CPSC-354 Report

Zack Klopukh
Chapman University

November 11, 2024

Abstract

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

Contents

1 Introduction

2 Week by Week

2.1 Week 1

Notes

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

Homework

5) Goal: $a + (b + 0) + (c + 0) = a + b + c$
`rw [add_zero b, add_zero c]`
$a + b + c = a + b + c$
rfl
`Here we use add_zero to rewrite and remove the extra zeros so that both sides are equal by reflexivity.`

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

`rw [add_zero b, add_zero c]`
$a + b + c = a + b + c$
rfl A nearly identical solution as number 5.

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

8) Goal: $2 + 2 = 4$
`rw [four_eq_succ_three]`
2 + 2 = succ 3
`rw [three_eq_succ_two]`
2 + 2 = succ (succ 2)
`rw [two_eq_succ_one]`
succ 1 + succ 1 = succ (succ (succ 1))
`rw [one_eq_succ_zero]`
succ (succ 0) + succ (succ 0) = succ (succ (succ (succ 0)))
`rw [add_succ]`
succ (succ (succ 0) + succ 0) = succ (succ (succ (succ 0)))
`rw [add_succ]`
succ (succ (succ (succ 0) + 0)) = succ (succ (succ (succ 0)))
`rw [add_zero]`
succ (succ (succ (succ 0))) = succ (succ (succ (succ 0)))
`rfl`
This is the longest proof in the homework and involves multiple rewrittings to show. The process I took was turning both 2's and the 4 into their forms as successors of 0. From that point I added successors to the left side and by doing so I rewrite the two separate 2's into one large successor. Finally by using add zero, I removed one of the two left terms completely making it a single term on the left and right, both representative of 4.

Comments and Questions

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

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

2.2 Week 2

Homework

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

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

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

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

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

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

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

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

Proof:

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

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

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

Proof:

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

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

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

Proof:

```
induction a with b hd
rw [zero_add]
rw [zero_add]
rw [add_comm]
rfl
rw [add_assoc]
nth_rewrite 1 [add_assoc]
nth_rewrite 2 [add_comm]
rfl
```

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

Comments and Questions

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

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

2.3 Week 3

Homework

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

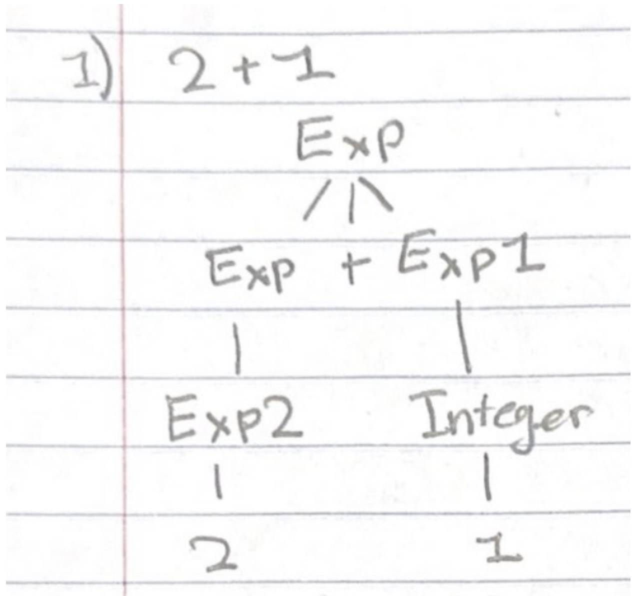
What role does syntax play in the usability of programming languages?
For a comprehensive report detailing my research and findings, please refer to the following link:
https://github.com/zackklopukh/LLMReport/blob/main/README.md

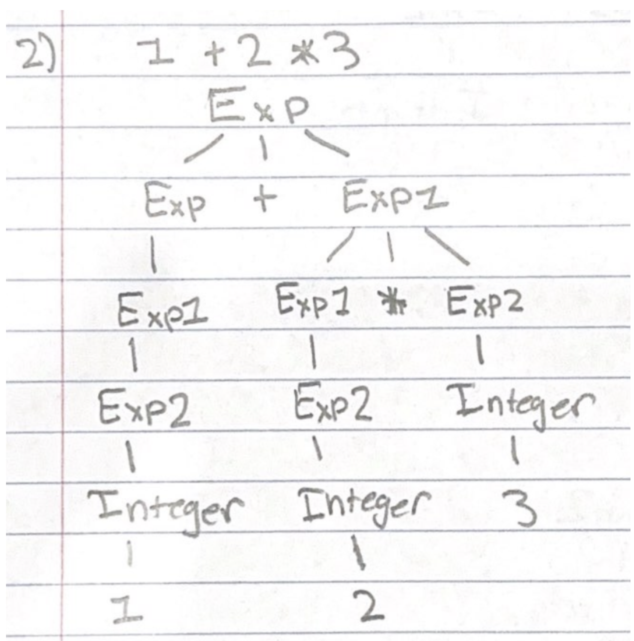
Discord Posting

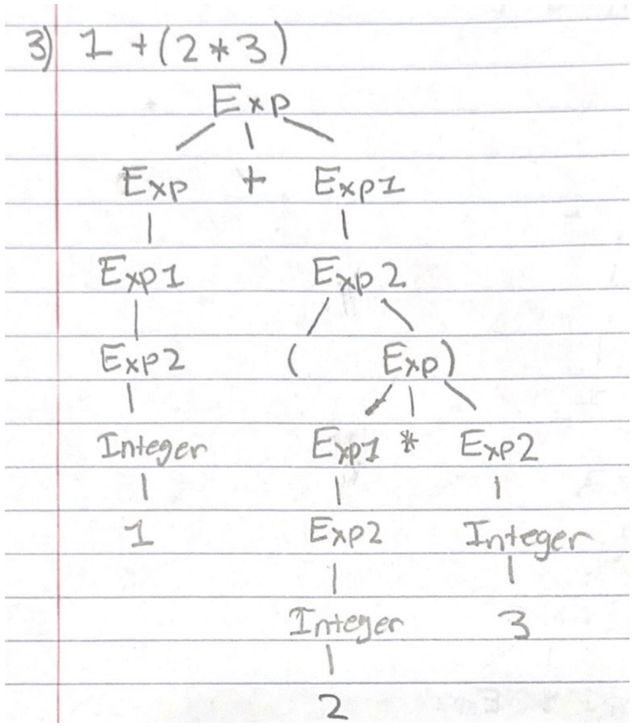
In my literature review with GPT-4o, I examined the profound influence of user-friendly interfaces on programming languages. Initially, programming was confined to those with deep technical knowledge due to complex syntax and low-level languages. However, the evolution of programming syntax to more readable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

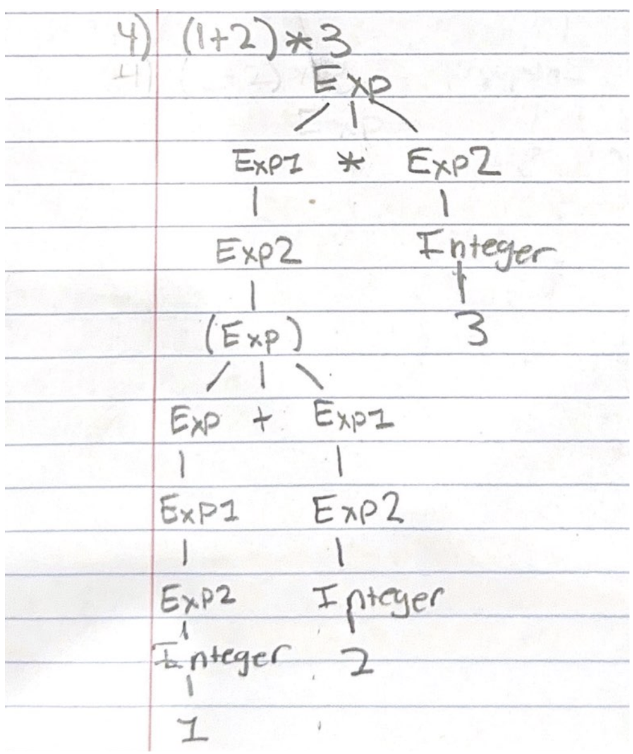
2.4 Week 4

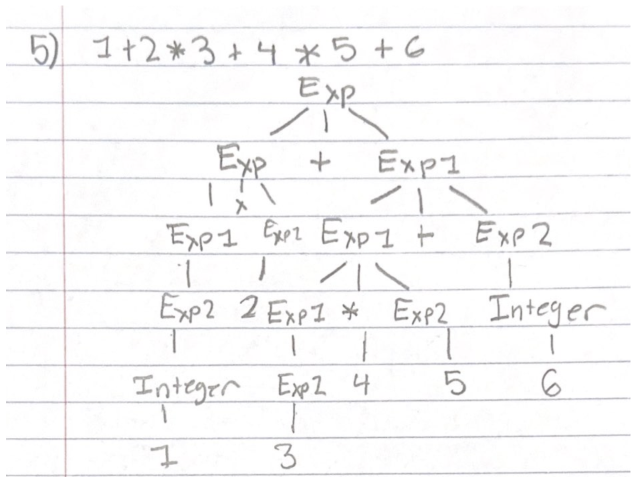
Homework






3) 1 + (2*3)
Exp
Exp + Exp1
Exp1
Exp2
(Exp)
Exp1 * Exp2
Integer
Integer
1
Exp2
Integer
3
Integer
2


4) (1+2)*3
Exp
Exp1 * Exp2
Exp2
Integer
3
(Exp)
Exp + Exp1
Exp1
Exp2
Exp2
Integer
Integer
2
1



Comments and Questions

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

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

2.5 Week 5

Homework

1. Exhibit evidence that you're planning a party.
`exact todo_list`

2. Fill a box, label correctly.
`exact and_intro p s`

3. Three x and_intro.
`exact and_intro (and_intro a i) (and_intro o u)`

4. Exhibit evidence that Pippin is coming to the party.
`exact and_left vm`

5. Both P and Q entails just Q as well!
`exact h.right`

6. and_intro, and_left, and_right.
`exact and_intro (and_left h1) (and_right h2)`

7. Navigate the tree.
`exact h.left.left.right`

8. Take apart and build evidence.
`exact (h.left.right, h.right.right.left.left, h.left.left.left, h.left.left.right)`

Comments and Questions

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

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

2.6 Week 6

Homework

1. Exhibit evidence that cake will be delivered to the party.
`exact bakery_service p`

2. Cake is Cake.
`exact` $\lambda h : C \to h$

3. `Show that  is commutative.`
`exact` $\lambda h : I \wedge S \to$ `and_intro`$(h.right, h.left)$

4. `Show that` $\to$ `is transitive.`
`exact` $\lambda h : c \to h2(h1h)$

5. `Riffin is bringing a unicorn snack.`
`exact h5 (h1 p)`

6. `Conjunction interacting with implication.`
`exact` $\lambda c : C \to \lambda d : D \to h(c, d)$

7. `Conjunction interacting with interaction.`
`exact` $\lambda c\_and\_d : C \wedge D \to h\, c\_and\_d.left\, c\_and\_d.right$

8. $\to$ `distributes over` $\wedge$.
`exact` $\lambda s : S \to (h.lefts, h.rights)$

9. `Write the necessary nested function(s)!`
`exact` $\lambda r : R\,(\lambda s : S \to r, \lambda ns : -S \to r)$

Comments and Questions

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

2.7 Week 8-10

Notes

`In recursive evaluations, such as those found in lambda calculus interpreters, it is crucial to understand the recursive structure and how the evaluation proceeds through function calls and returns. When evaluating lambda expressions, each function call may invoke further recursive calls to process sub-expressions, such as applications or lambda abstractions, and these calls deepen the recursion.`

`As the interpreter encounters applications, it needs to evaluate both the function and the argument, which leads to deeper recursive invocations of the evaluate() function. This recursive descent continues until a terminal case, such as a variable or fully reduced lambda expression,` is reached. At this point, the interpreter begins to return from these calls, completing substitutions or reductions in reverse order of the calls. This is why the trace of recursive calls often shows numbers going up as the recursion deepens and going down as the interpreter returns to earlier points in the code.

Understanding this pattern is key to grasping the operational semantics of lambda calculus interpreters and similar recursive evaluation strategies in programming languages.

Homework

Question 2

In this question, we simplify the expression $a\,b\,c\,d$ step-by-step and show how the left-associative function application works in lambda calculus.

This is interpreted as:

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

Explanation:

- Left-Associative Application: Function application in lambda calculus is left-associative, meaning it groups from the left.
- Step-by-Step Grouping:

  1. Apply a to b: $(a\,b)$
  2. Then, apply the result to c: $((a\,b)\,c)$
  3. Finally, apply that result to d: $(((a\,b)\,c)\,d)$

Therefore, the original expression simplifies to:

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

Question 3: Capture-Avoiding Substitution

Capture-avoiding substitution is a technique in lambda calculus used to replace variables in expressions while ensuring that no variable names are inadvertently "captured" during the substitution process.

Explanation: The potential issue arises when a bound variable in a lambda expression shares the same name as a free variable in the term being substituted. To avoid this, a process called "renaming" or "alpha-conversion" is applied, where bound variables are renamed to fresh variables before substitution, preventing clashes. This ensures that the substitution respects the scope of each variable.

Question 4: Do all computations reduce to normal form?

Not all lambda calculus expressions reduce to normal form. While some expressions, such as those representing arithmetic operations or simple functions, will eventually reduce to a normal form (a fully evaluated state), others may not.

Explanation: Expressions that do not reduce to normal form often arise due to infinite loops or recursive definitions. A classic example is the "omega" combinator, which leads to infinite self-application without ever reaching a normal form.

Question 5: The Smallest Lambda Expression That Does Not Reduce to Normal Form

The smallest lambda expression that does not reduce to normal form is the omega combinator. This expression essentially calls itself in an infinite loop, continuously reapplying the function to itself without any base case to terminate the recursion.

Explanation: This minimal working example (MWE) illustrates the issue of non-termination, where the lambda expression keeps expanding rather than simplifying. The omega combinator is a foundational concept in understanding recursion and non-termination within the theoretical framework of lambda calculus.

Question 7: Evaluating $((\lambda m.\lambda n.m\, n)\,(\lambda f.\lambda x.f\,(f\,x)))\,(\lambda f.\lambda x.f\,(f\,(f\,x)))$

1. Apply the first function $\lambda m.\lambda n.m\, n$ to $\lambda f.\lambda x.f\,(f\,x)$:

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

2. Substitute $n = \lambda f.\lambda x.f\,(f\,(f\,x))$:

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

3. After substitutions and simplifications, the final result is:

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

Question 8: Tracing Recursive Calls to `evaluate()` with $((\lambda m.\lambda n.m\, n)\,(\lambda f.\lambda x.f\,(f\,x)))\,(\lambda f.\lambda x.f\,x)$

Below is the trace, with line numbers and indentation representing the recursion depth:

```
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)))))
        39: eval (\m.(\n.(m n)))
        39: eval (\f.(\x.(f (f x))))
    53: substitute (\n.(m n)) [m -> \f.\x.(f (f x))]
    12: eval (\n.(\f.\x.(f (f x)) n))
```

Comments and Questions

How does the use of functional representations in lambda calculus, such as Church numerals for encoding natural numbers, influence our understanding of data abstraction and manipulation in modern programming languages?

2.8 Week 11

Homework

Question:

For each of the possible combinations of the properties confluent (C), terminating (T), and having unique normal forms (UNF), determine whether such an ARS exists. Provide an example if possible, or explain why it cannot exist. For the examples, draw the ARS diagrams.

Confluent (C)	Terminating (T)	Unique Normal Forms (UNF)	Example
True	True	True	Yes
True	True	False	No
True	False	True	Yes
True	False	False	Yes
False	True	True	No
False	True	False	Yes
False	False	True	No
False	False	False	Yes

Answer:

1. Confluent: True, Terminating: True, Unique Normal Forms: True

Example:

Consider the ARS with elements $S = \{a, b, c\}$ and reduction rules:

$$a \to b, \quad b \to c$$

Diagram:

$$a \longrightarrow b \longrightarrow c$$

Analysis:

- Terminating: Yes. All reduction sequences eventually reach c, which is a normal form. - Confluent: Yes. There are no diverging reduction paths. - Unique Normal Forms: Yes. Every element reduces to c, the unique normal form.

---

2. Confluent: True, Terminating: True, Unique Normal Forms: False

Explanation:

This combination is impossible. In a confluent and terminating ARS, every element reduces to a unique normal form. Therefore, unique normal forms must exist.

---

3. Confluent: True, Terminating: False, Unique Normal Forms: True

Example:

Consider the ARS with elements $S = \{a, b, c\}$ and rules:

$$a \to b, \quad b \to a, \quad a \to c, \quad b \to c$$

Diagram:

$$\begin{tikzcd} a \arrow[r, leftrightarrow] \arrow[d] & b \arrow[dl] \\ c & \end{tikzcd}$$

Analysis:

- Terminating: No. There is an infinite loop between a and b: $a \to b \to a \to b \to \ldots$ - Confluent: Yes. Both a and b reduce to c, and any divergent paths converge at c. - Unique Normal Forms: Yes. All elements reduce to c, the unique normal form.

---

4. Confluent: True, Terminating: False, Unique Normal Forms: False

Example:

Consider the ARS with elements $S = \{a, b\}$ and rules:

$$a \to b, \quad b \to a$$

Diagram:

$$a \longleftrightarrow b$$

Analysis:

- Terminating: No. There is an infinite loop between a and b. - Confluent: Yes. There are no diverging paths; the reductions cycle between a and b. - Unique Normal Forms: No. Neither a nor b is a normal form (both can reduce further), and there are no normal forms in S.

---

5. Confluent: False, Terminating: True, Unique Normal Forms: True

Explanation:

This combination is impossible. If an ARS has unique normal forms, it must be confluent.

---

6. Confluent: False, Terminating: True, Unique Normal Forms: False

Example:

Consider the ARS with elements $S = \{a, b, c\}$ and rules:

$$a \to b, \quad a \to c$$

Diagram:

$$\begin{tikzcd} & b \\ a \arrow[ur] \arrow[dr] & \\ & c \end{tikzcd}$$

Analysis:

- Terminating: Yes. Reductions terminate at b or c, which are normal forms. - Confluent: No. From a, we can reach two different normal forms, b and c, which are not joinable. - Unique Normal Forms: No. The element a has two distinct normal forms.

---

7. Confluent: False, Terminating: False, Unique Normal Forms: True

Explanation:

This combination is impossible. Unique normal forms imply that the ARS is confluent and normalising.

---

8. Confluent: False, Terminating: False, Unique Normal Forms: False

Example:

Consider the ARS with elements $S = \{a, b, c\}$ and rules:

$$a \to b, \quad b \to a, \quad a \to c$$

Diagram:

$$\begin{tikzcd} a \arrow[rr, leftrightarrow] \arrow[dr] & & b \\ & c & \end{tikzcd}$$

Analysis:

- Terminating: No. There is an infinite loop between a and b. - Confluent: No. From a, we can either loop indefinitely between a and b, or reduce to c. - Unique Normal Forms: No. c is a normal form reachable from a, but b and a do not reduce to c if we stay in the loop.

2.9 ...

...

3 Lessons from the Assignments

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

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

4 Conclusion

Soon to be.

References

[BLA] Author, Title, Publisher, Year.