



**Learning Support  
for Writing Proofs  
in Coq**

**Jeremy Yew Ern**

**Capstone Final Report for BSc (Honours) in  
Mathematical, Computational and Statistical Sciences**

**Supervised by: Professor Olivier Danvy**

**AY 2019/2020**

# **Yale-NUS College Capstone Project**

## **DECLARATION & CONSENT**

1. I declare that the product of this Project, the Thesis, is the end result of my own work and that due acknowledgement has been given in the bibliography and references to ALL sources be they printed, electronic, or personal, in accordance with the academic regulations of Yale-NUS College.
2. I acknowledge that the Thesis is subject to the policies relating to Yale-NUS College Intellectual Property ([Yale-NUS HR 039](#)).

## **ACCESS LEVEL**

3. I agree, in consultation with my supervisor(s), that the Thesis be given the access level specified below: [check one only]

☐ **Unrestricted access**

Make the Thesis immediately available for worldwide access.

☐ **Access restricted to Yale-NUS College for a limited period**

Make the Thesis immediately available for Yale-NUS College access only from \_\_\_\_\_ (mm/yyyy) to \_\_\_\_\_ (mm/yyyy), up to a maximum of 2 years for the following reason(s): (please specify; attach a separate sheet if necessary):

\_\_\_\_\_.

After this period, the Thesis will be made available for worldwide access.

☐ **Other restrictions: (please specify if any part of your thesis should be restricted)**

\_\_\_\_\_  
\_\_\_\_\_

\_\_\_\_\_  
Name & Residential College of Student

\_\_\_\_\_  
Signature of Student

\_\_\_\_\_  
Date

\_\_\_\_\_  
Name & Signature of Supervisor

\_\_\_\_\_  
Date

## *Acknowledgements*

I would like to acknowledge Prof Danvy for his generous and dedicated guidance.

YALE-NUS COLLEGE

# *Abstract*

B.Sc (Hons)

## **Learning Support for Writing Proofs in Coq**

by Jeremy YEW

This project provides learning support for students enrolled in YSC3216: Functional Programming and Proving (FPP), by providing proof-reader: an Emacs-integrated tool that acts a set of 'safety rails', guiding students as they build muscle memory for good proving and programming habits. proof-reader analyzes student submissions and generates corrective suggestions, thus automating intervention on syntax issues, which is usually performed via written feedback by the course instructor. This allows the instructor to focus on substantive rather than superficial feedback. The tool is intended to be used interactively by students, both during proof editing and before submission.

# Contents

<b>Acknowledgements</b>	<b>ii</b>
<b>Abstract</b>	<b>iii</b>
<b>1 Context</b>	<b>1</b>
1.1 Introduction . . . . .	1
1.2 Functional programming (FP) . . . . .	2
1.2.1 Proving . . . . .	2
1.2.2 The Coq proof assistant . . . . .	3
1.2.3 YSC3236 Functional Programming and Proving (FPP) . . . . .	3
1.2.4 The GNU Emacs Editor . . . . .	4
1.2.5 Proof General . . . . .	4
<b>2 Motivation</b>	<b>6</b>
2.1 Building muscle memory . . . . .	6
2.2 Syntax issues . . . . .	7
2.2.1 Abuse of tactics . . . . .	7
2.2.2 Misuse of tactics . . . . .	8
2.3 The goal: automated intervention on syntax issues . . . . .	10
<b>3 Solution: The proof-reader tool</b>	<b>11</b>
3.1 Parsing student submissions with proof-reader . . . . .	11

3.2	Usage . . . . .	12
3.3	Setup . . . . .	12
3.4	Examples . . . . .	13
3.4.1	Example 1: Warning user of instances where unpermitted tactics are used . . . . .	13
3.4.2	Example 2: Warning user of instances of incorrect arity . . . . .	13
3.5	Possible errors . . . . .	14
<b>4</b>	<b>Design and Implementation</b>	<b>15</b>
4.0.1	Backus-Naur Form (BNF) . . . . .	15
4.0.2	The grammar module: a BNF-inspired data structure	16
4.0.3	Parsing input to generate a syntax tree . . . . .	18
4.0.3.1	Constructing syntax tree nodes . . . . .	18
4.0.3.2	Constructing terms and subterms . . . . .	21
4.0.3.3	Example syntax tree . . . . .	23
4.0.4	Feature 1: Recognizing unpermitted tactics . . . . .	26
4.0.5	Feature 2: Verifying arity . . . . .	27
4.0.5.1	Collecting arity signatures of built-in library theorems . . . . .	27
4.0.5.2	Collecting arity signatures of the syntax tree	28
4.0.5.3	Checking arity of the syntax tree . . . . .	29
4.0.6	User interface . . . . .	30
4.0.7	Extending proof-reader . . . . .	30
4.0.8	Unit and acceptance testing . . . . .	30
<b>5</b>	<b>Discussion</b>	<b>32</b>

5.1	Time and space complexity . . . . .	32
5.1.1	Regular expression matching in <code>construct_node</code> . .	32
5.1.2	Constructing and traversing the syntax tree . . . . .	33
5.1.3	Overall complexity . . . . .	33
5.2	Limitations of the grammar module . . . . .	34
5.3	Alternative approaches . . . . .	35
5.3.1	Modifying Coqtop source code . . . . .	35
5.3.2	Using a parser generator . . . . .	36
5.4	Future work . . . . .	37
5.5	Reflections . . . . .	38
	<b>Bibliography</b>	<b>39</b>

# Chapter 1

## Context

### 1.1 Introduction

The goal of this project is to provide learning support for students enrolled in YSC3216: Functional Programming and Proving (FPP), by providing a tool that generates corrective suggestions for syntax issues, to be used by students to check their code.

FPP is a course in Yale-NUS College under the Mathematical, Computational and Statistical Sciences major, most recently offered in AY19/20 Semester 1 and taught by Professor Olivier Danvy. FPP introduces students to the Coq proof assistant, which is a system for writing and verifying formal proofs. Throughout the course, students learn that precision and orderliness in their code both reflects and encourages clarity of thought. Therefore, one of the primary learning goals for the first half of the course is to build muscle memory for basic proof techniques and programming habits.

To this end, I implement a program called the proof-reader that acts a set of 'safety rails' to guide students towards good proving and programming habits via automated intervention on syntax issues. In particular,



this tool will help enforce prescribed techniques: using only tactics introduced in the course, and applying tactics explicitly. Since these issues are often highlighted in written feedback from the instructor, the tool also supports students' learning by greatly reducing the feedback cycle and allowing the instructor to focus on substantive rather than superficial feedback.

The program relies on a grammar specification of a subset of Coq as an input to the tool, which the instructor may modify as the course evolves. The program's interface is a simple Emacs command, and is intended to be used interactively by students, both during proof editing and before submission.

## 1.2 Functional programming (FP)

Functional programming is a programming paradigm that models programs as mathematical functions. Students taking FPP are expected to have completed the Introduction to Computer Science module, which will have trained them in functional programming (amongst other concepts) using the language OCaml. Coq has a language of programs that is very similar to OCaml, and is in fact written in OCaml.

### 1.2.1 Proving

In mathematics, a proposition is a statement that either holds or does not hold; a proposition is also sometimes called a theorem or lemma. Proofs may be defined as a logical argument about whether a proposition holds. Proofs use logical rules to demonstrate that what we know or assume to

be true – an axiom – implies the truth of something that we do not know - a proposition.

Propositions often contain equations, which are statements asserting the equality of two expressions containing variables. When writing proofs (including proofs in Coq), we exercise equational reasoning: we apply axioms to equations in order to incrementally transform them into something that is clearly true.

### 1.2.2 The Coq proof assistant

Many proofs in mathematics or computer science are natural language proofs - that is, they are written in a natural language, like English. Since natural languages are often ambiguous, natural language proofs are susceptible to misinterpretation or misconception.

On the other hand, just as there programming languages that express a set of instructions to be executed by a computer, there are also domain-specific languages for writing formal proofs that can be automatically, or mechanically, verified by a computer. Coq allows us to write formal, verifiable proofs in a structured logical language called Gallina, and will also automatically verify that our proofs are syntactically correct as well as type-correct. See [Coq homepage](#).

### 1.2.3 YSC3236 Functional Programming and Proving (FPP)

FPP is taken not only by Yale-NUS students, but also PhD and post-doctoral students from the National University of Singapore (NUS) School of Computing (SoC). Through the course, students gain an appreciation for the interconnectedness of computer programs and logical proofs -

which have previously been presented to them as distinct domains of knowledge. For example, they are led to realize that a Coq proof exactly corresponds to an equivalent mathematical proof they have written in detail, by hand.

Students engage in weekly assignments consisting of rigorous, progressive exercises involving:

- writing mathematical proofs
- writing programs, and proofs about the properties of programs
- eventually, stating their own theorems and proving them.

By the end of the course, students will have independently written more proofs than they have ever written in their lives, all of which would have been formally verified.

### 1.2.4 The GNU Emacs Editor

Emacs ([GNU Emacs homepage](#)) is a family of real-time text editors characterized by their customizability and extensibility. GNU Emacs was written in 1984 by GNU Project founder Richard Stallman. At Yale-NUS College, GNU Emacs is used in Intro CS, Intro to Algos and Data Structures, and FPP. GNU Emacs provides a language called Emacs Lisp ([Emacs Lisp homepage](#)) that is used to write programs run within Emacs. The proof-reader tool uses Emacs Lisp to provide a user interface.

### 1.2.5 Proof General

Proof General ([Proof General homepage](#)) is an Emacs interface for proof assistants, developed at the University of Edinburgh since 1992. It provides

---

a common interface across various proof assistants, including Coq, and allows users to interactively edit proof scripts. The proof-reader tool interacts with Proof General functions in order to provide its functionality.

## Chapter 2

# Motivation

In this section, we relate some of the learning goals of FPP to a specific problem faced in class, in order to motivate the solution presented. **This paper's contributions therefore rely on the pedagogical approach taken by Professor Danvy in FPP.** Readers may refer to "Mystery Functions" (Danvy, 2020) – presented as a keynote at the International Symposium on Implementation and Application of Functional Languages 2019 – which conveys the spirit of this approach, its assumptions, and its results.

### 2.1 Building muscle memory

The learning philosophy of FPP is that programming and proving is similar to training in any skilled discipline such as martial arts, cooking, or dance: beginner training should build muscle memory for basic skills and habits. For example, if you are training to be a chef, but you don't develop proper knife skills early on, this will hurt you for the rest of your career.

Therefore, in the first half of the course, students complete rigorous, progressive exercises in order to practice specific proof techniques and

programming habits. In the second half of the course, students can then rely on this muscle memory to write proofs with greater creativity and efficiency.

## 2.2 Syntax issues

Just as there are many ways to write the same program, there are many equivalent versions of a Coq proof, especially because Coq is flexible and allows you to take shortcuts. However, for new learners, this power can be counterproductive. In the context of FPP, several issues arise.

### 2.2.1 Abuse of tactics

First, students may abuse tactics that have not been introduced in the course. When students get stuck on a proof, they might Google for related solutions or search the Coq documentation for anything that will solve the proof. They might end up using a ‘magical’ tactic, for example the tactic ‘trivial’, as in the example proof below.

```
Lemma SSSn_is_3_plus_n :  
  forall n : nat,  
    S (S (S n)) = 3 + n.  
Proof.  
  trivial.  
Qed.
```

Under the hood, the ‘trivial’ tactic iterates over various strategies to solve the current formula. However, in the first half of the course the focus is for students to understand every single proof step they write. Therefore,

using a tactic like ‘trivial’ goes against the objective of the exercise. Instead, the proof should demonstrate every step:

```
...
Proof.
  intro n.
  rewrite <- (Nat.add_1_1 n).
  rewrite <- (plus_Sn_m 1 n).
  rewrite <- (plus_Sn_m 2 n).
  reflexivity.
Qed.
```

Yet these tactics still appear in student submissions. This causes time between resubmissions to be wasted on superficial feedback.

### 2.2.2 Misuse of tactics

Second, even when students use tactics that have been introduced, they may misuse them. For instance, the `rewrite` tactic is used to apply a rewrite rule to the current goal. A rewrite rule is a function that expects arguments that refer to corresponding terms in the goal; Coq will rewrite the corresponding terms in the goal. For example, the rewrite rule `Nat.add_assoc` accepts three arguments, `n`, `m`, and `p`:

```
Check Nat.add_assoc.
# Nat.add_assoc : forall n m p : nat, n + (m + p) = n + m + p.
```

However, Coq is flexible with the number of arguments given to terms. As the example proof below demonstrates, you could give the rewrite rule three, two, one or zero of the rewrite arguments required, and Coq will

simply infer the intended application, by picking the first terms in the formula that it can apply the rule to.

```

Proposition add_assoc_nested :
  forall a b c d e: nat,
    a + b + c + d + e = a + (b + (c + (d + e))).
Proof.
  intros a b c d e.
  rewrite -> (Nat.add_assoc a b).
  rewrite -> (Nat.add_assoc (a + b)).
  rewrite -> Nat.add_assoc.
  reflexivity.
Qed.

```

However, in FPP, tactic applications should be explicit, i.e. rewrite rules should be supplied with the exact number of arguments required, so that it is clear which part of the goal is being rewritten:

```

...
Proof.
  intros a b c d e.
  rewrite -> (Nat.add_assoc a b (c + (d + e))).
  rewrite -> (Nat.add_assoc (a + b) c (d + e)).
  rewrite -> (Nat.add_assoc (a + b + c) d e).
  reflexivity.
Qed.

```

This issue arises for the tactics `rewrite`, `exact` and `apply`.



## 2.3 The goal: automated intervention on syntax issues

These two issues - abuse and misuse of tactics - correspond to issues of *abstract syntax* (what language constructs are represented in the grammar) and *concrete syntax* (what structures are used to represent language constructs) respectively. These issues are often highlighted in written feedback from the instructor, resulting in much 'superficial' feedback. Superficial feedback is comments on syntax issues, whereas substantive feedback is comments on the logical content of the proof.

`proof-reader` is a tool that anticipates and identifies both abstract and concrete syntax issues. By automatically intervening during the proof editing process, `proof-reader` guides students towards solutions that do not require superficial feedback, allowing the instructor to focus on substantive feedback.

## Chapter 3

# Solution: The proof-reader tool

### 3.1 Parsing student submissions with proof-reader

The proof-reader tool is a parser that emits two types of warnings corresponding to the syntax issues described in the previous section:

1. Warns user of instances where unpermitted tactics are used.
2. Warns user of instances of incorrect arity in terms supplied to tactics such as "rewrite", "exact", "apply", etc.

proof-reader is intended to be used by students, both during proof editing and as a final check before submission. As students are writing proofs, the proof-reader keeps them on the right track by correcting issues that might be affecting their thought process. When used as a final check, it will help them correct proofs that might have been accepted by Coq but do not demonstrate the intended learning goals of the exercise. The proof-reader can be used directly on student submissions by the instructor as well.

## 3.2 Usage

To run proof-reader on your proof script, simply execute the following Emacs command while in Proof General, with the editor focused on the buffer containing the script:

```
M-x jeremy-proof-reader
```

The script will first be re-run from the beginning by Proof General. proof-reader will then evaluate the script and display any relevant warnings in the Emacs response buffer, for example:

```
WARNING: In tactic invocation REWRITE on parent term (Nat.add_assoc a b):  
Term "Nat.add_assoc" with arity 3 incorrectly applied to 2 terms (a),(b).
```

Or, if there are no warnings:

```
No warnings.
```

## 3.3 Setup

Simply download the source code or binary package and follow the installation steps from this repository: <https://github.com/jeremyyew/ync-capstone>.

## 3.4 Examples

### 3.4.1 Example 1: Warning user of instances where unpermitted tactics are used

When proof-reader is applied to the example proof script in the section [2.2.1 Abuse of tactics](#), the output is:

```
Parser error: Could not parse the substring "trivial.". "trivial" may be
↳ an unpermitted tactic, please only use tactics that have been
↳ introduced in the course.
```

### 3.4.2 Example 2: Warning user of instances of incorrect arity

When proof-reader is applied to the example proof script in the section [2.2.2 Misuse of tactics](#), the output is:

```
WARNING: In tactic invocation REWRITE on parent term (Nat.add_assoc a b):
Term "Nat.add_assoc" with arity 3 incorrectly applied to 2 terms (a),(b).

WARNING: In tactic invocation REWRITE on parent term (Nat.add_assoc (a+b)):
Term "Nat.add_assoc" with arity 3 incorrectly applied to 1 terms (a+b).

WARNING: In tactic invocation REWRITE on parent term (Nat.add_assoc):
Term "Nat.add_assoc" with arity 3 incorrectly applied to 0 terms.
```

### 3.5 Possible errors

proof-reader only accepts syntactically correct Coq code. It will first trigger Proof General to reevaluate the entire buffer. As long as Proof General accepts the script without error, proof-reader will process it.

If there are Coq syntax errors, proof-reader will display:

```
Coq error raised. Please correct and try again.
```

The parser will then terminate without evaluating the script. The Coq errors will be in the response buffer, as usual.

Furthermore, proof-reader only accepts a subset of Coq syntax, which has been pre-defined by the instructor (see "Supported syntax" in the Appendix on the repository). Therefore, if the script contains unsupported syntax, it is likely either a command that has been introduced in the course but not accounted for, or a bug. proof-reader will display:

```
Parser error: Could not parse "XXX". This syntax may not be currently  
↪ supported.
```

To extend the supported syntax or modify the parser behaviour, see [4.0.7 Extending proof-reader](#).

## Chapter 4

# Design and Implementation

### 4.0.1 Backus-Naur Form (BNF)

Before writing a parser for a language we must understand the specification of the language. Coq consists of various sublanguages for different purposes, each with their own specifications available in the documentation. The language used to write Coq proofs is the Gallina specification language, which interacts with the language of tactics. These sublanguages are specified in Backus-Naur Form.

The Backus-Naur Form is a notation describing a context-free grammar. A BNF specification consists of expressions on the right - consisting of both 'terminals' (literals) and 'non-terminals' (variables) - represented by non-terminals on the left.

BNF specifications for different languages will have particular conventions but follow the same overall structure. For example, here is a fragment of the BNF specification for the vernacular of Gallina:

```
assertion      ::= assertion_keyword ident [binders] : term .
assertion_keyword ::= Theorem | Lemma
                  Remark | Fact
                  Corollary | Property | Proposition
```

	Definition   Example
proof	<pre> ::= Proof . ... Qed . Proof . ... Defined . Proof . ... Admitted . </pre>

## 4.0.2 The grammar module: a BNF-inspired data structure

Code referenced here from *jeremy-parser/grammar.py*.

`proof-reader` parses a given input string into a syntax tree which can be easily traversed and thereby evaluated. In parsing a language, there are many levels of abstraction that we can utilize. The highest level of abstraction would be a parser generator that directly accepts a BNF specification and returns a fully functioning parser with zero or minimal lines of code needed (I explain why this path is not taken in [5.3.2 Using a parser generator](#).) Through trial and error we eventually arrived at a mid-level abstraction. We define a data structure that resembles BNF and dictates the flow of the parser, provided by the grammar module.

Instead of hard-coding conditional branches to determine matching patterns at each step, the grammar module provides the `GRAMMAR` variable, which is a **map (Python dictionary) from a rule to a tuple containing a regular expression and a list of valid child rules**. As the parser constructs a syntactical unit and its children, it performs lookups to the data structure to understand what rule patterns to apply at each step.

Here is a truncated version of the actual `GRAMMAR` map, with ellipses representing truncated rules; the full object is in the `grammar.py` file. As you can see, each rule mirrors a non-terminal on the left, and its regular expression and children represent its expression on the right. Note the

indentation does not correspond to actual nesting of data, but is intended to visually reflect the nested definitions.

```
import * from constants

GRAMMAR = {
  LABEL_DOCUMENT:
    (None,
     [LABEL_PROOF,
      LABEL_REQUIRE_IMPORT,
      LABEL_SEARCH, #...]),
  #...

  LABEL_PROOF:
    (fr"{KW_PROOF}\. (.*) (? : {KW_QED} | {KW_ADMITTED} | {KW_ABORT}) \. ",
     [LABEL_APPLY,
      LABEL_ASSERT, #...]),
  #...

  LABEL_REWRITE:
    (fr"{KW_REWRITE} ({REGEXP_REWRITE_ARROW}? {REGEXP_TERM_OPTIONAL_SPACE}) \
{REGEXP_IN_OCCURRENCE} {REGEXP_AT_OCCURRENCE} {REGEXP_TACTIC_END} \
{REGEXP_TACTIC_LOOKAHEAD} ",
     [LABEL_REWRITE_ARROW,
      LABEL_TERM]),

  LABEL_REWRITE_ARROW:
    (fr"({REGEXP_REWRITE_ARROW})", []),
  #...

  #...
}
```

For example, the PROOF rule states: *"a proof is a lazily matched substring beginning with the keyword 'Proof' plus a period, and ends with either the keyword 'Qed', 'Admitted', or 'Abort', plus a period. The inner string must consist of any number of child components matching the rules APPLY, ASSERT, etc"*.

Observe:



- For readability and modularity, we factor out keywords and regular expression fragments into the `constants.py` module. We interpolate these constants into regex patterns via Python raw f-strings.
- A terminal node has an empty rule list. For example, `LABEL_REWRITE_ARROW` is a terminal.
- This data structure is simple. While it serves the purpose of this project, see [5.2 Limitations of the grammar module](#).

Having generated a syntax tree, we are now in a position to traverse and evaluate it.

### 4.0.3 Parsing input to generate a syntax tree

*Code referenced here from `jeremy-parser/grammar.py`.*

#### 4.0.3.1 Constructing syntax tree nodes

First, we define a `Node` object.

```
class Node:
def __init__(self, label: str, val: str = None, children: list = None):
    # Each node's label identifies its "type", i.e. which syntactical
    # unit it represents.
    self.label = label
    # Each node's value is the contents of the substring it matched on.
    # This is useful for logging and constructing helpful warning
    # messages.
    self.val = val
```

```
# Each node has a list of children, since each syntactical unit may  
↪ be comprised of sub-components. Hence a node with no children is  
↪ a terminal node.  
self.children = children or []
```

The main parser function `construct_node` is responsible for recursively constructing a syntax tree composed of `Node` objects, and returning it. It accepts an input string `s` and `rule`, the label of the rule that the string has been matched on.

```
def construct_node(s: str, rule: str) -> Node:  
    def construct_children(s: str, parent: str, acc: list) -> List[Node]:  
        if not s:  
            return None, acc, ""  
        _, expected = grammar.GRAMMAR[parent]  
        if expected == []:  
            return s, acc, ""  
        exception = None  
        for item in expected:  
            pattern, _ = grammar.GRAMMAR[item]  
            match = re.match(pattern, s)  
            if not match:  
                continue  
            if item == LABEL_TERM:  
                term_s, remaining_s = get_next_subterm(match.group(1))  
                term = construct_term(term_s)  
                return term_s, acc+[term], remaining_s  
            try:  
                child, remaining_s = construct_node_helper(  
                    match.group(1), item)  
                remaining_s = remaining_s + s[match.end():]  
                return construct_children(remaining_s, parent, acc + [child])  
            except (UnmatchedToken, UnmatchedTactic) as e:  
                exception = e  
        if exception:  
            raise exception
```

```
if parent == LABEL_PROOF:
    raise UnmatchedTactic(s)
raise UnmatchedToken(s)

def construct_node_helper(s: str, rule:str) -> (Node, str):
    term_s, children, remaining_s = construct_children(s, rule, [])
    node = Node(rule, term_s or s)
    node.children = children
    return node, remaining_s

node, _ = construct_node_helper(s, rule)
return node
```

It is only called once by the main program, with the entire script as input. It initiates the construction of the tree by calling `construct_node_helper` and assumes there will be no remaining string to parse (since the root node `DOCUMENT` accepts the entire script).

`construct_node_helper` is responsible for recursively constructing the current subtree, which comprises of both the current node as well as all its children. It accepts an input substring `s` and `rule`, the label of the rule that the substring has been matched on. Only after successfully constructing all its children will it then create the current node and assign the children. It returns both a syntax subtree as well as the remaining string that contains sibling subtrees to be further parsed.

`construct_children` accepts an input substring `s`, its parent `rule`, and an accumulator `acc` containing the child nodes it will return. It returns `term_s` (the value of the parent substring, should this be different from the original substring `s`), a list of child nodes `children`, and the remaining string that contains sibling subtrees to be further parsed, `remaining_s`. It proceeds as such:

- `construct_children` first performs a lookup in the grammar module to obtain the expected children of rule.
- For each rule, it performs a lookup in the grammar module to obtain the corresponding regular expression. It attempts to match the expression against the beginning of `s`.
- On a match, it calls `construct_node_helper` to construct the current child's subtree, and then recurses on the remaining string to construct the rest of the children.

Observe that in `construct_children`, instead of recursing on the rest of the substring following the matched capture group `s[match.end():]`, we return `remaining_s` from `construct_node_helper`, which always demands it from `construct_children`, which is ultimately returned as an empty string in `construct_children`'s base cases, or extracted by `construct_term`. We then recurse on `remaining_s + s[match.end():]`. This is because we cannot rely solely on regular expressions to correctly capture the exact substring containing the next subtree.

When parsing a tactic invocation containing a term, we need to count the parentheses in the term to determine its endpoint. Afterwards, we can continue parsing the rest of the substring, prefixed with any extraneous characters the regular expression might have captured. Therefore, if the current substring matches a term, we make a call to `construct_term`.

#### 4.0.3.2 Constructing terms and subterms

In order to validate the arity of terms supplied to tactics such as `rewrite`, `exact` and `apply`, we need to parse a term into its subterms, which are

grouped in nested parenthesis. Regular expressions are not expressive enough to capture nested patterns. Here is an example substring:

```
exact (my_lemma_1 (my_lemma_2 n1) n2).
exact (my_lemma_3 n3).
```

Suppose we have constructed the first exact node, and now we need to capture the parent term `(my_lemma_1 (my_lemma_2 n1) n2)`, before trying to capture its children `my_lemma_1`, `(my_lemma_2 n1)` and `n2`.

- A lazy pattern on opening and closing parenthesis, such as `\(.+?\)`, would capture:

```
(my_lemma_1 (my_lemma_2 n1).
```

- On the other hand, a greedy pattern like `\(.+\)` would capture everything until the last parenthesis in the substring:

```
(my_lemma_1 (my_lemma_2 n1) n2).exact (my_lemma_3 n3).
```

Hence the regular expressions used in the generic `construct_node` function cannot construct this subtree, and we need a specialized `construct_term` function which is able to count parentheses:

```
def construct_term(term: str) -> Node:
    def construct_subterms(s: str) -> List[Node]:
        if s == "":
            return []

        subterm, remaining = get_next_subterm(s)
        child = construct_term(subterm)
```

```
        children = [child] + construct_subterms(remaining)
    return children

if term and term[0] == "(" and term[-1] == ")":
    term = term[1:-1]
    node = Node(LABEL_TERM, term)
    if re.fullmatch(r"^\s+", term):
        return node
    node.children = construct_subterms(term)
    return node
```

The familiar problem of counting parenthesis is implemented iterative-style in `get_next_subterm`:

```
def get_next_subterm(s: str) -> (str, str):
    k = 0
    for i, c in enumerate(s):
        if c == " " and k == 0:
            return s[:i], s[i+1:]
        elif c == '(':
            k += 1
        elif c == ')':
            k -= 1
    if k != 0:
        raise Exception("Invalid parentheses.")
    return s, ""
```

#### 4.0.3.3 Example syntax tree

Here is an example proof from the lecture notes of week 2 of FPP AY 19/20 Semester 1.

```

Require Import Arith Bool.
Check Nat.add_0_r.

Proposition first_formal_proof :
  forall n : nat,
    n + 0 = 0 + n.

Proof.
  intro n.
  Check (Nat.add_0_r n).
  rewrite -> (Nat.add_0_r n).
  Check (Nat.add_0_l n).
  rewrite -> (Nat.add_0_l n).
  reflexivity.
Qed.

```

Here is the resulting syntax tree, pretty-printed:

```

|- DOCUMENT:
"Require Import Arith Bool...Qed."
  |- REQUIRE_IMPORT:
    "Require Import Arith Bool."
  |- CHECK:
    "Check Nat.add_0_r."
  |- ASSERTION:
    "Proposition first_formal_proof : forall n : nat, n + 0 = 0 + n"
      |- ASSERTION_KEYWORD:
        "Proposition"
      |- ASSERTION_IDENT:
        "first_formal_proof"
      |- FORALL:
        "n : nat"
        |- BINDER:
          "n"
        |- TYPE:
          "nat"
      |- ASSERTION_TERM:

```

```

      "n + 0 = 0 + n"
|- PROOF:
  "intro n.Check (Nat.add_0_r n).rewrite -> (Nat.add_0_r n).Check (Nat.add_0_l
  ↪ n).rewrite -> (Nat.add_0_l n).reflexivity."
  |- INTRO:
    "n"
  |- CHECK:
    "Check (Nat.add_0_r n)."
  |- REWRITE:
    "(Nat.add_0_r n)"
    |- REWRITE_ARROW:
      " -> "
    |- TERM:
      "Nat.add_0_r n"
      |- TERM:
        "Nat.add_0_r"
      |- TERM:
        "n"
    |- CHECK:
      "Check (Nat.add_0_l n)."
    |- REWRITE:
      "(Nat.add_0_l n)"
      |- REWRITE_ARROW:
        " -> "
      |- TERM:
        "Nat.add_0_l n"
        |- TERM:
          "Nat.add_0_l"
        |- TERM:
          "n"
    |- REFLEXIVITY:
      "reflexivity."

```

Observe:

- ASSERTION subtrees have ASSERTION\_IDENT as well as BINDER arguments so the collect\_arity function simply has to store the identifier with the number of arguments.



- TERM subtrees have subterms as children, so the `check_arity` function simply has to validate the number of terms at each depth.

#### 4.0.4 Feature 1: Recognizing unpermitted tactics

In general, when proof-reader encounters unsupported syntax, it will not continue parsing since it is impossible to differentiate the unsupported syntactical unit and the rest of the script. Thus it raises the `UnmatchedToken` exception (see `construct_node` in [4.0.3.1 Constructing syntax tree nodes](#)), which will be used to display the remaining substring.

When it encounters an unrecognized token that might be an unpermitted tactic invocation – i.e. any unsupported syntax within a `PROOF` node – it raises `UnmatchedTactic` instead, which attempts to extract a specific tactic invocation from the remaining substring. This is used to display a more precise warning message, such as the one in [3.4.1 Example 1: Warning user of instances where unpermitted tactics are used](#).

```
class UnmatchedToken(Exception):
    def __init__(self, remaining):
        self.remaining = remaining

class UnmatchedTactic(Exception):
    def __init__(self, remaining):
        self.remaining = remaining
        self.tactic = None
        match = re.match(fr"(.+?) {REGEXP_TACTIC_END}
{REGEXP_TACTIC_LOOKAHEAD}",
                        self.remaining)
```

```
if match:
    self.tactic = match.group(1)
```

### 4.0.5 Feature 2: Verifying arity

Code referenced here from *jeremy-parser/proof\_reader.py*. Having constructed a syntax tree without errors, we are now in a position to traverse and evaluate it. As we traverse the syntax tree, we need to compare the arity of its assertions with assertions that have been defined in the environment – both assertions that have been defined earlier in the script, as well as predefined theorems from built-in libraries that have been imported.

#### 4.0.5.1 Collecting arity signatures of built-in library theorems

We must have a database of arities for built-in library theorems. This only needs to be done once, or whenever new modules are added to the course (see Appendix on repository for modules used in the course).

For each module, we manually issue a Search command, e.g. Search \_ inside Nat to list all theorem definitions in the response buffer. We save each list as a string, process the string so it resembles Coq code, and generate a syntax tree containing those definitions using the same construct\_tree used for input code. We then run collect\_arity, defined below, returning the dictionary arity\_db, a map from assertion names to arities. We save this dictionary as a file, to be loaded whenever proof-reader is run. Later, we add user-defined lemmas to this dictionary we parse the input syntax tree.

This procedure is performed by the script `jeremy-parser/script_parse_theory_lib.py`.

```
def collect_arity(t: Node, arity_db : dict) -> dict:
    assert(t.label == LABEL_DOCUMENT)
    for child in t.children:
        if child.label != LABEL_ASSERTION:
            continue
        assertion = child
        if len(assertion.children) < 3:
            continue
        ident = assertion.children[1]
        forall = assertion.children[2]
        binders = [c for c in forall.children if c.label ==
            ↪ LABEL_BINDER]
        arity = len(binders)
        arity_db[ident.val] = arity
    return arity_db
```

#### 4.0.5.2 Collecting arity signatures of the syntax tree

In `check_arity`, before evaluating the input syntax tree, we use the same `collect_arity` function to collect arity signatures of assertions that have been defined in the script.

```
def check_arity(t: Node, arity_db: dict) -> list:
    #...

    def check_subterms(subterms: list, parent_term: Node,
        ↪ parent_tactic_label: str):
        #...

    def traverse(t: Node):
```

```
#...  
arity_db = collect_arity(t, arity_db)  
traverse(t)  
return warnings
```

We could collect arity signatures and check arity at the same time instead of two separate steps, therefore performing a single traversal instead of two traversals, since the tree traversal is left-to-right, and assertions must be declared before they are referenced. This was the approach in an earlier implementation. However, we implement two separate traversals for the sake of readability. Since each traversal is done in constant time, there is insignificant difference in actual performance. Furthermore, Coq does not allow repeat definitions so we will not have to deal with multiple signatures for the same assertion.

#### 4.0.5.3 Checking arity of the syntax tree

The `check_arity` function can be found in `jeremy-parser/proof_reader.py`. It is responsible for traversing the input syntax tree `t` given a map of arity signatures `arity_db`, and returning a list of warnings indicating instances where the expected arity is not equal to the actual arity. Expected arity is the integer value found in `arity_db` if the assertion has been defined. Actual arity is the number of subterms following the first term. `check_arity` simply explores relevant nodes –`REWRITE`, `EXACT`, and `APPLY` – and counts the number of subterms at each level, generating a warning when arity is incorrect.

### 4.0.6 User interface

*See `jeremy-parser/proof-reader.el` in the repository.*

We implement a user interface by defining a simple Emacs command. The command calls the Proof General function `proof-process-buffer` to process the current buffer and verify that the script is syntactically correct. It uses the hook `proof-shell-handle-error-or-interrupt-hook` to terminate if Proof General emits an error, and only runs the Python program when there is no error. Providing an Emacs command allows students to seamlessly check their script without leaving the editor. The book "Writing GNU Emacs Extensions" (Glickstein, 2010) was helpful.

### 4.0.7 Extending proof-reader

The grammar module abstraction enables syntax to be extended conveniently. To extend the supported syntax - for example, adding a permitted tactic - the instructor simply has to add a rule definition to the grammar module, comprising of a regular expression and the expected child rules. Not all desired syntax may be expressible in the current framework; see [5.2 Limitations of the grammar module](#).

### 4.0.8 Unit and acceptance testing

*Code referenced here from `jeremy-parser/tests.py`.*

`TestParser` performs unit tests to verify that `construct_node` fulfils its specifications. For each test input it compares the generated syntax tree with the expected syntax tree. Each unit test verifies that variations of a particular syntactical unit is correctly parsed.

`TestParserAcceptance` also performs acceptance tests for `construct_node`, running it on sample student submissions which have provided by the instructor.

`TestParityCheck` performs unit tests to verify that `collect_arity` and `arity_check`. It generates and evaluates the syntax tree for each test input, and compares the output warnings with the expected warnings. It performs both positive tests (inputs that should trigger no warnings) and negative tests (inputs that should trigger warnings), and each input contains variations of `exact`, `rewrite` and `apply` syntax.

## Chapter 5

# Discussion

### 5.1 Time and space complexity

Note that the generated syntax trees are expected to be shallow with some constant depth, with most branches having only a few levels, since the grammar contains non-recursive definitions. The only source of arbitrary depth is `TERM`, but the expected depth of `TERM` subtrees for actual student submissions is around 1-3 levels, since highly nested terms are quite rare in simple proofs, and might call for a goal to be factored into a standalone lemma.

#### 5.1.1 Regular expression matching in `construct_node`

Regular expressions can be computationally expensive - for example, they might grow exponentially in complexity when catastrophic backtracking occurs. However, all the regex patterns in the grammar module use lazy quantifiers, thus avoiding backtracking. Furthermore, almost all matches are performed on Coq sentences or fragments of sentences, which are very short substrings. Lastly, since we only accept input that is syntactically validated by `coqtop`, the input is quite predictable.

Therefore we can reasonably assume total  $O(N)$  time where  $N$  is the length of the input string; the tree is expected to be shallow, so the same substring will only be processed some constant number of times. We also have  $O(1)$  space complexity. Counting parentheses in `get_next_subterm` incurs  $O(N)$  time complexity as well.

### 5.1.2 Constructing and traversing the syntax tree

The `construct_node` function takes  $O(N)$  time to construct the entire tree, and  $O(N)$  space on the callstack (since `construct_node` is recursive and Python does not have tail-call optimization), where  $N$  is the length of the input string and the total number of nodes constructed is proportional to  $N$ . A more precise estimate might be  $O(\log_k N)$  where  $k$  is the average number of nodes generated for each level of the tree. Due to the space consumption, Python's recursion limit should be increased if long proofs are expected (currently set to 10000).

`collect_arity` and `arity_check` will explore nodes only at the first and second level, incurring a  $O(N)$  time complexity and  $O(1)$  space complexity.

### 5.1.3 Overall complexity

Therefore, the overall complexity for constructing and evaluating the syntax tree is  $O(N)$  time and  $O(N)$  space. We also run performance tests on real input as well as large input, the results of which can be found in the Appendix on the repository.



## 5.2 Limitations of the grammar module

The grammar module is limited in its expressiveness because of its simple structure. Firstly, rules do not directly express patterns with distinct subcomponents. For example, an assertion is broken down into a keyword, identifier, 'for all' statement, and a term:

```
LABEL_ASSERTION:
    (fr"({REGEXP_ASSERTION} .+?:.+)\". {REGEXP_DOC_LOOKAHEAD} ",
    [LABEL_ASSERTION_KEYWORD,
    LABEL_ASSERTION_IDENT,
    LABEL_FORALL,
    LABEL_ASSERTION_TERM])
```

However, since `construct_node` only expects one capture group, it must iteratively match all subcomponent rules on that single captured substring after the substring has been matched on the assertion. The pattern cannot express subcomponents that appear in different locations of the substring. Furthermore, the structure does not express which subcomponents are required. Each subcomponent rule is treated as optional; it could match on only one of them once, or one of them repeatedly, and as long as the string is eventually consumed the assertion subtree will be treated as a successful match. One solution is for the parser to iterate over a list of capture groups and a list of expected patterns, or a map of named capture groups and their expected patterns.

Secondly, rules do not express alternate patterns. A single regex pattern defines the acceptable structure for each rule. So far we are able to express alternate patterns within the regular expressions as a single pattern with

alternating parts. However, a better solution is to define a rule as a list of pattern/children pairs instead of a single pair, with each pair representing an additional matching option.

Fortunately, these issues do not seem to pose a problem for the input we have tested on, but making the module more expressive would make matching more precise, and improve extensibility.

## 5.3 Alternative approaches

### 5.3.1 Modifying Coqtop source code

Given that Coq's ability to infer missing arguments is an additional feature, it seems natural to modify the source code to provide an option to turn the feature off, as opposed to building an external parser from scratch. There are two reasons why this approach was not taken:

Firstly, and most importantly, a source code approach reduces usability and maintainability of the tool. Unless my changes are accepted into the master branch of the open-source Coq codebase on Github, students will be locked in to the Coq version I worked with. They would not benefit from updates to Coq and might have limited access to the Coq ecosystem. The features described in this report are quite prescriptive and highly specific to FPP's learning goals. While the tool has broader applications as an approach to educators with similar goals, its features may not be relevant to the average Coq user. Granted, we could maintain a modified branch in a separate repository, into which we merge Coq updates. But this involves repeated reintegration and might also introduce dependency or installation issues.

Secondly, a source code approach did not seem feasible. I judged that it was out of scope for this project due to my limited experience with large-scale Ocaml applications. Even if I managed to achieve my desired functionality, refactoring source code might introduce invisible bugs in other components.

### 5.3.2 Using a parser generator

Writing parsers for programming language is a well-understood problem, and parser generators automate the implementation of parsing algorithms. A parser generator accepts a grammar specification and produces a parser that can evaluate the specified language. I spent a significant amount of time exploring the use of parser generators to build my tool. Using a parser generator was appealing because I did not want to 're-invent the wheel', and it promised quick development, high-level abstraction, and high performance.

I tried using several parser generators, for example CEDET's built-in Wisent, and the python package Lark. Each had their own issues. For example, I had some success specifying the grammar of my Coq sublanguage with Lark, but there were often bugs that I had to find workarounds for because I did not understand the error messages. The algorithms were quite complex and I did not have full visibility or understanding of the underlying processes. Furthermore, there were certain functionalities (e.g. collecting and storing previous arity signatures) that did not seem possible in the existing frameworks - modifying the source code was possible but complicated.

Ultimately, for the purposes of my project, writing a parser from scratch gave me granular control of my development process and allowed me to make informed decisions on the level of abstraction to use for different components.

## 5.4 Future work

Firstly, implementing a more expressive grammar structure would improve extensibility and precision, as detailed in [5.2 Limitations of the grammar module](#).

Secondly, implementing the entire program in Emacs-lisp would allow the program to be run directly in Emacs instead of as a child process. This would likely improve performance and eliminate any installation or interoperability issues.

Thirdly, modifications or extensions to the grammar should be made possible at runtime instead of requiring source-code modification. This would avoid rebuilding the project and improve the user experience for the instructor. In this implementation, this can be done by importing the grammar module from outside the binary package, similar to how a parser generator might take a specification as input. Closer integration with Proof General would also allow new library modules to be added via an Emacs command.

Fourthly, performance profiling can be run to identify bottlenecks - string slicing should be eliminated to avoid unnecessary copying, and an iterative implementation might be faster.

## 5.5 Reflections

In the process of developing proof-reader I gained some insights on development in general. First and foremost, we need to have clear specifications from which we can write unit tests. In this project I had a set of example inputs and expected warnings, but would have saved time if I expanded on them by running more acceptance tests earlier in order to discover more edge cases.

Secondly, we should always take advantage of any features that ensure type correctness, as early as possible (I used type annotations from Python's built-in `typing` module). They help us think more precisely especially in the planning phase, which is really where most of the work happens.

Lastly, in designing a program, we need to find the right level of abstraction. Even though using a parser generator was appealing, like most 'automagical' tools, it provided a high level of abstraction and fast development, at the cost of flexibility and understanding. I learned this the hard way after spending an entire semester on it only to switch approaches – but it was a lesson worth learning.

# Bibliography

*Coq homepage*. URL: <https://coq.inria.fr/>.

Danvy, Olivier (2020). “Mystery Functions: Making specifications, unit tests, and implementations coexist in the mind of undergraduate students”. In: *Proceedings of International Symposium on Implementation and Application of Functional Languages*. IFL ’19. New York, NY: ACM. DOI: [10.1145/1122445.1122456](https://doi.org/10.1145/1122445.1122456). URL: <https://doi.org/10.1145/1122445.1122456>.

*Emacs Lisp homepage*. URL: [https://www.gnu.org/software/emacs/manual/html\\_node/elisp/](https://www.gnu.org/software/emacs/manual/html_node/elisp/).

Glickstein, Bob (2010). *Writing GNU Emacs Extensions*. O’reilly Media, Inc.  
*GNU Emacs homepage*. URL: <https://www.gnu.org/software/emacs/further-information.html>.

*Proof General homepage*. URL: <https://proofgeneral.github.io/>.