Learning Support for Writing Proofs in Coq

Context (TODO)

Introduction

The goal of this research project is to provide learning support for students enrolled in YSC3216: Functional Programming and Proving (FPP), by building a tool that checks for syntax issues in student's proof submissions.

FPP is a course in Yale-NUS College taught by Professor Olivier Danvy, under the Mathematical, Computational and Statistical Sciences major. FPP introduces students to the Coq proof assistant, which is a system for writing and verifying formal proofs.

The learning goal 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 that can act as a set of 'safety rails' to guide students towards developing the proper muscle memory. In particular, the program will enforce explicit tactic application within a subset of Coq, amongst other syntax rules. The Lecturer will be able to provide a grammar specification as an input to the tool. The program will be written as an extension of the Emacs text editor, and can therefore be used by students interactively.

Functional programming (FP)

Functional programming is a programming paradigm that models pro- grams as mathematical functions. That is, a program defines a mapping of every possible input to exactly one output value. Functional program- ming is partly characterized by its 'declarative' style, in which the pro- grammer directly expresses the desired output, derived from the input.

Students taking FPP are expected to have completed Intro to Com- puter Science taught in Yale-NUS, which trains them in functional pro- gramming with the language OCaml. Coq has a language of programs that is very similar to OCaml, and is in fact written in OCaml.

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 can be defined as a logical argument about whether a proposi- tion holds. Proofs use logical rules to demonstrate that what we are sure of (an axiom) implies the truth of something we were not sure of.

In mathematical proofs, propositions often contain equations, which are statements asserting the equality of two expressions containing vari- ables (unknown values). In equational reasoning, we apply axioms to equations in order to incrementally transform them into something that is clearly true.

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. Even though they may use jargon and formal symbols, they may be consid- ered informal. Since natural languages are often ambiguous, natural lan- guage proofs are susceptible to misinterpretation or misconception. Fur- thermore, informal proofs rely on humans to check for logical errors, but humans are fallible.

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.

Therefore Coq allows us to write formal, verifiable proofs in a struc- tured logical language called Gallina, and will also automatically verify that our proofs are correct.

Proving is done as such: 1. First, we state a theorem (or lemma, propo- sition, etc) in the logical language of Coq. 2. Then, we solve 'subgoals' generated by Coq (sub-statements we need to prove in order to prove the theorem) by stating a sequence of 'tactics' (the method we use at each step). As we apply each tactic to the current subgoal (by executing each line of code), Coq will progressively transform the subgoal. 3. Once all our subgoals have been transformed into something that is clearly true, our proof is complete. Every proof step has been demonstrated to progress logically from each other; this process can be reproduced by any other user executing the same proof. Thus the the proof is verifiably cor- rect.

YSC3236: Functional Programming & Proving (FPP)

FPP is a course in Yale-NUS College taught by Professor Olivier Danvy, under the Mathematical, Computational and Statistical Sciences major. The class 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).

FPP introduces students to the Coq proof assistant. 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 real- ize that an explicitly written Coq proof exactly corresponds to an equiv- alent mathematical proof they have written in detail, by hand.

Students engage in weekly assignments consisting of rigorous, pro- gressive exercises involving:

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

The GNU Emacs text editor

Emacs is a family of real-time text editors which are characterized by their customizability and extensibility. GNU Emacs was written in 1984 by GNU Project founder Richard Stallman.

The user interacts with files displayed in 'buffers' - a view of a text file - via **commands** invoked by 'macros' - keystroke sequences. Feedback and status messages are displayed in a smaller buffer at the bottom of the screen - the 'minibuffer'. The user can create and dismiss buffers, and multiple buffers can exist without all being on display.

- Emacs is customizable because users can change the behaviour of some commands via parameters, without having to redefine or mod- ify the underlying code of the command itself. Users can also easily redefine key mappings.
- Emacs is extensible because users can write new commands as pro- grams and bind them to new macros.
- GNU Emacs provides a language based on Lisp, Emacs Lisp, that is used to write extensions/programs run within Emacs.

  GNU Emacs is used in Intro CS, Intro to Algos and Data Structures, and FPP, so students are expected to have familiarity with its interface and indeed will be required to use it, since the class uses the Proof Gen- eral interface.

Proof General

Proof General is a powerful, configurable and generic 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 interface presents users with three buffers (windows): one buffer in which the Coq script is to edited, one buffer to display subgoals, and one buffer to display other responses like search results or error mes- sages.

Motivation (TODO)

Building muscle memory

The learning philosophy of FPP is that programming and proving is sim- ilar 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. By the end of the course, students will have independently written more proofs than they have ever written in their lives, and all of these proofs would have been verified by Coq.

Common beginner mistakes

With programming languages, there are usually many ways to write the same program. In the same way, there are many equivalent represen- tations of a Coq proof, because Coq is flexible and allows you to take shortcuts. However, for new learners, this flexibility can be counterpro- ductive. In the context of FPP, several issues arise.

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 '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 uses some heuristics to automati- cally try various strategies to solve the current formula. However, in the first half of the course the focus is for students to understand every sin- gle proof step they write, because if students cannot explain what they are doing, they do not really understand it. Therefore, using a tactic like 'trivial' completely goes against the objective of the exercise. The proof should read:

```
...
Proof.
  intro n.
  rewrite <- (Nat.add_1_l n).
  rewrite <- (plus_Sn_m 1 n).
  rewrite <- (plus_Sn_m 2 n).
  reflexivity.
Qed.
```

Yet these tactics still appear in student submissions, because they might still have the bad programmer mindset of "if it works, its fine". This causes time between resubmissions to be wasted on superficial feedback.

2. Misuse of tactics

Second, even when students use tactics that have been introduced, they may misuse them by taking shortcuts. For instance, the rewrite tactic is used to apply a rule to the current formula. A rewrite rule is a function that expects specific terms in the formula as arguments; Coq will rewrite the given terms. For example, the rewrite rule below accepts three arguments, n, m, 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 you give it. As the example proofs below demonstrate, you could give the rewrite rule three, two, one or zero of the rewrite arguments required, and Coq will simply pick 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 (c + (d + e))).
  rewrite -> (Nat.add_assoc (a + b) c (d + e)).
  rewrite -> (Nat.add_assoc (a + b + c) d e).
  reflexivity.
Qed.
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 the first half of the course, the focus is on understanding the proof at a low level. Clearly, students need to be aware of exactly which terms they have changed at every step. Otherwise, they may get stuck in a proof because they applied a rewrite rule to the wrong term, or they might reach a solution without knowing how. Therefore, taking advantage of this shortcut goes against the spirit of the exercise.

Furthermore, this issue is not easy to check manually, especially with assignments that are hundreds of lines long.

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.

Therefore, it would be nice to have a system that can anticipate and identify both abstract and concrete syntax issues, to save both students and the Lecturer's time and help achieve the learning goals of the course.

3. Not utilizing unfold lemmas

- Content of unfold lemmas is derived directly from the theorem to be proved, and style should be consistent. Therefore writing unfold lemmas is mechanical.
- It could be automated but that defeats the purpose, because its part of the training to understand why it's important. Instead we'll check if they do it correctly, or at all.

The goal: automated intervention on formal issues to build muscle memory

The idea is for the proposed tool to cut down on the amount of **'superficial'** feedback - e.g., 'don't use this tactic, because...', or 'this is bad style, please correct it in this way', etc. - that the Lecturer must give repeatedly to individual students, and instead automatically lead students towards solutions that only require **substantive** feedback - e.g., ideas to pursue, possible restructuring of the proof, etc. The less superficial feedback is required, the more time the Professor can spend on provid- ing substantive feedback. Also, students will spend less effort correcting style errors if they do so immediately.

Yet, superficial feedback is not merely incidental. Superficial feedback reflects the formal concerns of the course and helps reinforces good pro- gramming habits, which will not only assist the learning experience of students, but benefit them in future endeavors. Therefore, the tool does not simply emphasize pedantic concerns; it makes concrete the formal training prescriptions of the course.

Solution: the `proof-reader` tool

Parsing student submissions with `proof-reader`

I implement a parser which provides learning support by checking student submissions and warning them of the three common mistakes described above. Corresponding it has three main features:

1. Warns user of instances where unpermitted tactics are used (TODO).

1. Warns user of instances of incorrect arity in terms supplied to “rewrite” and “exact” tactics.
2. Warns user of missing unfold lemmas when appropriate, and verifies the form of existing unfold lemmas (TODO).

Setup

1. Make sure you have the file `jeremy-parser.el` - it is included in the binary download, but you can also copy or download it from the Github repository.
2. In the script `jeremy-parser` function, change the file path to reflect the location of the Python parser script (TODO: package as binary):

```
(defun jeremy-parser (s)
  "Call parser program in shell and display program output as message."
  (message (shell-command-to-string
      (format "python3 /Users/Macintosh/github/ync-capstone/jeremy-
parser/parser.py --input \"%s\"" s))))
```

3. Navigate to your Emacs initialization file, which might be one of three options: `~/.emacs, ~/.emacs.el, or ~/.emacs.d/init.el.`
4. Insert this line anywhere in the init file: `load ("path/to/jeremy-parser.el")` . The file can be named and located as you like.
5. Restart Emacs. The file will now be loaded whenever Emacs is started.

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-parse-buffer
```

The script will be re-run from the beginning by Proof General and if it is accepted by Coq, `proof-reader` will then evaluate the script and display any relevant warnings in the Emacs response buffer, for example:

```
WARNING: In term (add_comm n):
 Term (add_comm) with arity 2 incorrectly applied to 1 terms (n).
```

Or, if there are no warnings:

```
No warnings.
```

Examples (TODO)

Example 1: Warning user of instances where unpermitted tactics are used (TODO)

Example 2: Warning user of instances of incorrect arity

When `proof-reader` is applied to the example proof script in chapter XX, the output is:

```
WARNING: In term (Nat.add_assoc a b):
 Term (Nat.add_assoc) with arity 3 incorrectly applied to 2 terms (a),(b).

WARNING: In term (Nat.add_assoc (a + b)):
 Term (Nat.add_assoc) with arity 3 incorrectly applied to 1 terms (a + b).

WARNING: In term (Nat.add_assoc):
 Term (Nat.add_assoc) with arity 3 incorrectly applied to 0 terms .
```

Example 3: Warning user of missing unfold lemmas, and verifying existing unfold lemmas (TODO).

Possible errors

The proof script should be syntactically correct Coq code. To confirm this, the parser will first trigger Proof General to reevaluate the entire buffer. As long as Proof General accepts it, the parser will accept 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 "Appendix/Supported syntax"). Therefore, if the script contains unsupported syntax, `proof-reader` will display:

```
Parser error: Unrecognized tokens found.
```

This means that the script should be rewritten without using the unsupported syntax. To extend the supported syntax or modify the parser behaviour, see "Design and implementation/Extensibility".

Lastly, `proof-reader` only checks the arity of terms that have been directly defined in the script, as well as modules that have been pre-registered (i.e. the `Nat`, `Bool` and `Peano` modules). If the problem happens to call for built-in theorems outside of these modules, then this could be a source of false negatives (no warning for incorrect arity) as `proof-reader` will simply not have the arity signatures for those theorems, and will ignore them. But it is more likely that those theorems have not been taught and are not permitted.

Design and implementation

Parsing input to generate a syntax tree

The code referenced in this section can be found in `jeremy-parser/parser.py` unless otherwise specified.

In order to check the program, we parse the input string into a syntax tree that can be conveniently evaluated.

Constructing syntax tree nodes

First, we preprocess the input string to remove any extraneous tabs, spaces, etc in order to simplify the matching logic later on (`preprocess` function in `jeremy-parser/parser.py`).

We define a `Node` object:

```
class Node:
    def __init__(self, label, val=None, children=None):
        # We label every node by its 'type', for evaluation purposes.
        self.label = label
        # The node's actual value (e.g. the identifier of a term) is not needed
for evaluation, but is used for logging and displaying warning messages.
        self.val = val
        # Each node has a list of children, or subcomponents.
        self.children = children or []
```

The recursive function `construct_node` and its helper `construct_children` then consumes the input string by matching regex patterns on the beginning of input substring, while recursively constructing a syntax tree composed of `Node` objects.

- `construct_node`
    - The main parser function `construct_node` returns a syntax tree. It assumes the input has already been matched on a syntax rule `rule`, and `s` is the remaining substring to be evaluated for the node's subcomponents.
    - First, it constructs an appropriately labeled `Node`.
    - It performs a lookup in the `grammar.GRAMMAR` map to obtain the expected children of `rule`.
    - It then calls `construct_children`, and assigns the result to the current node.
    - `construct_children`
        - Attempts to match the beginning of `s` using each rule in the list `expected`.
        - For each rule, it performs a lookup in the `grammar.GRAMMAR` map to obtain the corresponding regex pattern (see 'Design and implementation/BNF-like grammar

abstraction').

- On a match, it consumes the matched substring, recursively generating a subtree using that substring, before constructing the siblings of that subtree by recursing on the remaining string `s[match.end():]` with the same `expected` rules.

```python
# Edited for brevity
def construct_node(s: str, rule) -> Node:
    def construct_children(s: str, expected) -> List[Node]:
        if not s:
            return []
        for item in expected:
            pattern, _ = grammar.GRAMMAR[item]
            match = re.match(pattern, s)
            if match:
                # if item == LABEL_TERM:
                #     child = construct_term(s)
                #     return [child]
                try:
                    child = construct_node(match.group(1), item)
                    children = [child] + construct_children(
                        s[match.end():],
                        expected
                    )
                    return children
                except Exception as e:
                    if str(e) != "No match":
                        raise e
        raise Exception("No match")
    _, expected = grammar.GRAMMAR[rule]
    node = Node(rule, s)
    if expected == []:
        return node
    children = construct_children(s, expected)
    node.children = children
    return node
```

Constructing terms and subterms

The above functions are generic enough to construct most syntactical units in our sublanguage. However, in order to validate the arity of terms supplied to the `rewrite` and `exact` tactics, 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 match on opening and closing parenthesis, such as `\(.?\)`, would capture:
  - `(my_lemma_1 (my_lemma_2 n1)`.

- On the other hand, a greedy match like `\(.+\)` would capture everything until the last parenthesis in the substring:
    - `(my_lemma_1 (my_lemma_2 n1) n2).exact (my_lemma_3 n3).`

Even if we use some strategy to exclude literals to avoid greedy matches across separate tactics, consider this example:

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

Suppose we have captured the entire parent term as well as the first subterm `my_lemma_1`, and now we need to capture the second subterm `(my_lemma_2 n1)`, from the substring `(my_lemma_2 n1) (my_lemma_3 n2)`. However, a greedy match would give us the entire substring

- `(my_lemma_2 n1) (my_lemma_3 n2)`

Hence the generic `construct_node` cannot construct this subtree. Thankfully, this is a familiar problem of counting parenthesis, implemented iterative-style here:

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

Then we just need a specialized `construct_term` function, which mirrors `construct_node` except for two key differences:

- the helper function `construct_subterms` simply looks for the next subterm instead of matching iteratively on a list of expected rules
- we terminate a recursion when the current term has no subterms (there are no spaces, so it is a single term), instead of iterating over a terminal node's empty list of expected tokens.

```
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
```

And now we only need to delegate the construction of term subtrees to `construct_term` by uncommenting the following lines in `construct_node`:

```
for item in expected:
    pattern, _ = grammar.GRAMMAR[item]
    match = re.match(pattern, s)
    if match:
        # if item == LABEL_TERM:
        #     child = construct_term(s)
        #     return [child]
```

(Note: Consider refactoring so that `construct_node` implements `construct_term`, since `get_next_term` has the same output as a regexp.)

Example syntax tree (TODO)

Here is an example proof:

```
Lemma A_1 : forall a b : nat, a + b = a + b.Proof.Admitted.Lemma A_2 : forall (a b : nat), a + b = a + b.Proof.Admitted.Lemma A_3 : forall a b:nat, a + b = a + b.Proof.Admitted.Lemma A_4 : forall (a b:nat), a + b = a + b.Proof.Admitted.Lemma A_5 : forall a b, a + b = a + b.Proof.Admitted.
```

Here is the resulting syntax tree, pretty-printed:

```
|- DOCUMENT:
   ("Lemma A_1 : forall a b : nat, a + b = a + b.Proof.Admitted.Lemma A_2 : forall (a b : nat), a + b = a + b.Proof.Admitted.Lemma A_3 : forall a b:nat, a + b = a + b.Proof.Admitted.Lemma A_4 : forall (a b:nat), a + b = a + b.Proof.Admitted.Lemma A_5 : forall a b, a + b = a + b.Proof.Admitted.")
      |- ASSERTION:
         ("Lemma A_1 : forall a b : nat, a + b = a + b")
            |- ASSERTION_KEYWORD:
               ("Lemma")
            |- ASSERTION_IDENT:
               ("A_1")
            |- FORALL:
               ("a b : nat")
                  |- BINDER:
                     ("a")
                  |- BINDER:
                     ("b")
                  |- TYPE:
                     ("nat")
            |- ASSERTION_TERM:
               ("a + b = a + b")
```

```
      |- PROOF:
      |- ASSERTION:
         ("Lemma A_2 : forall (a b : nat), a + b = a + b")
            |- ASSERTION_KEYWORD:
               ("Lemma")
            |- ASSERTION_IDENT:
               ("A_2")
            |- FORALL:
               ("a b : nat")
                  |- BINDER:
                     ("a")
                  |- BINDER:
                     ("b")
                  |- TYPE:
                     ("nat")
            |- ASSERTION_TERM:
               ("a + b = a + b")
```

Observe:

- `TERM` subtrees have subterms as children, so the `check_arity` function simply has to validate the number of terms at each depth.
- `ASSERTION` subtrees have `ASSERTION_IDENT` as well as `BINDER`s (args) so the `collect_arity` function simply has to store the identifier and count the number of args.

BNF grammars (TODO)

- Explain what is BNF.

BNF-like grammar module

The code referenced in this section can be found in `jeremy-parser/grammar.py` unless otherwise specified.

The `grammar` module provides the `GRAMMAR` variable, which is a map of grammar rules intended to emulate the logic of a BNF grammar notation.

Each key-value pair maps a grammar rule name to a tuple containing a regexp pattern and the children it should (or could) contain:

- `RULE: (PATTERN, [RULE...RULE])`
  - `PATTERN`: A regexp pattern that the parser will try to match the beginning of the current string with. If successful, then the current string contains this syntactical construct, provided the parser can recursively consume the substring contained in the capture group using the children's rules.
  - `[RULE...RULE]`: A list of child rules that the parser will attempt to recursively match, in order, on the captured substring. If empty, this rule is a terminal/leaf node, i.e. a rule that is not defined in terms of other rules.

Here is a truncated version of the actual `GRAMMAR` map. Note the indentation does not correspond to actual nesting of data, but is intended to visually reflect the nested definitions.

```
GRAMMAR = {
```

```
    LABEL_DOCUMENT:
        (None,
         [LABEL_PROOF,
          LABEL_ASSERTION,
          ...]),

        LABEL_PROOF:
            (r"Proof\.(.*?)(?:Qed|Admitted|Abort)\.",
             [LABEL_INTRO,
              LABEL_REWRITE,
              ...]),

            LABEL_INTRO:
                (r"intro\s?(.*?){}".format(REGEXP_TACTIC_END),
                 []),

            LABEL_REWRITE:
                (r"{}\s?((?:->|<-)?\s?\(?.+?\)?){}(?={}|$)".format(KW_REWRITE,
REGEXP_TACTIC_END,
TACTIC_KEYWORDS),
                 [LABEL_REWRITE_ARROW, LABEL_TERM]),

                LABEL_REWRITE_ARROW:
                    (r"(->|<-)\s?",
                     []),
                LABEL_TERM:
                    (r"(.+)",
                        []),

        LABEL_ASSERTION:
            ("(" + ASSERTION_KEYWORDS + r" .+?)\.",
             [LABEL_ASSERTION_KEYWORD,
              LABEL_ASSERTION_IDENT,
              LABEL_FORALL,
              LABEL_ASSERTION_TERM]),

            LABEL_ASSERTION_KEYWORD:
                ("(" + ASSERTION_KEYWORDS + ")",
                 []),

            LABEL_ASSERTION_IDENT:
                (r"\s*([^\s]+?)\s*:\s*",
                 []),

            LABEL_FORALL:
                (r"forall \(?(.+?)\)?,\s*",
                 [LABEL_BINDER, LABEL_TYPE]),

                LABEL_BINDER:
                    (r"([^:\s]+)\s*",
                     []),

                LABEL_TYPE:
```

```
                    (r":\s*(.+)",
                     []),

            LABEL_ASSERTION_TERM:
                (r"(.+)",
                 [])
    ...
}
```

To explain an example in detail:

```
LABEL_PROOF:
    (r"Proof\.(.*?)(?:Qed|Admitted|Abort)\.",
        [LABEL_INTRO,
        LABEL_REWRITE,
        ...]),
```

Here, the `LABEL_PROOF` rule states that *"a proof is a lazily matched substring beginning with the keyword 'Proof' plus a period, and ends with either 'Qed', 'Admitted', or 'Abort', plus a period. Furthermore, the inner string (in parenthesis) must consist of any number of child components matching the rules `intro`, `rewrite`, etc"*.

Observe:

- Rules are identified by constants, which are given by `LABEL`-prefixed names.
- `LABEL_DOCUMENT` is the root node, so it has no prerequisite matching pattern. Thus the first value is `None`.
- For readability and code reuse, we inject constants into a regexp via the string method `format` (string interpolation/templating would be even more readable, but cannot be used together with Python's `r` regexp syntax highlighting.)
- A terminal node has an empty rule list. For example, `LABEL_ASSERTION_TERM` is terminal, and thus the regex pattern simply captures the entire string as its content.

The `grammar` module is an abstraction over the branching logic that the constructor function follows as it recursively constructs the syntax tree, allowing new rules to be defined simply by adding a key-value pair, without modifying the constructor function. Without the abstraction, we would have one logical branch (with repeated branches for multiple references) for each rule, or one function definition for each rule, which would create significant code duplication.

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

Feature 1: Recognizing unpermitted tactics (TODO)

Feature 2: Checking arity

The code referenced in this section can be found in `jeremy-parser/parser.py` unless otherwise specified.

We now have a syntax tree which we can traverse to find errors.

(To be refactored into two functions. Arity check should return only warning data to be formatted separately).

```

def check_arity(t, arity_db):
    warnings = []
    warnings_output = []

    def check_subterms(subterms, parent_term):
        if not subterms:
            return
        first_term = subterms[0]
        if first_term.val in arity_db:
            arity_expected = arity_db[first_term.val]
            arity = len(subterms) - 1
            args = [term.val for term in subterms[1:]]
            arg_strings = ",".join([f"({arg})" for arg in args])
            if arity != arity_expected:
                warning_str = utils.warning_format(
                    parent_term.val, first_term.val,
                    arity_expected, arity, arg_strings)
                warnings_output.append(warning_str)
                logger.info(warning_str)

        if not first_term.children and first_term.val not in arity_db:
            logger.info(
                f"In {parent_term.val},direct term {first_term.val} not seen or
registered.")

        assert((not first_term.children or len(subterms) == 1))

        check_subterms(first_term.children, parent_term)
        for subterm in subterms[1:]:
            if not subterm.children:
                check_subterms([subterm], parent_term)
            else:
                check_subterms(subterm.children, parent_term)

    def collect_arity(assertion):
        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
        logger.info(f"New term {ident.val} arity added in db: {arity_db}.")

    def traverse(t):
        logger.info(f"TRAVERSING {t.label}")
        if t.label in [LABEL_DOCUMENT, LABEL_PROOF]:
            for child in t.children:
                traverse(child)
        elif t.label in [LABEL_EXACT, LABEL_REWRITE]:
            if t.children[0].label == LABEL_REWRITE_ARROW:
                t.children = t.children[1:]
            check_subterms(t.children, t.children[0])
        elif t.label == LABEL_ASSERTION:
            collect_arity(t)
    traverse(t)
```

```
55      return warnings, "\n".join(warnings_output)
```

Since the tree traversal is left-to-right, and assertions must be declared before they are referenced, we could collect arity signatures and check arity at the same time, in a single traversal instead of two traversals. This was the approach in an earlier implementation. However, I decided that two separate functions makes for more readable and maintanable code, with no change to big-O time complexity, and a negligible cost in actual performance.

Collecting arity for built-in library theorems (TODO)

- For each built-in theorem module, used command `Search _ inside Nat` to list all theorems in response buffer.
- Saved each list as a text file and preprocessed before parsing it using the same `construct_tree` and `collect_arity`, to generate `arity_db` dictionary, containing all the theorems' arity signatures.
- Pass `arity_db` into `check_arity` function so that it will recognize library theorems.

Feature 3: Identify missing unfold lemmas, and verify existing ones (TODO)

Extending the parser

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 regex pattern and the expected child rules. For consistency, new label and keyword constants should also be defined in `terminals.py`.

Refer to "Discussion/Limitations of the `grammar` module" to see if the intended syntax addition can be expressed in the current framework, or if the base recursion has to be modified.

Unit tests

The code referenced in this section can be found in `jeremy-parser/tests.py` unless otherwise specified.

For each unit test, the testing apparatus `TestParser` generates the syntax tree and compares it with the expected syntax tree. Each unit test verifies that variations of a particular syntactical unit is correctly parsed.

For each unit test, `TestParityCheck` generates the syntax tree and evaluates the tree to find instances of incorrect arity. It compares the output warnings with the expected warnings. We have both positive tests (there should be no warnings) and negative tests (there should be warnings), and each input contains variations of `exact` and `rewrite` syntax. Negative tests contain nested errors as well as errors with varying number of arguments.

Acceptance tests (TODO)

The code referenced in this section can be found in `jeremy-parser/tests.py` unless otherwise specified.

Discussion

Time and space complexity

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.
- Almost all matches are performed on Coq sentences or fragments of sentences, which are very short substrings.
- Since we only accept input that is syntactically validated by `coqtop`, the input is predictable.
- We try to arrange expected rules in order of frequency of occurence, so we reach a match sooner rather than later, similar to regex alternation optimization. This reduces the number of match attempts on each substring, which would be a constant factor anyway.

Therefore we can reasonably assume total `O(N)` time (where `N` is the length of the input string) and `O(1)` space complexity for regex matching.

Note that counting parentheses in `get_next_subterm` incurs `O(N)` time complexity as well.

The recursive `construct_node` function

- `O(N)` time to construct the entire tree
- `O(N)` space on the callstack, since neither `construct_node` or `construct_children` is tail-recursive

where `N` is total the number of nodes in the tree.

Therefore python's recursion limit should be increased if long proofs are expected (default is 1000).

Traversing the syntax tree in `check_arity`

The depth-first traversal will explore all nodes, incurring a `O(N)` time complexity, where `N` is the number of nodes, which is a constant fraction of the length of the input string. The space complexity is `O(1)`.

String slicing

In Python, strings are immutable, hence all instances of string slicing (e.g. `s[match.end():]` in `construct_node` and `s[i+1:]` in `construct_term`) will create a new copy of the string, which involves `O(N)` time complexity, where `N` is the length of the sliced substring.

This is sometimes a concern for processing long strings. In this case, the expected depth of `TERM` subtrees for actual student submissions is 2-4 levels (highly nested terms would be highly unreadable, and also quite difficult to construct deliberately in simple proofs), so we would only copy the entire substring a few times. Thus, we can assume a worst-case constant multiple of substring length, i.e. `O(kN)`.

Furthermore, string slicing is only used on term substrings, which are expected to be short.

Overall complexity

Therefore, the overall complexity for constructing and evaluating the syntax tree is O(N) time and O(N) space.

Performance (TODO)

- Run on long proof.

Limitations of the `grammar` module (TODO)

- Does not express specific subcomponents of a unit, since the parser function expects to process only one captured substring.
    - Instead, we always expect the captured substring to potentially contain any quantity of any type of child rule in any sequence.
        - In other words, we assume 0...n quantity for each possible child rule; we can't express specific `k`-quantity of any particular child syntactical unit.
        - Furthermore, a captured substring can contain any sequence of all included child syntactical unit, i.e. we cannot express homogenous lists of only one type (or a subgroup) of children.
    - Our sublanguage does not require this feature.
    - If we needed to express this, we could simply have the parser function expect n number of capture groups, and then each rule list could be a list of lists of rules, with each group of expected children corresponding to each capture group, in order.
- Does not express alternate patterns for one rule. A single regex pattern defines the acceptable structure for each rule.
    - Under the current system, if needed, we would construct a distinct rule for each alternate pattern and associated group of expected children, even though logically they may be the same syntactical unit.
    - Our sublanguage does not require this feature.
    - If we needed to express this, we could define a rule as a list of pattern/rule list pairs, with each pair representing an additional matching option.

Alternative approaches

Parsing techniques (TODO)

- bottom up vs top down

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 three reasons why this approach was not taken:

First, and most importantly, source code modification 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.

Indeed, there are many unpredictable factors that determine whether my proposed changes would be accepted - in particular, whether my features are appropriate for general use, which is quite subjective. The features described in this report are quite prescriptive and highly specific to FPP's learning goals. Therefore, I think the tool has broader applications as a possible approach to educators with similar goals, but 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 long-term maintenance and might introduce dependency or installation issues. Furthermore, future changes to the internal structure or implementation of Coq might force a re-implementation of my modifications.

Second: source code modification might not be feasible. After spending some time exploring this option, I judged that working with source might be out of scope for this project due to my limited experience and resources. Navigating and understanding the codebase was difficult primarily due to the sparseness of external-facing developer documentation. The sheer size and complexity of the codebase, coupled with my unfamiliarity with advanced OCaml programming constructs and engineering practices (testing frameworks, build process), contributed to my uncertainty. Even if I managed to get something working, the potential refactoring of existing code might involve wide-reaching changes with invisible consequences on other components.

I would like to emphasize that regardless of feasibility, the issues of usability and maintainability alone outweigh any benefits of the source-code approach.

Lastly, building a parser from scratch ensures that I have a complete of every component of my tool. This allowed me to explore additional features easily (e.g. checking of unfold lemmas), which might have been more difficult to work into an existing codebase.

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. In the initial stages I explored using parser generators to build my tool. Using a parser generator was appealing because:

- I did not want to 're-invent the wheel'.
- It promised quick development, high-level abstraction, and high performance.

I tried using several parser generators (e.g. 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, 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 what went wrong each time. 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 seemed to be more effort than it was worth.

Ultimately, like any tool, parser generators provided a high level of abstraction and fast development, at the cost of customizability and understanding. After all, if you can't explain it, you don't understand it, and obviously I should understand my own Capstone. For the purposes of my project, I realized that writing a parser from scratch gave me full control of my development process and allowed me to make my own decisions on the level of abstraction for different components.

Potential improvements (TODO)

- More expressive grammar structure to make syntax more extensible.
- Tail recursion to avoid O(N) space complexity.
- Implementation in Emacs-lisp for direct execution in Emacs.
    - Likely faster performance.
    - Likely zero installation, dependency or interoperability issues.
- Interaction between Proof General and `proof-reader`.
    - Allow for dynamic addition of new modules, or even for the arity of all terms to be checked dynamically by querying Coq with the "Check" command.
    - Allow for arity checker to register induction hypotheses and other assumptions in the current goal, and check their arity when applied. They are currently ignored.

Reflections

Acknowledgements (TODO)

References (TODO)

Appendix (TODO)

Supported syntax

Questions

- If my regex does not contain greedy quantifiers, does that mean the regex engine in theory never backtracks?
- My parser allows for backtracking, but so far I have not encountered any backtracking scenarios.
    - Do you think there would be any cases where backtracking is ever needed?
    - If not, is the correct term 'unambiguous` or is there another term for languages that do not require backtracking?