CPSC-354 Report

Zack Klopukh
Chapman University

November 14, 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 $\mathrm{succ}(a) + b = \mathrm{succ}(a + b)$.
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 $\mathrm{succ}(a)$ to b results in $\mathrm{succ}(a + b)$. 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.

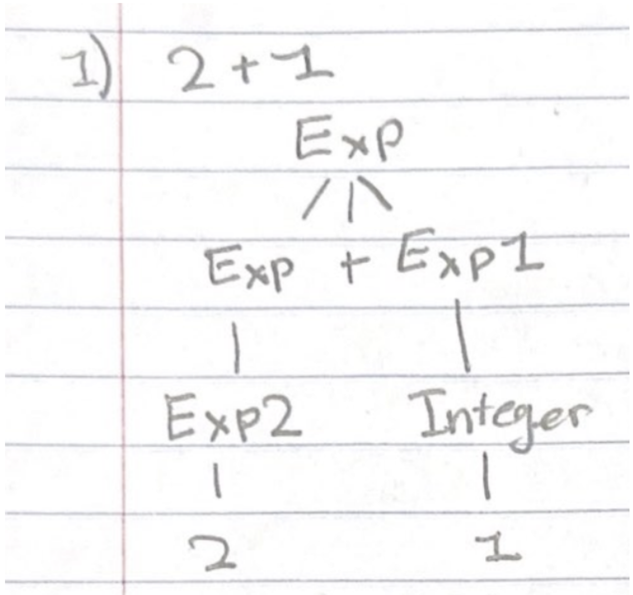
What role does syntax play in the usability of programming languages?
For a comprehensive report detailing my research and findings, please refer to the following link:
https://github.com/zackklopukh/LLMReport/blob/main/README.md

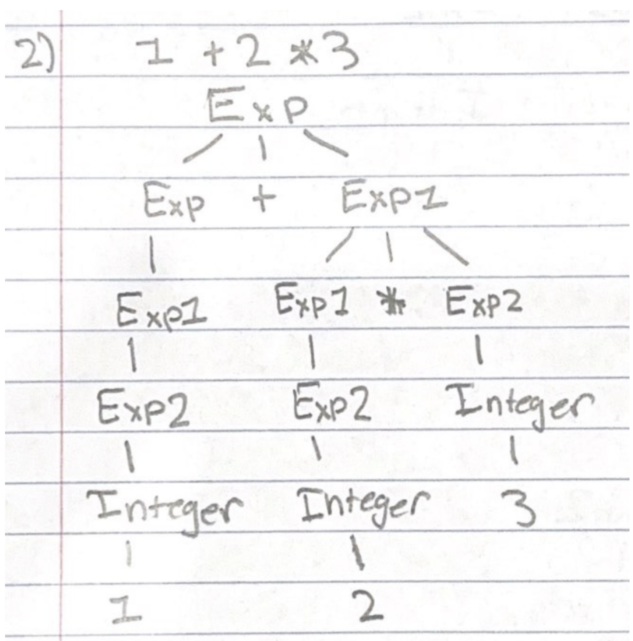
Discord Posting

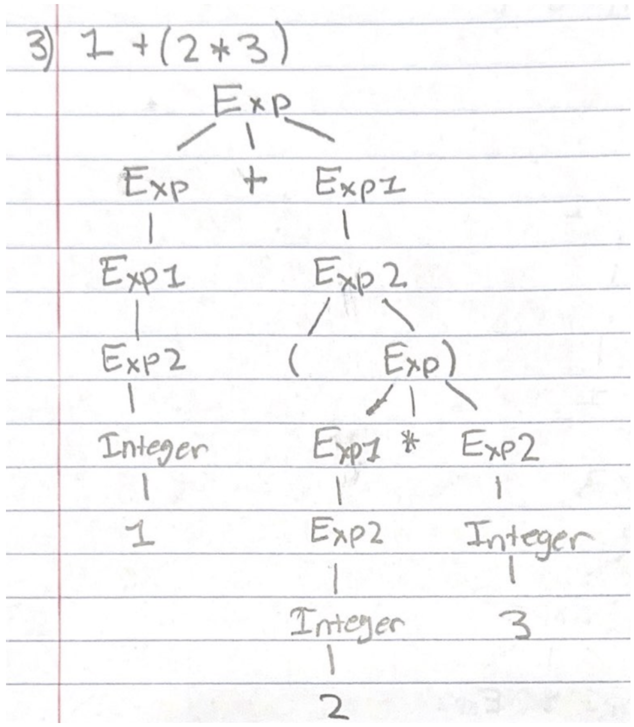
In my literature review with GPT-4o, I examined the profound influence of user-friendly interfaces on programming languages. Initially, programming was confined to those with deep technical knowledge due to complex syntax and low-level languages. However, the evolution of programming syntax to more readable 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

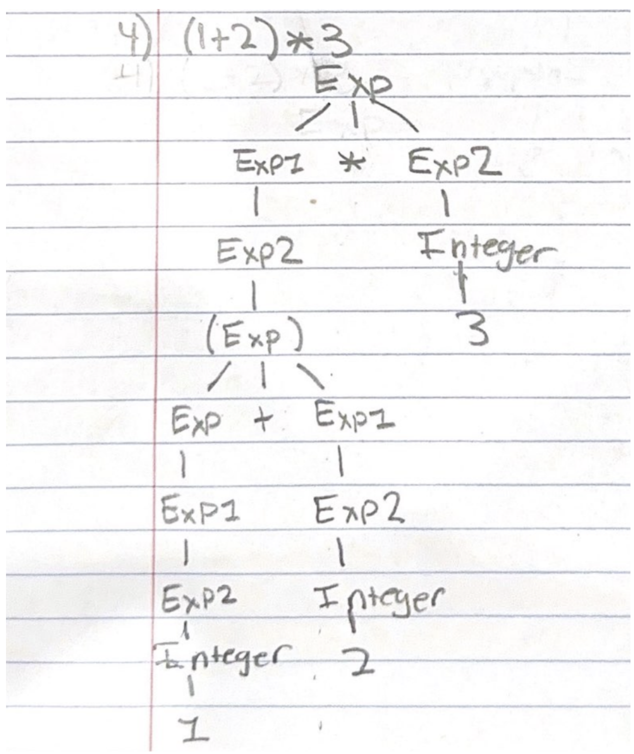
2.4 Week 4

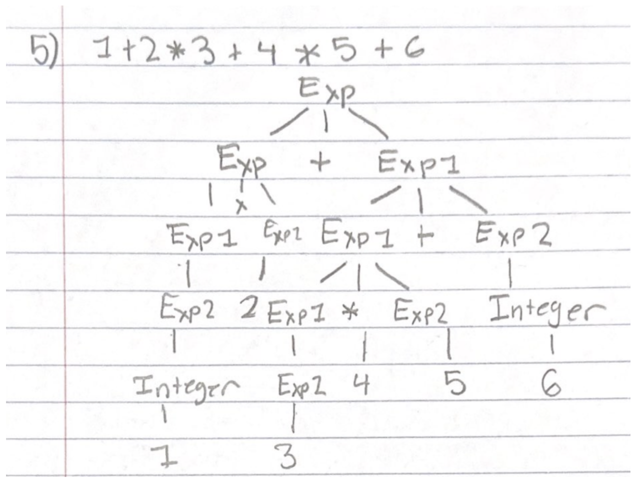
Homework






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


4) (1+2)*3
Exp
Exp1
*
Exp2
Exp2
Integer
(Exp)
3
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, like those in lambda calculus interpreters, it's essential to grasp the recursive structure and how evaluation progresses through function calls and returns. When evaluating lambda expressions, each function call can trigger additional recursive calls to handle sub-expressions, such as applications or lambda abstractions, leading to deeper levels of recursion.`

`As the interpreter processes applications, it must evaluate both the function and its argument, further invoking the 'evaluate()' function recursively. This descent continues until it reaches a terminal case, such as a variable or a fully reduced lambda expression. At that point, the` interpreter starts returning from these calls, completing substitutions or reductions in the reverse order of invocation.

This recursive pattern explains why the trace of calls typically shows increasing depth as recursion progresses and decreasing depth as the interpreter unwinds. Understanding this behavior is fundamental to comprehending the operational semantics of lambda calculus interpreters and similar recursive evaluation strategies in programming languages.

Homework

Question 2

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 method in lambda calculus for replacing variables in expressions while ensuring that no variable names are unintentionally "captured" by the scopes of bound variables. This technique preserves the intended variable bindings and avoids scope conflicts during substitution.

Explanation: A problem can occur when a bound variable in a lambda expression has the same name as a free variable in the term being substituted. To prevent this, a technique called "renaming" or "alpha-conversion" is used. This involves renaming bound variables to fresh, unique names before performing the substitution, ensuring that variable scopes remain distinct and no conflicts arise.

Question 4: Do all computations reduce to normal form?

Not all lambda calculus expressions reduce to a normal form. While certain expressions, like those representing arithmetic operations or straightforward functions, eventually simplify to a fully evaluated state, others may result in non-termination, endlessly expanding without reaching a normal form.

Explanation: Expressions that fail to reduce to a normal form typically result from infinite loops or recursive definitions. A well-known example is the omega combinator, which undergoes infinite self-application, never achieving a normal form.

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

The omega combinator is the simplest lambda expression that does not reduce to a normal form. It effectively creates an infinite loop by repeatedly applying the function to itself, lacking a base case to halt the recursion.

Explanation: This minimal working example (MWE) demonstrates the problem of non-termination, where a lambda expression continues to expand indefinitely instead of simplifying. The omega combinator plays a key role 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, Terminating, and Unique Normal Forms: True

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 | True, Reduction sequences eventually reach c. Confluent | True, No diverging reduction paths. Unique Normal Forms | True, elements reduces to c, the unique normal form.

2. Confluent and True, Unique Normal Forms False

This is impossible. In a confluent and terminating ARS, every element reduces to a unique normal form. Unique normal forms will be true.

3. Confluent and Unique Normal Forms True | Terminating False

Example:

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 | False, Infinite loop between a and b: $a \to b \to a \to b \to \ldots$ Confluent | True. a and b reduce to c, and any divergent paths converge. Unique Normal Forms | True, All elements reduce to c, the unique normal form.

4. Confluent True, Terminating and Unique Normal Forms False

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

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

Diagram:

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

Analysis:

Terminating | False, Infinite loop between a and b. Confluent | True, No diverging paths. Reductions cycle between a and b. Unique Normal Forms | False, Neither a nor b is a normal form.

5. Confluent False, Terminating and Unique Normal Forms True

Explanation:

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

---

6. Confluent and Unique Normal Forms False, Terminating True

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

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

Diagram:

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

Terminating | True, Reductions terminate at normal forms. Confluent | False, From a, can reach two different normal forms, b and c, which aren't joinable. Unique Normal Forms | false because The element a has two distinct normal forms.

7. Confluent and Terminating are False, Unique Normal Forms, True

Reasoning: Unique normal forms imply that the ARS is confluent and normalising.

8. Confluent, Terminating, and Unique Normal Forms are 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:

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

Terminating: No, infinite loop between a and b. Confluent: No. either loop indefinitely between a and b, or reduce to c. Unique Normal Forms: No, 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.