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`

1.1 Requests and Complaints

Please tell Nick the moment something doesn't work, or something is harder than it should be. This is meant to simplify your life significantly: if there is a particular tool you think might help you write things more efficiently, Nick would love to at least hear about it (or tell you why it's impossible, that's fun too). If something is confusing or buggy, also say so – I can't fix something if I don't know it's broken!

Also feel free, if you are good with Python, to submit a pull request in git.

2 Problem Files

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>
  <usedin year="2015">Homework 5</usedin>
  <usedin year="2014" private="true">Midterm 1</usedin>
  <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 the extension '.xml' or it will not be incorporated into PDF builds.

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 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 tagname 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 insensitive) 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 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
- bijections
- 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]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.

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!
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, and is equivalent to a 'tag' in many other contexts. (The name 'type' is used to differentiate from an XML tag.) 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 these logical expressions to English (see **notation**); counting questions (these usually 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. (It does not need to say 'element method' explicitly)

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

large This question has multiple (> 3) *small* parts *that build upon each other*, generally while getting progressively harder. If you think your question is hard or long, but it 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.

needs_work If you come across a problem that you think needs to be looked at again or rewritten, add the 'needs_work' type so that someone can come look at it.

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 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.

2.4 Usage

Look at the next section for a summary of tools to help you create and edit problem files. The section following that looks at how to build problem files into PDFs.

3 The *Edit* Tool

3.1 Creating a New Problem

If you are building a new problem from scratch, you should first make sure that no question like it already exists. You can check this by opening the all-problem PDF (ask the HTAs where this is) and searching for some keywords about your problem.

Then, you should make a new XML file in the root `problems/` directory, wherever makes the most sense to you.[3] There is a simple tool for constructing the skeleton of a new problem XML. On a command line, run:

`python edit.py new` *`filename.xml`*

This command builds a new problem with one version, the current year, your login as the sole author, and other required fields ready to be filled in.

The optional `-i` flag allows you to select the topics and types of the new file with a convenient interactive command-line interface.

3.2 Creating a New Version

If you are adding a new version to an existing problem, the edit tool will let you add a new version with minimal effort. Run:

`python edit.py branch` *`a_problem.xml`* `[-c|-e|-i]`

The final flag is optional, and defaults to `-e`.

`-e` means the new version will be empty; the version number will be correctly incremented, the year will be set to the current year, and the sole author will be your login, but all other fields will be empty or "TODO".

`-i` is the same as `-e`, except it uses the same interactive interface as `python edit.py new` to let you choose topics and types.

`-c` copies the previous version's contents (topics, types, parameters, dependencies, body, solution, and rubric), and updates the version number, author, and year in the same way that `-e` does.

[3] Replace this once there's some kind of organization convention!

3.3 Validating a Problem File

To make you feel reassured that your problem XML looks like it's up to the specifications, you can run

```
python edit.py validate a_problem.xml
```

Validation is actually stricter (but more verbose) than the build tools: it looks closely at things like LaTeX style, probable errors, and possible misspellings of topics which the build tool would simply reject without much explanation. It also prints all likely exceptions simultaneously, rather than halting at the first sign of a problem. You should always run validate, even if your problem builds successfully.

Note that validate cannot help you if your XML is invalid (since the third-party parser will fail), and will not verify that your LaTeX will correctly compile.

Note: The current version does not have some of the features mentioned above (it will have these features within a few days), and is approximately equivalent to the build tool validator.

3.4 Editing Topics and Types

You can start the interactive topic and type editor on a problem using

```
python edit.py edit a_problem.xml
```

Optionally, the flag `--remove-todo` automatically removes the todo topic and type, when fixing such problems.

4 The *Build* Tool

4.1 Building Specific Problems

You can build any problem(s) into a PDF in one step by running the following command:

```
python build.py problems output[.pdf] problem1.xml [problem2.xml ...]
                          [-s] [-m] [-r] [--title ‘‘TITLE’’]
```

The build script is overwrite-safe and tolerant of multiple people building the same pdf simultaneously. In particular, it will build a randomly named temporary file and check for overwriting afterwards, allowing you to specify another name or to overwrite if necessary. The generated `.tex` file is removed, as are the logs and auxiliary files – if you need these for some reason, let Nick know and he'll add an option to keep them. Exception: the `.tex` file is not removed if there were LaTeX errors, so that you can find where the error occurred.

The most recent version (by highest ID) of each problem will be built in order, with repetition if requested. The header on top is temporary and will be removed in a later version of the builder. To change the title of the output, use the `--title` flag (which takes one argument, so may need quotes).

The `-s`, `-m`, and `-r` flags control whether solutions, metadata (like filename and topics), and rubrics are printed, respectively. All are optional, and they can be used together arbitrarily.

An invalid problem XML will be skipped and a warning will be printed, but the build will continue.

4.2 Building All Problems of a Type

Given a directory full of problem XML files, the builder script can search all of them individually and build those that meet a set of criteria. You should run

```
python build.py from problems/  output[.pdf] [CRITERIA]
                        [-s] [-m] [-r] [--title ‘‘TITLE’’]
```

If no criteria are given, the above command builds every problem in the directory `problems/`, with the four optional flags acting just as in the previous section.

The criteria for inclusion can be specified with any or all of the following flags, where a problem will be included only if it satisfies all specified flags.

`--allowed-topics` After this flag, include one or more topic names, separated by spaces. A problem will be included only if its topics are a subset of this list. **Common use case**: List all topics that the students have learned so far, and you will never include problems that they are not qualified to answer.

`--grep` After this flag, include one or more search terms separated by whitespace. A problem will be included only if each search term is present in either the body, solution, or rubric. The search is case insensitive, but note that it is searching the raw LaTeX so is sensitive to formatting and whitespace.

`--required-topics` After this flag, include one or more topic names, separated by spaces. A problem will be included only if it has at least one required topic. Note that it does not need to possess *all* required topics. **Common use case**: List the topics that the homework should focus on to get a good set of tentative problems for a homework.

`--required-types` After this flag, include one or more topic names, separated by spaces. A problem will be included only if it has at least one required type. Note that it does not need to possess *all* required types. **Common use cases**: Finding (for instance) an inductive proof on a particular topic; listing all 'pieces' related to a topic, to get inspiration; finding things that need work.

`--todo` If present, only includes problems that have "TODO" in their solution or rubric. Meant to make it easier to find problems still being worked on.

5 Assignment Files

We don't release problems by themselves - we release assignments! LaTeXML uses a different specification to specify assignments and the problems within them, which simplify the process of building the same set of problems repeatedly.

Assignments are designed in such a way that they may be *finalized*. Finalization is meant to signify that an assignment has been released – or, more accurately, that an assignment's solutions have been released. When finalized, the assignment updates all of its contained problems with a `usedin` tag, allowing later years to know how long it has been since a problem was used. It also reifies the version attribute of all problems, so that if a problem is later given a new version, the old homework will not be changed.

5.1 Assignment XML Specification

```
<assignment>
  <name>Homework 4</name>
  <year>2016</year>
  <due>Tuesday, January 85, 2016</due>
  <problem>problems/problem1.xml</problem>
  <problem>problems/problem2.xml</problem>
  <problem version="4">problems/problem1.xml</problem>
</assignment>
```

Note: The assignment XML specification is still in progress, in the hopes of providing more flexibility in the near future. Thus, any document files not kept in the 22 course folder are not guaranteed to be forward-compatible - Nick will fix any document files kept in the correct location.

All assignments MUST have be given a name (equivalently, a *title*) in a `name` (equiv. `title`) tag. The name is used as the header of the built document, in a very large font.

All assignments MUST have the year that the assignment was made. This year should be accurate– when an assignment is finalized, this is the year the problem will be marked as `usedin`.

All assignments MUST have a due date, which will be used in the header and the introduction of the document exactly as written.

All assignments MUST have at least one `problem`, and an unlimited number (with repeats allowed). These problems will be added in the same order as they are in the XML file, with repetition intact. The *`version`* attribute can be used to mark which version of the problem is desired. If none is provided, the most recent is automatically used.

The text of a problem tag should be the filepath of an existing, validated problem XML file. (If it is invalid or does exist, the assignment build will fail.)

5.2 Building

You can build an assignment XML with the build tool as follows:

```
python build.py doc assign.xml output[.pdf]
                    [-s] [-m] [-r]
```

The `-s`, `-m`, and `-r` flags control whether solutions, metadata (like filename and topics), and rubrics are printed, respectively. All are optional, and they can be used together arbitrarily.

5.3 Finalizing

Finalizing an assignment is simple, and has no options:

```
python edit.py finalize assign.xml
```

6 Features Being Implemented

- Building from a document XML file which allows you to control things like headers, the blurb, problems (including versions), etc. (This is mostly done, but is missing features and the command-line tool isn't written.)
- 'Finalizing' a document XML file: marks that the contained problems were `usedin` a given year, so that we can check for recent usage.
- Edit tool automatically organizing the XML in a canonical way.
- Further criteria: has a TODO in solution or rubric, by year (after, before, unknown)