1 Installation

To use LaTeXML, you must be using some version of Python 2, and must have `pdflatex` installed on your system. The following Python packages are also required:

- `argparse`
- `xml`

2 Writing a Problem

To write a problem using LaTeXML, you must first understand the basics of the XML encoding being used.

2.1 Problem XML Specification

Here is a sample problem, for the sake of explanation:

```
<problem>
  <used year="2015">Homework 5</used>
  <used year="2014" private="true">Midterm 1</used>
  <version id="2">
    <author>cjk</author>
    <year>2016</year>
    <topics>number_theory graph_theory</topics>
    <types>proof induction</types>
    <param name="modulus">3</param>
    <dependency>tikz</dependency>
    <body>
      The modulus is \modulus.
    </body>
    <solution>
      TODO
    </solution>
    <rubric>
      TODO
    </rubric>
  </version>
  <version id="1">
    ...
  </version>
</problem>
```

A Problem XML file MUST have root tag `problem`, which is made up of one or more child `versions`. Loosely speaking, two 'versions' should be part of the same 'problem' if having the solution to one would make writing the other's solution trivial. For example: adding a part to, theming, or changing the numbers in a previous year's problem would call for a new version of the problem. If several different problems are combined into one, that is generally a new problem (rather than a version of any of them).

Each version MUST have exactly one year (in a `<year>` tag), which may be "Unknown". This is the year that the version was written.

Each version MUST have at least one author, at least one topic, and at least one type. Each of these tags accepts a comma and/or whitespace separated list of items; thus,

```
<authors>cjk nschank, kl47</authors>
```

would be parsed as three authors: cjk, nschank, and kl47. Additional instances of a tag will append to the list, so e.g. multiple `<author>` tags within a version are allowed.

Each of these fields accepts both the singular and plural of their tagname, purely for convenience. Thus...

Authors `<author>` or `<authors>`

Topic `<topic>` or `<topics>`

Type `<type>` or `<types>`

Note that, while either tagame is accepted, the beginning and ending tags must match.

See the next section for an explanation of acceptable values for topic and type.

Each version MUST have a body, a solution, and a rubric. Each of these fields should be filled with arbitrary LaTeX; whatever you would have put between `\begin{document}` and `\end{document}` goes here. It is expected that, if a solution or rubric is not complete, the four letters "TODO" (case sensitive) should appear somewhere within their text. This allows the LaTeXML tools to keep track of any problems that need attention.

As an important note, the characters &, <, and > are special characters within XML. In order for the problem to be parsed, you MUST escape them with the XML sequences `&amp;`, `&lt;`, and `&gt;` respectively. They will be unescaped before being parsed as LaTeX, so should be treated identically to their corresponding characters (e.g. tables will contain many instances of "`&amp;`").

Each version MUST have an `id` attribute, unique within the problem, set to a positive integer, such that the newest version has the highest ID.

A version MAY have zero or more `param` or, equivalently, `parameter` tags, which MUST have a name attribute. This is equivalent to temporarily creating a command `\name` which produces the value given in the tag's field. In the provided example, the command "`\modulus`" will evaluate to 3. This field is intended for use within problems that can be easily changed without needing to create a new version. As an example, the name of a person or object within a problem should be refactored into a parameter, so that the problem can be changed easily.

A version MAY have zero or more `dependency` (also allowed: `dep`, `deps`, `dependencies`) tags, each of which should be a comma and/or whitespace-separated list of packages which are required in order to build the problem. `tikz` is the most commonly included by far. These dependencies will by dynamically imported when building an assignment including this problem.

`usedin` tags should never be created or edited by hand.

2.2 Topics

A **topic** is a unit of the class which a question is attempting to focus on. A problem-version may have multiple topics (we encourage it!), and the topics should be kept specific but not overly so. Any unrecognized topic within a topic list is considered a fatal error.

Allowed topics are:

- basic[1]
- big_o
- bijection
- circuits
- counting
- equivalence_relations
- graph_theory
- logic
- mod
- number_theory
- pigeonhole
- probability
- relations
- set_theory
- todo[2]

If ever confused about what topic(s) a problem covers, consider that the topics field is meant to be organizational. Keep two questions in mind:

1. If Carly were making the final exam, and wanted to include a question on this topic, would she want to see this problem? If so, include the topic!

[1]The 'basic' topic should be included on anything that could go on the first homework, before we have really introduced any definitions. As an example, *Prove that* $\sqrt{2}$ *is irrational* would be in the basic topic.

[2]If the topics field of a version should be looked at later (e.g. you are unsure about one and want to return to it), *always* include todo as a topic.

2. Does the question mention or incorporate a topic, even if the question doesn't seem to be "about" that topic? For instance, does this graph theory question define a relation? Is this probability question asking about circuits? Always include *both* topics – we like to include questions that bring together multiple units, so that students can find connections and recall old definitions.

2.3 Types

A **type** is a characteristic of the question itself. Like topics, questions can (and usually do) have multiple types. Any unknown type is a fatal error.

The acceptable types (and their meanings):

computation Any part of the question requires numerical computation *with little justification* or asks for an explicit instance of something.

Examples: Encrypt this using RSA; determine the size of this set; provide a coloring of this graph; use Chinese Remainder Theorem; show these logical statements are equivalent using set algebra.

Non-example: Convert this sentence to a logical expression (see **notation**); counting questions (these require justification)

core This question is an extremely common question that is definitely simple to Google, but also very important to the class. Some subset of these questions are proven in class every year.

Examples: Prove the infinitude of primes; prove that mod is an equivalence relation; prove that there are an equal number of odd- and even-cardinality subsets of any set.

contradiction Any part of the question explicitly asks for a proof by contradiction, or it is the only feasible way to solve the problem.

contrapositive Any part of the question is a proof by contrapositive.

direct Any part of the question explicitly asks for a direct proof, or it is the only feasible way to solve the problem.

element_method Any part of the question asks for a proof that two sets are equal. Note that for only this proof category, it does not need to ask for element method explicitly.

induction Any part of the question is a proof by induction.

large This question has multiple *small* parts *that build upon each other*, generally while getting progressively harder. If you think your question is hard or long, but does not satisfy the previous sentence, you should fix that.

As a general note, the first part of a 'large' problem should be very easy (often just a computation question meant to ease the student in). See a later section on writing good problems.

notation This question is meant to help the student practice notation.

Example: Put this into set-builder notation.

piece This question is very short, and could probably be incorporated into another problem. This type is a bit subtle: in general, a 'piece' is a question that has one or two parts (usually asking for short proofs) that are not all that interesting by themselves. They are called 'pieces' because, when trying to come up with interesting problems, we can use several of them as inspiration to build a 'large' problem with many small parts.

Example: "Given a set of 52 distinct integers, show that there must be 2 whose sum or difference is divisible by 100." Easy and not very interesting, but could be used very well as a lead-in to a question about a more general property of divisibility and pigeonhole.

proof Any part of the question asks for a non-trivial proof.

repetitive A question which asks you to do the same thing to several different examples.

Example: For each of the following relations, state whether they are injective, surjective, or both; turn the following logical expressions into circuits.

todo If the types field of a version should be looked at later (e.g. you are unsure about one and want to return to it), *always* include todo as a type.