply it in the same parentheses or the farthest lambda to the left if you only have the function in one parenthesis but multiple inputs and variables. When solving problems, you may arrive at to capture avoiding substitution, when multiple inputs are being placed into a variable, many solutions act differently here it will place it into all of them while some of them place it only into the first variable possible. To solve this, they created bound and free variables where bound variables are set within the function at the beginning, while free variables are those that are inputted.

2.6.2 Homework

Finish the Lean Logic Game Tutorial World. Show the completed work for levels 1 through 9.

Level 1: If bakery_service is $P \rightarrow C$, then C .

exact bakery_service p

Level 2: Show that $C \rightarrow C$.

exact $\lambda h \Rightarrow h$

Level 3: Show that $I \wedge S \rightarrow S \wedge I$.

exact $\lambda h \Rightarrow \langle h.right, h.left \rangle$

Level 4: If $h1$ is $C \rightarrow A$ and $h2$ is $A \rightarrow S$, show that $C \rightarrow S$.

$$\text{exact } \lambda c \Rightarrow h2(h1\ c)$$

Level 5: If $h1$ is $P \rightarrow Q$, $h2$ is $Q \rightarrow R$, $h3$ is $Q \rightarrow T$, $h4$ is $S \rightarrow T$, and $h5$ is $T \rightarrow U$, then U .

$$\text{exact } h5(h3(h1\ p))$$

Level 6: If h is $C \wedge D \rightarrow S$, then $C \rightarrow D \rightarrow S$.

$$\text{exact } \lambda c\ d \Rightarrow h\langle c, d \rangle$$

Level 7: If h is $C \rightarrow D \rightarrow S$, then $C \wedge D \rightarrow S$.

$$\text{exact } \lambda\langle c, d \rangle \Rightarrow h\ c\ d$$

Level 8: If h is $(S \rightarrow C) \wedge (S \rightarrow D)$, then $S \rightarrow C \wedge D$.

$$\text{exact } \lambda s \Rightarrow \langle h.\text{left}\ s, h.\text{right}\ s \rangle$$

Level 9: Show that $R \rightarrow (S \rightarrow R) \wedge (\neg S \rightarrow R)$.

$$\text{exact } \lambda r \Rightarrow \langle \lambda s \Rightarrow r, \lambda s \Rightarrow r \rangle$$

2.6.3 Comments and Questions

What are some alternative solutions to capture-avoiding substitution beyond using bound and free variables?

2.7 Week 7

2.7.1 Notes

Through the usage of syntax and semantics of lambda calculus we are able to create examples of programs that are made. Pure lambda calculus only have functions and we can use those functions to encode data types such as booleans, numbers, and lists. To define a boolean, we define that whenever a function is true it will return the first item but if it's false it will return the second item. For example, $\lambda m\ \lambda n\ m$ is true because it returns the first item while $\lambda m\ \lambda n\ n$ returns false due to it returning the second item. Through this we are able to get an example of how lambda calculus creates functions to get basic outputs. Another example is the creation of numbers through Church numerals. We can define numbers as the amount of times a function is applied onto a variable. For example, 0 being defined as $\lambda f\ \lambda x\ x$, while 1 is $\lambda f\ \lambda x\ f\ x$.

Lambda calculus allows us to use variable binding to name functions using the keyword `let`. Through this we are able to define keywords that can be used to reference a function previously defined in the code.

Another idea that is explored is the implementation of recursive functions through lambda calculus. With the definition of conditionals, numbers and keywords, we are able to investigate terms that reduce (compute) to themselves. This is done through defining the function in a keyword and then using the reducing that definition of the function into itself, with the goal of having the keyword's function appear in the function again.

2.7.2 Homework

1. Reduce the following lambda term: $((\lambda m.\lambda n.mn)(\lambda f.\lambda x.f(fx)))(\lambda f.\lambda x.f(f(fx)))$

$$\begin{aligned} & ((\lambda m.\lambda n.mn) (\lambda f.\lambda x.f(fx))) (\lambda f.\lambda x.f(f(fx))) \\ & (\lambda n. (\lambda f.\lambda x.f(fx))n) (\lambda f.\lambda x.f(f(fx))) \\ & \lambda x. (\lambda f.\lambda x.f(f(fx))) x ((\lambda f.\lambda x.f(f(fx)))x x) \\ & \lambda x. \lambda x1. (\lambda f.\lambda x.f(f(fx))) x ((\lambda f.\lambda x.f(f(fx))) x x ((\lambda f.\lambda x.f(f(fx))) x x x1)) \\ & \lambda x. \lambda x1. (\lambda x1.x(x x1)) ((\lambda f.\lambda x.f(f(fx))) x ((\lambda f.\lambda x.f(f(fx))) x x1)) \\ & \lambda x. \lambda x1. x(x((\lambda f.\lambda x.f(f(fx))) x ((\lambda f.\lambda x.f(f(fx))) x x1)))) \\ & \lambda x. \lambda x1. x(x(x(x(x(\lambda f.\lambda x.f(f(fx))) x x1)))) \\ & \lambda x. \lambda x1. x(x(x(x(x(\lambda x1.x(x x1))) x1)))) \\ & \lambda x. \lambda x1. x(x(x(x(x(x(x x1))))))) \end{aligned}$$

2. Explain what function on natural numbers $(\lambda m.\lambda n.mn)$ implements.

The lambda expression is about applying one numeral to another numeral. Church numerals are a way to represent natural numbers using functions. For example, we can represent 3 through $f(f(f(x)))$ or "applying a function twice". In this expression, m is a numeral and n is another numeral and what it does is apply m to n . It takes two numerals or numebrs and applys one to the other like functions. This gives a way to combine the numerals and numbers using function application and not a direct operations such as addition or multiplication.

2.7.3 Comments and Questions

With the definition of lambda calculus in the basis of many programming languages, are there any issues that come when modeling complex data structures and alogrithms? If there are, how do programmers overcome them?

...

3 Lessons from the Assignments

4 Conclusion

References

[label] Andrew Moshier, [Contemporary Discrete Mathematics](#), M&H Publishing, 2024