

CPSC-354 Report

Gabriel Davidson
Chapman University

December 15, 2024

Abstract

This report examines the progression of concepts within the umbrella of programming languages theory through 13 of weeks of theory-driven lecture, homework, and assignments. The driving concepts are lambda calculus and associated content, especially with respect to theorems and proofs, recursion as a programming strategy, parsing theory and its applications on interpreted languages, and context free grammars and the parsing of input into abstract syntax trees. We implemented the theory learned in class into 4 programming assignments that built off of each other to explore foundational topics, especially recursion, context-free grammars, abstract syntax trees, and the implementation of interpreters for custom languages. The final assignment culminates with an interpreter capable of handling complex operations, including conditional logic, variables, if-then-else blocks, lists, and recursive functions. Through these assignments, and through exercises practicing concepts core to the class's curriculum, how to implement meaningful concepts into concrete behaviour.

Contents

1	Introduction	2
2	Week by Week	2
2.1	Week 1	2
2.2	Week 2	3
2.3	Week 3	5
2.4	Week 4	8
2.5	Week 5	9
2.6	Week 6	11
2.7	Week 7	12
2.8	Week 8	12
2.9	Week 9	12
2.10	Week 10	13
2.11	Week 11	14
2.12	Week 11	14
2.13	Week 12	17
2.14	Week 13	17
3	Lessons from the Assignments	18
3.1	Assignment 1	18
3.2	Assignment 2	19
3.3	Assignment 3	20
3.4	Assignment 4	20
3.4.1	Milestone 1	20
3.4.2	Milestone 2	21

3.4.3 Milestone 3	22
4 Conclusion	22

1 Introduction

Programming languages serve as the basis for computer science and enable the expression of complex behaviour, systems, and problems into executable programs. Languages allow develops to take on these problems, just as spoken languages allow humans to express themselves, by providing syntax and semantics. The theory behind programming languages is a huge canvas of concept and topics, and in this class we take on understanding the role of lambda calculus and its associated content, parsing theory, and context free grammars, among others. These topics are central to understanding the origins of programming languages as written languages, and also to designing and implementing custom languages of our own.

This report will focus on that journey, from proving the validity of logic programs to creating our own context free grammars. By integrating theory with practical implementations, this course has taught me how abstract ideas can translate into tangible behaviour, and how we can utilize that behaviour to capture the essence of written syntax and transform (interpret!) it into executable logic.

2 Week by Week

2.1 Week 1

Notes

This week in class we mainly focused on learning Lean, setting up L^AT_EX, and a brief lecture on the basis of *Proof = Program*. This idea shows how logical mathematical proofs are a constructive process, building on previously founded theorems and definitions. This idea transfers to theoretical programming in the sense that programs are also constructed proofs. The execution of a program is the execution of many logical steps in a proof. I enjoy how this relates to the "human" process as well - our activities are the execution of previously learned strategies in a logical way.

Question

I was interested in learning more about Proof = Program, and some additional sleuthing showed me how a running a program is simply executing the steps in a logical proof. I was wondering how more complex program design methods such as recursion adhere to this idea, and if there are other programming methodologies that do not adhere to Proof = Program?

Homework

Level 5 - Adding Zero

In this level we prove the theorem that $\mathbf{a+0=a}$ using $\mathbf{a+(b+0)+(c+0)=a+b+c}$. Here is how I solved this theorem.

```
repeat rw add zero
rfl
```

Level 6 - Adding Zero

In this level we built on the solution from the previous level to learn how to use precision rewriting.

```
rw add zero c
```

```
repeat rw add zero
rfl
```

Level 7 - Add Suc

In this level we prove the theorem that $\text{succ}(a)=a+1$.

```
rw one eq add zero
repeat rw add zero
rfl
```

Level 8 - Add Suc

In this level we prove the equation that $2+2=4$. This was the final level in the Tutorial World, and required the accumulation of definitions and theorems learned so far. I will provide the assumptions that I used when deciding my proof in Lean.

```
rw four eq succ three — Any number  $n = \text{succ}(\text{pred}(n))$ 
rw three eq succ two — Any number  $n = \text{succ}(\text{pred}(n))$ 
rw two eq succ one — Any number  $n = \text{succ}(\text{pred}(n))$ 
rw one eq succ zero — Any number  $n = \text{succ}(\text{pred}(n))$ 
repeat rw add succ — Using  $a + \text{succ } b = \text{succ}(a + b)$ 
rw add zero — Using  $a + 0 = a$ 
rfl — Proves the goal  $X = X$ 
```

Comments and Questions

Ask at least one **interesting question**¹ on the lecture notes. Also post the question on the Discord channel so that everybody can see and discuss the questions.

2.2 Week 2

Notes

Translating Lean into Math by matching each line in Lean to the corresponding mathematical equation and assumptions. Being able to reverse the proof and translate from Math into Lean is also important. Lean reads from the goal to the axioms, whereas Math is written from the axioms to the goal (usually).

Defining data types recursively (in terms of itself). Syntax varies by language.

Recursion example with the Tower of Hanoi. Breaking logical puzzles into iterative steps, then into recursive steps.

¹It is important to learn to ask *interesting* questions. There is no precise way of defining what is meant by interesting. You can only learn this by doing. An interesting question comes typically in two parts. Part 1 (one or two sentences) sets the scene. Part 2 (one or two sentences) asks the question. A good question strikes the right balance between being specific and technical on the one hand and open ended on the other hand. A question that can be answered with yes/no is not an interesting question.

Question

In class we used recursion to prove an algorithmic solution for the tower of hanoi game. Within the scope of proofs, what are the biggest drawbacks/advantages to using iterative vs recursive techniques?

Homework

Level 1 - Zero Add

In this level we prove the theorem that $0 + n = n$. It was our first use of proof by induction.

induction n with d hd

rw add zero

rfl

rw add succ

rw hd

rfl

Level 2 - Succ Add

In this level we prove the theorem that $\text{succ}(a) + b = \text{succ}(a + b)$.

induction b with d hd

repeat rw add zero

rfl

rw add succ

rw hd

rw add succ

rfl

Level 3 - Add Comm

In this level we prove the theorem that $a + b = b + a$.

induction b

rw add zero

rw zero add

rfl

rw succ add

rw add succ

rw nih

rfl

Level 4 - Add Assoc

In this level we prove the theorem that $(a + b) + c = a + (b + c)$.

```
induction b
rw add zero
rw zero add
rfl
rw add succ
rw succ add
rw nih
rw add comm
rw succ add
rw add succ
rfl
```

Level 5 - Add Comm Right

In this level we prove the theorem that $(a + b) + c = (a + c) + b$. I will show the mathematical assumptions I used when deciding my proof in Lean.

```
rw add assoc a b — (RHS) By associativity rule,  $a + c + b = a + (c + b)$ 
rw add assoc — (LHS) By associativity rule,  $a + b + c = a + (b + c)$ 
rw add comm b c — (LHS) By commutative rule,  $a + (b + c) = a + (c + b)$ 
rfl — Proves the goal  $X = X$ 
```

2.3 Week 3

Notes

This week we talked a lot about context free grammars, especially within the context of parsing mathematical expressions. I really enjoyed learning more about CFGs, especially visualized, because it helped me to understand how CFGs are so important for programming languages. I also couldn't help but think how cool the visualized abstract syntax tree was. It makes me think more about how this sort of approach can be applied to other areas. I had unintentionally started designing my calculator using a similar paradigm, recursively splitting the expression into two small parts until each expression was complete. (I ended up switching approaches, but I thought it was cool anyway).

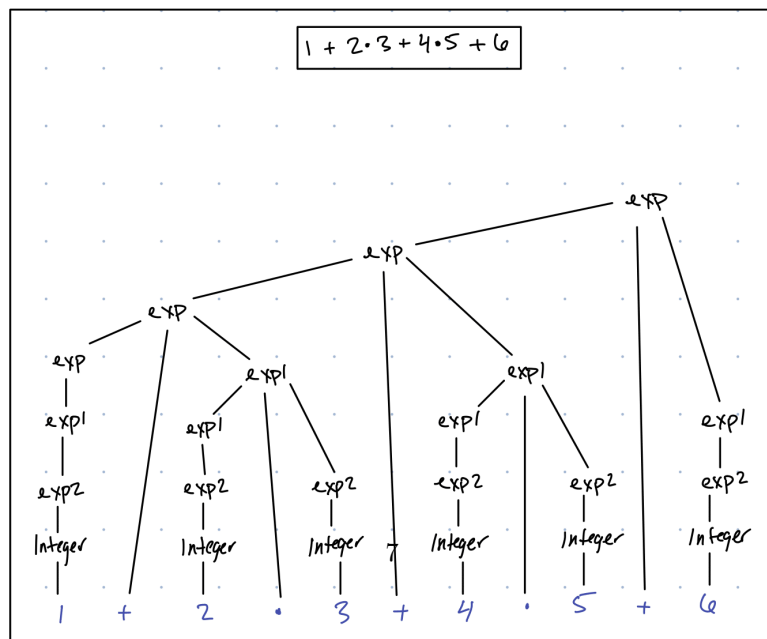
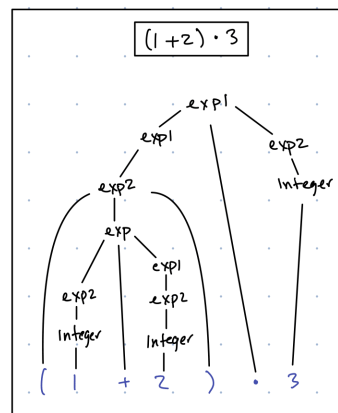
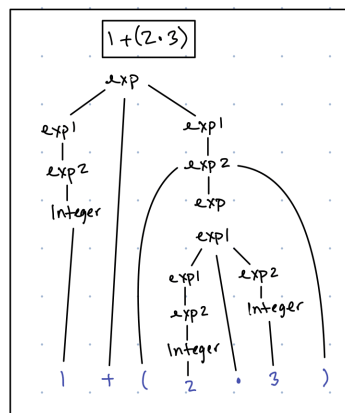
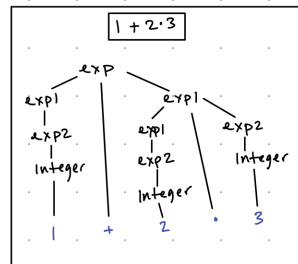
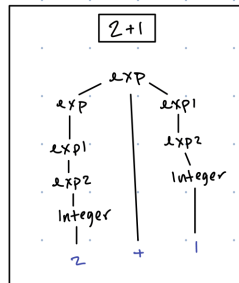
I was doing some additional research into ASTs, and was intrigued by the subsequent processes also required as part of compilation. Semantic analysis, for example. How many other steps are necessary for compilation? Does this process change for interpreted languages? I would be interested in learning more about the overhead costs of script compilation.

Question

Last week we talked about context free grammars, especially within the context of mathematical syntax. Reading online, I learned about context-sensitive grammars. Can a CFG also be described as a CSG, or

vice-versa? From what I could tell online, CSGs are invariably more complex than CFGs, so are there any actual applications of context sensitive grammars, and how is this reflected in its AST?

Homework



2.4 Week 4

Notes

This week we discussed creating a calculator with CFGs and then about the next Lean world on logic. The logic world, similar to the math world, is about breaking down larger problems into manageable parts and tackling them one by one. If we can prove every infinitesimal aspect of a larger problem, then we solve that large problem. I have really enjoyed working in the logic world so far because it is very similar to code design in game development. In game development, there are a number of components that control more context specific components, from the most high-level management context to the most basic mechanic.

Question

Similar to Lianal's question, why does the logic world in Lean allow for varying syntax, while the math Lean world did not? Is this reflected in real (written) math/logic proofs?

Homework

Level 1 - Exactly! It's in the premise

Assumptions: $(P : Prop)(todo_list : P) : P := by$

(1) exact todo_list

Level 2 - And Introduction

Assumptions: $(PS : Prop)(p : P)(s : S) : P \wedge S := by$

(1) exact $\langle p, s \rangle$

Level 3 - The Have Tactic

Assumptions: $(AIOU : Prop)(a : A)(i : I)(o : O)(u : U) : (A \wedge I) \wedge O \wedge U := by$

- (1) have ai := and_intro a i
- (2) have ou := and_intro o u
- (3) exact $\langle ai, ou \rangle$

Level 4 - And Elimination

Assumptions: $(PS : Prop)(vm : P \wedge S) : P := by$

- (1) have p := vm.left
- (2) exact p

Level 5 - And Elimination 2

Assumptions: $(PQ : Prop)(h : P \wedge Q) : Q := by$

- (1) have q := h.right
- (2) exact q

Level 6 - Mix and Match

Assumptions: $(AIOU : Prop)(h1 : A \wedge I)(h2 : O \wedge U) : A \wedge U := by$

(1) exact $\langle h1.left, h2.right \rangle$

Level 7 - More Elimination

Assumptions: $(CL : Prop)(h : (L \wedge (((L \wedge C) \wedge L) \wedge L \wedge L \wedge L)) \wedge (L \wedge L) \wedge L) : C := by$

(1) exact $h.left.right.left.left.right$

Level 8 - Rearranging Boxes

Assumptions: $(ACIOPSU : Prop)(h : ((P \wedge S) \wedge A) \wedge \neg I \wedge (C \wedge O) \wedge U) : A \wedge C \wedge P \wedge S := by$

(1) have $c := h.right.right.left.left$

(2) have $psa := h.left$

(3) have $p := psa.left.left$

(4) have $s := psa.left.right$

(5) have $a := psa.right$

(6) exact $\langle a, c, p, s \rangle$

2.5 Week 5

Notes

This week we discussed lambda higher order functions in the context of Lambda functions and had an interesting discussion about Curry Howard correspondance (currying) and languages. Lambda calculus functions are nameless, type-free functions. Within the context of lean, at least, this has been a difficult concept to grasp, because it is antithetical to the programming philosophy I have learned and practiced. However, the nature of lambda calculus is the same, but instead of relying on type-specific context, it relies on underlying logic within implicit type-deriving and preservation. I think this was confusing to me also because it throws explicit type safety out the window. On Thursday we discussed currying, α -Conversion, β -Reduction, (and more) and had an interesting discussion about languages, and meaning behind language.

Question

How do modern programming languages utilize (or avoid) lambda calculus in their syntax? In C#, `linq` utilizes higher order functions encapsulating (what appears to be) explicitly defined lambda calculus functions, for example `"enum.Where(x \Rightarrow x.a > 10)"`. It also seems like SQL syntax follows currying of lambda calculus functions, for example `"SELECT * FROM _ WHERE _ ORDER BY _"`, despite SQL not being a functional programming language.

Homework

Level 1 - Cake Delivery Service

Assumptions: $(PC : Prop)(p : P)(bakery_service : P \rightarrow C) : C := by$

(1) exact $(bakery_service\ p)$

Level 2 - Identity

Assumptions: $(C : Prop) : C \rightarrow C := by$

(1) exact $\lambda \text{ var } \mapsto \text{var}$

Level 3 - Cake Form Swap

Assumptions: $(IS : Prop) : I \wedge S \rightarrow S \wedge I := by$

(1) exact $\lambda h : I \wedge S \mapsto \text{and_intro } h.\text{right } h.\text{left}$

Level 4 - And Elimination

Assumptions: $(CAS : Prop)(h1 : C \rightarrow A)(h2 : A \rightarrow S) : C \rightarrow S := by$

(1) exact $\lambda c \mapsto h2 (h1 c)$

Level 5 - Riffin Snacks

Assumptions: $(PQRSTU : Prop)(p : P)(h1 : P \rightarrow Q)(h2 : Q \rightarrow R)(h3 : Q \rightarrow T)(h4 : S \rightarrow T)(h5 : T \rightarrow U) : U := by$

(1) exact $h5 (h3 (h1 p))$

Level 6 - and_imp

Assumptions: $(CDS : Prop)(h : C \wedge D \rightarrow S) : C \rightarrow D \rightarrow S := by$

(1) exact $\lambda c d \mapsto h (\text{and_intro } c d)$

Level 7 - and_imp 2

Assumptions: $(CDS : Prop)(h : C \rightarrow D \rightarrow S) : C \wedge D \rightarrow S := by$

(1) exact $\lambda \langle c, d \rangle \mapsto h c d$

Level 8 - Distribute

Assumptions: $(CDS : Prop)(h : (S \rightarrow C) \wedge (S \rightarrow D)) : S \rightarrow C \wedge D := by$

(1) have $\langle l, r \rangle := h$

(2) exact $\lambda s \mapsto \langle l s, r s \rangle$

Level 9 - Uncertain Snacks

Assumptions: $(RS : Prop) : R \rightarrow (S \rightarrow R) \wedge (\neg S \rightarrow R) := by$

(1) exact $\lambda r \mapsto \text{and_intro } (\lambda _ \mapsto r) \lambda _ \mapsto r$

2.6 Week 6

Notes

This week we dove into theory discussions and lectures on application of currying, more complex lambda calculus and church numerals. To be honest, it is hard to remember much notes-wise because the theory was so intense, but I can say with confidence that I was confused - and embraced it! To help with my understanding, I looked for resources online and found this paper by Helmut Brandl titled **Limits of Computability in Lambda Calculus** (<https://hbr.github.io/Lambda-Calculus/computability/text.html>) which actually covered nicely what we have been learning about recently. In the preamble, Kurt Goedel's *Godel Numbering* is discussed, and I thought the paper gave some interesting historical context to a memorable moment in logic and maths history; "In his famous incompleteness theorem (1931) Goedel demonstrated that paradoxical statements can be injected into all formalisms which are powerful enough to express basic arithmetics." Anyway, I digress.

Question

Since lambda calculus is comprised of only abstraction and application, and cannot examine itself, how can infinitely recursing expressions like $(\lambda x.xx)(\lambda x.xx)$ terminate, and is that simply a non-issue? On a computer this would cause a stack overflow, but conceptually it follows the rules. Aside from physical computational constraints, are there other factors that do not permit translation from lambda calculus theory to implementation?

Homework

1) Reduce the following lambda term:

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

(1) $((\backslash m.\backslash n. m n) (\backslash f.\backslash x. f (f x))) (\backslash f2.\backslash x2. f2 (f2 (f2 x2)))$

(2) $(\backslash n. (\backslash f.\backslash x. f (f x)) n) (\backslash f2.\backslash x2. f2 (f2 (f2 x2)))$

(3) $(\backslash f.\backslash x. f (f x)) (\backslash f2.\backslash x2. f2 (f2 (f2 x2)))$

(4) $\backslash x. (\backslash f2.\backslash x2. f2 (f2 (f2 x2))) ((\backslash f2.\backslash x2. f2 (f2 (f2 x2))) x)$

(5) $\backslash x. (\backslash f2.\backslash x2. f2 (f2 (f2 x2))) (\backslash x2. x (x (x x2)))$

(6) $\backslash x.\backslash x2. (\backslash x2. x (x (x x2))) ((\backslash x2. x (x (x x2))) ((\backslash x2. x (x (x x2))) x2))$

(7) $x\backslash.\backslash x2. (\backslash x2. x (x (x x2))) ((\backslash x2. x (x (x x2))) (x (x (x x2))))$

(8) $x\backslash.\backslash x2. (\backslash x2. x (x (x x2))) (x (x (x (x (x (x x2))))))$

(9) $\backslash x.\backslash x2.x (x (x (x (x (x (x (x (x x2))))))))$

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

This lambda expression implements similar behaviour to the identity function, but with church numerals. It returns the application of m on n .

2.7 Week 7

Notes

This week we started working on assignment 3, which is creating an interpreter of lambda calculus. We started with a base interpreter that does not perform as expected in all cases. Uses this as a template, we started working to understand its limitations and to address them. Dr. Kurz gave a brief overview of using the debugger in VScode, which will be a valuable asset during the development of the interpreter.

Question

I recently learned about programming language where the language aspect is built upon the Figma white-board feature. See <https://devpost.com/software/unreal-engjam> (Thanks @JadenJ for sharing!). The language was developed in TypeScript and generates an AST. I was intrigued into what other mediums can serve as effective interfaces; node-based visual scripting (e.g. Unity, UE) is an immediate thought. What other human \rightarrow code interfaces exist? What makes a medium an effective tool for interfacing with code?

Homework

Homework this week is working on assignment 3.

2.8 Week 8

Notes

This week in class we worked on Assignment 3 and talked about strategies for reducing lambda expressions. With the goal of reducing lambda expressions to normal form we learned and practiced using the VSCode debugger to navigate the interpreter program.

Question

Homework

Please see **Homework** under **Week 9** below for the combined homework for weeks 8 and 9.

2.9 Week 9

Notes

This week we worked on assignment 3 in class and talked more about the logic behind reducing lambda expressions.

Question

What separates the possible use cases of call-by-name and call-by-value methodologies? For example, I would think that a type-safe environment would prefer call-by-value because the input variable is evaluated before substitution takes place.

Homework

Question 2 In lambda calculus, expressions are evaluated by applying functions to arguments. For this to happen, functions and arguments need to be grouped in pairs. This is why "a b c d" reduces to "(((a b) c) d)", as it allows for the application of a onto b, c on to the result of that application, and finally d on to the result of the previous applications. "(a)" reduces to "a" since there are no applications happening. Since there is no grouping required, the parentheses are dropped.

Question 3 Capture-avoiding substitution is a method used to replace a variable in a mathematical expression without changing its meaning. Imagine you have a formula with placeholders, and you want to swap one placeholder for another value. If you aren't careful, some parts of the formula might accidentally change meaning because the new value might clash with an existing one. To avoid this, the substitution is done carefully, sometimes renaming other parts, so everything stays clear and correct.

Question 4 No, you do not always get the expected result in lambda calculus. Not all computations reduce to normal form; some, like the omega combinator, can result in infinite loops and never simplify. While many expressions do reduce to a final form, some expressions will continue expanding indefinitely.

Question 5 The smallest lambda expression that does not reduce to normal form is called the omega combinator. It looks like this: $(\lambda x. x x) (\lambda x. x x)$. When you try to reduce it, it infinitely applies itself, leading to an endless loop of expansion. As a result, this expression never reaches a final, simplified form, which is why it does not have a normal form.

Question 7

```
((\m.\n. m n) (\f.\x. f (f x))) (\f.\x. f (f (f x)))  
((\Var1. (\f.\x. f (f x)) Var1) ) (\f.\x. f (f (f x)))  
((\Var2.(\Var4.(Var2 (Var2 Var4)))) (\f.\x.(f (f (f x)))))  
(\Var5.((\f.(\x.(f (f (f x))))) ((\f.(\x.(f (f (f x))))) Var5)))
```

2.10 Week 10

Notes

This week in class we talked about algorithms as rewriting systems, the characteristics these expressions involving confluence, termination, and unique normal forms. We went into detail talking about the buggle sort as an example of an ARS.

Question

In the notes, an ARS is described as a non-deterministic algorithm for determining equivalency within the ARS. What about ARSs makes this process non-deterministic, and how can a non-deterministic process define equivalency?

Homework

2.11 Week 11

Notes

On Tuesday we played around with the sliding puzzle and discussed solvability and invariability. Invariability is a system that doesn't change when transformations are applied to it.

Question

Homework

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

The most challenging aspect was trying to figure out what functionality the program lacked. Because the program is recursive, tracking the steps individually was difficult.

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

We came up with the key insight into solving assignment 3 by focusing less on the the program and more on what, conceptually, needed to happen at each iteration of the evaluation process. This way, we could see that any lam statement in the AST needed to be preceeded by an app statement, and figure out exactly which step was being missed by the program.

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

My most interesting takeaway definitely concerned the debugger. Conceptually learning more abot how an interpreter program works and recurses was exciting, but learning about another tool and getting it under my belt felt very valuable.

2.12 Week 11

Notes

On Tuesday we walked through an example of an ARS that described XOR behaviour. We applied the relationship between the ARS and the XOR truth table to an even/odd table, as well. This helped me to think about ARS's as describing non-determinate systems, because the behaviour that they describe (the equivalency behaviour) can be applied to many contexts, in the same way that a function doesn't care about the context that it is called from, or the context of its parameters. We continued to work on understanding other examples of ARSs.

Question

When thinking about how an ARS could describe a physiological system (e.g. homeostasis), is it possible to construct an ARS that can adapt its ruleset over time? For example, the characteristic of homeostasis, or rather its impact on bodily systems, is different if the person is in very cold weather as opposed to very hot weather.

Homework

1. $A = \{\}$

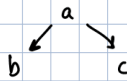
2. $A = \{a\}, R = \{\}$

a

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

$a \looparrowright$

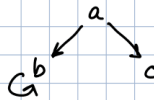
4. $A = \{a, b, c\}, R = \{(a, b), (a, c)\}$



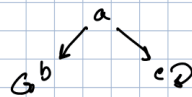
5. $A = \{a, b\}, R = \{(a, a), (a, b)\}$


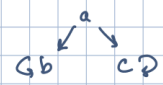


6. $A = \{a, b, c\}, R = \{(a, b), (b, b), (a, c)\}$



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



CONFLUENT	TERMINATING	UNIQUE NORM FORMS	EXAMPLE
TRUE	TRUE	TRUE	$A = \{a, b\}, R = \{(a, b)\}$ $a \rightarrow b$
TRUE	TRUE	FALSE	Not possible
TRUE	FALSE	TRUE	Not possible
TRUE	FALSE	FALSE	$A = \{a, b\}, R = \{(a, b), (b, b)\}$ $a \rightarrow b^2$
FALSE	TRUE	FALSE	$A = \{a, b, c\}, R = \{(a, b), (a, c)\}$ 
FALSE	TRUE	TRUE	Not possible
FALSE	FALSE	TRUE	Not possible
FALSE	FALSE	FALSE	$A = \{a, b, c\}, R = \{(a, b), (a, c), (b, b), (c, c)\}$ 

2.13 Week 12

Notes

This week in class we talked about invariants, specifically within the context of a sliding puzzle on Tuesday. This puzzle was an interesting demonstration of how moving pieces around a puzzle which was impossible to begin with will never make it solvable, and vice versa. This also leads off of the discussion that we had about splitting computations into different equivalence classes, such that the defining characteristics of those computations will always remain regardless of how the system is changed. We discussed fixed-point combinators on Thursday, and how they can be used to implement recursion into lambda calculus. To do this we re-visited conditionals in lambda calculus and defined naming.

Question

Are there other methods of performing recursion in lambda calculus besides fixed-point combinators? Does the definition of naming functional or is it just syntactic sugar?

Homework

$$\begin{aligned}\text{let fact}(3) &\equiv (\text{fix } (f.x \ x)) \text{ in } 3 && \leftarrow \text{def of let rec} \\ &\equiv \beta \text{ f.x } (\text{fix } (f.x \ x)) \text{ in } 3 && \leftarrow \text{beta rule: sub fix F} \\ &\equiv \beta 3 * (\text{fix } (f.x \ x)) \text{ in } 2 && \leftarrow \text{beta rule: sub fix F}\end{aligned}$$

2.14 Week 13

Notes

The linked story in the schedule about Achilles and tortoise was a very fun read, and had some interesting reflections on the termination of systems. If someone must continually accept new premises to justify previous conclusions, then the system will never terminate. In recursion, this endless cycle is handled with a base case, which is compared to at the beginning of each recurse. In the story, it was interesting because as opposed to continually moving toward a solution, Achilles and the tortoise always moved further away from it. In programming, a recursion problem such as fibonacci always moves toward a problem in order to intersect with the base case and therefore terminate. In this way it is interesting that the story described a problem that could not terminate because the tortoise (unreasonably) continually changed the base case.

Question

How transferable are heuristics of puzzle solving in real life (i.e. solving in our heads) vs using a computer to solve. Computers use algorithms to solve puzzles (assuming they are solvable), but are there algorithms that are less efficient when performing them in real life?

Homework

This week we worked on the next milestone for our interpreter.

Exercise 5

(1) Reduce the following strings using the rewrite rules:

- 1. $abba \rightarrow aba \rightarrow aa \rightarrow \epsilon$
- 2. $bababa \rightarrow baabba \rightarrow bbba \rightarrow bba \rightarrow ba \rightarrow a$

(2) The ARS does not terminate because $ab \rightarrow ba$ and $ba \rightarrow ab$ are cyclic.

(3) There are 3 equivalence classes:

- 1. $\epsilon \rightarrow \epsilon$
- 2. Even number of $a's \rightarrow \epsilon$
- 3. Odd number of $a's \rightarrow a$

(4) Yes, either of the following options would work

- 1. $ab \rightarrow ba$ becomes $ab \rightarrow a$
- 2. $ba \rightarrow ab$ becomes $ba \rightarrow a$

These make the ARS terminate without changing the equivalence classes because it does not change the number of a s in the string.

(5) There are 2 NFs for this ARS, being ϵ and a . The termination strategy in (4) produces an invariant in the number of a s in the string.

Exercise 5b

(1) Reduce the following strings using the rewrite rules:

- 1. $abba \rightarrow aba \rightarrow aa \rightarrow a$
- 2. $bababa \rightarrow ababa \rightarrow aaba \rightarrow aaa \rightarrow aa \rightarrow a$

(2) The ARS does not terminate because $ab \rightarrow ba$ and $ab \rightarrow ba$ are cyclic.

(3) There are 2 equivalence classes:

- 1. $\epsilon \rightarrow \epsilon$
- 2. Even number of $S \rightarrow a$ where S is any string.

(4) Because the number of a s does not matter anymore, we can change either or both of the following rules:

- 1. $ab \rightarrow ba$ becomes $ab \rightarrow a$
- 2. $ba \rightarrow ab$ becomes $ba \rightarrow b$

These make the ARS terminate without changing the equivalence classes because it does not change the number of a s in the string.

(5) There are 2 NFs for this ARS, being ϵ and a . The termination strategy in (4) produces an invariant in the number of a s in the string.

3 Lessons from the Assignments

3.1 Assignment 1

This was a very fun project and straightforward project to work on, mainly because I could utilize my coding skills with pure python. Without the overhead of needing to understand and organize the baseline code utilized in later assignments, especially with respect to grammars.

The specifications for this assignment were straightforward but I was still able to apply some conceptual knowledge towards my solution. The code relies on recursing through parentheses-enclosed statements and evaluating a stack of stored terms. I remember we talked about the value of recursion as a problem-solving technique because it encourages a certain way of structuring code (exactly akin to how the evaluate and

substitute functions are structured in assignment 3).

To this end, let's talk a bit more about the high-level specifications of my solution.

- Parsing
 - I utilized regex to parse the input string
 - This separated the string into a sequence of tokens
 - I allowed for integers, floats, parentheses, and operations
- Evaluation
 - For each sub-expression (encapsulated within parentheses), the contents are tokenized and then evaluated
 - One stack contained all values, and another contained all operators
 - If an operator is encountered
 - * While the operator at the top of the stack is higher priority than the encountered operator
 - * Two values are popped from the stack and the operator is applied to them
 - * The calculated value is added to the stack
 - * Finally, the encountered operator is added to the stack
 - This continues until the stacks are empty (when the tokens have all been evaluated)

I learned about applying recursion in a dynamic situation without relying on complex behaviour to capture edge cases. This prepared me to take on Assignment 2.

3.2 Assignment 2

This assignment was our first time utilizing a context free grammar to parse data and evaluate it based on a set of defined contexts. Using a grammar.lark file, we defined a set of rules by which the input would be evaluated. It was interesting to see how type-checking worked via the transformer, and to see that concept reflected in the evaluation.

After we had discussed ASTs in length in class, putting it to use was very cool. I was also very interested in its relation to automata, which I had really enjoyed studying in algorithm analysis. Having an understanding of that concept allowed me to better understand how the context free grammar would be utilized by the transformer.

The specifications of this assignment are not super interesting (compared to Assignment 3), but I will go over them briefly.

- Parsing
 - The transformer contains definitions for each terminal in the CFG.
 - These definitions return a tuple with the correct production of concrete syntax.
- Evaluation
 - Takes in the AST and evaluates it by each terminal syntax.
 - Correctly produces the expected result of each operation.

This assignment taught me about structuring a CFG and, more specifically, about how the transformer interprets terminal and non-terminal transitions. I thought this was very cool, especially considering how you can structure the grammar to recognize specific syntax, or to pass to another variable (which I read being called "syntactic categories" on Wikipedia).

3.3 Assignment 3

Assignment 3 (and continued into assignment 4) is the culmination of the skills and concepts accrued during the class. Our group designed an interpreter to evaluate an AST parsed from a CFG. We constructed the CFG to reflect the required components of the language as well as their nature as operations in concept.

For this assignment, we created a CFG to interpret lambda calculus operations *lam* and *app*. *Lam* is an un-typed, anonymous function with a single parameter that receives one argument. The *app* refers to the application (or injection) of the argument into the function. In our grammar, we recognize lambda expressions as an expression encapsulated in parentheses that demarcated with a backslash ("**").

The AST is evaluated recursively, and handles each operation singularly (meaning one case at a time). This allows for the recursive function to isolate each operation, which is critical given the AST is (of course) a tree. Our program recognizes the application and expects a lambda to follow. It applies the argument to that expression.

To evaluate lambda expressions, our program follows these specifications:

- Parsing (Grammar \rightarrow AST)
 - The CFG defines terminals for *lam*, *app*, and *var*
 - These definitions return a tuple with the correct production of concrete syntax
- Evaluation
 - Recognizes *app* and finds the enclosed *lam* operation within its siblings
 - Evaluates the argument and substitutes (injects) it into the function (following the technique of isolating each operation)

For this assignment, Szymon and I started by constructing the grammar to produce a valid AST. We evaluated the parsed AST ourselves in order to get the best idea of how to implement the recursive evaluate function. This approach was very valuable, because it allowed us to utilize the skills and concepts we practiced in class, and assure that the evaluate function follows that same logic.

3.4 Assignment 4

This assignment is the true culmination of the semester's content and knowledge. Our group implemented an interpreter to handle languages with mathematical and equality-related operations, lambda calculus functions, equality oper, variable naming, if-else-then blocks, recursive lambda functions (following the Y-combinator concept), and handling of multiple programs.

I was truly excited to work on this assignment, specifically because of my experience on Assignment 2, which was a very fun (and, thankfully, simple) initial dive into implementing interpreted languages. I was excited about writing CFGs to define syntax within a language, and how that model can be extended to account for so many operations. Assignment 4 was my opportunity to really work through how CFGs are written to account for an assortment of different operations in a much more complex context.

3.4.1 Milestone 1

The goal of milestone 1 was essentially to take the previous 2 assignments and combine them into one program.

Since we already had all the necessary grammar and evaluation code to accomplish this milestone, the only challenge lay in integrating them together in the grammar, and then reflecting any necessary changes in the `interpreter.py` file.

We noticed that we needed to re-order the syntactic categories. The lower in the grammar a category lies, the higher priority it is (rather, it will be evaluated against first). We made sure that lambda-calculus related operations maintained a higher priority, and that the *app* terminal maintained the highest priority. Mathematical operations were ordered based on PEMDAS. Expressions within parentheses had a higher priority than *app* operations, and negatives were above multiplication/division, which were above addition/subtraction. We implemented the necessary functions into the Lark transformer and then implemented handling of each operation in the `evaluate` function.

We reused the same implementation of *app*, *lam*, *var*, as well as the mathematical operations. Since each operation is isolated due to recursion, debugging was straightforward and Szymon and I were able to successfully run each test case very quickly. The last thing I will mention about milestone 1 is the use of *number*, which was used in Assignment 3 as well. An evaluation on a *number* tuple simply returned its enclosed value as a float.

3.4.2 Milestone 2

The purpose of this milestone is to extend the progress from milestone 1 with new operations, including if-else-then blocks and other equality operations, variable naming, and recursive functions.

I thought that implementing *let* functionality was interesting within the greater context of lambda calculus because it is for anonymous functions. Despite this, *let* allows us to inject an expression into a lambda function without violating that property. I thought this was cool not only because we maintained that anonymous property, but because it took our language to a higher level, closer to modern languages, and separates the name from its implementation (just like how an abstract class separates the implementation of a class from its behaviour).

Implementing recursion was pretty challenging. Szymon and I needed a couple weeks to properly implement it. The biggest difficulty lay in the grammar, and constructing it in a way that best allows the evaluation to manage the if-else-then block while maintaining arguments. In this way, we wanted to maintain that *fixf* should evaluate to $f(\text{fixf})$ when applicable. Our confusion lay in navigating the AST and (re)constructing a valid tree from the result of each recurse, with respect to how *lam* and *app* need to be parsed into the AST, even if there is no lambda calculus in the function itself.

In the end, the logic for handling recursion follows these specifications:

- Evaluate the *let* expression fully, thereby substituting the function into the body
- The expression will always begin with a *fix*
- If the *fix* expression handles a *lam* operation, restructure the tree to begin with an *app* (with the appropriate arguments) and evaluate
- Fully evaluate, extending the tree until the break condition is met
- Recurse through tree until all operations are handled and resolve to the result

Equality operations were easy to implement, simply requiring some expression on either side of a `"=="` or `"!="` variable. This is implemented as the native equality comparison of each fully evaluated operand. If-else-then was implemented similarly. We can expect an equality operation as the first expression within an if clause, and therefore the fully evaluated result of that expression (which will always be 1.0 for True, and 0.0 for False) will dictate if we recurse evaluate into the *then* or *else* expressions.

3.4.3 Milestone 3

The goal of milestone 3 was to implement list and program (sequencing) functionality into our language.

We started by first implementing list functionality. We added the appropriate syntax to our grammar within the second-highest priority syntactic category. For list syntax, we can expect variables, numbers, variable renaming, recursive functions, and encapsulated expressions. Following these expectations, the aforementioned operations are placed in the highest-priority syntactic category. Throughout the grammar, we made sure to always place data type terminals at a higher priority than operation terminals. This made sure that variables would be appropriately interpreted before introducing any operation syntax.

Since we could be sure that any two variables in a list were encapsulated within *cons*, we could layer variables within a list within a chain of *cons*. The implementation of *cons* allowed that, the evaluation of the first variable will always be the head of the list. If the second variable is a *cons*, then there are other variables in the list, which we can continue to recurse on. Otherwise, we have found tail of the list.

Implementing programs was very straightforward. If we encountered ";" in the input, we know we must isolate the implementation of each program. In accomplish this, the grammar separates the interpretation of each program. When evaluating, we can fully evaluate each program, and then return the combination of each program result joined by ";".

4 Conclusion

This class was a joy to take, and especially to take high level concepts and apply them in meaningful ways. Besides a focus on the technical implementation of programming language interpreters, and the underlying concepts behind how languages are constituted, I enjoyed learning about the history of closely-related logic systems, puzzles and exercises that provide substance to concepts, and thinking about the application of lessons onto my personal areas of interest. I think that the content, and specifically the assignments, provided an invaluable opportunity to practice a very interesting form of development. I had never worked with grammars before, which are very cool, but most of all I have never developed a program quite like interpreter.py. This recursive, operation-based approach was very interesting to wrap my head around, but once I had, extending it came more easily.

One concept in particular was very interesting, and applicable to my field of interest (being game development). Invariants are properties or conditions that remain true during the course of a program. I immediately drew comparisons to game design, wherein the idea of directives, objectives, and consistent behaviour are paramount. Throughout the course of gameplay, there are systems that must remain consistent, and asserting an invariant can help us maintain that consistency. AI systems, for example, comprise a set of predefined behaviours. While the state of the system can change, the underlying conditions that define those states always remain the same. Conflicting conditions are commonplace in these systems (e.g. in-combat and in-combat-low-health share at least one condition), and yet the underlying invariant comprises a set of conditions and can manage these overlaps.

References

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