# Contents

# 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`

The environment variable `LATEXML_CONFIG` must be set to the location of a valid configuration XML file in order to run LaTeXML. If you are a CS22 TA (in the `cs0220ta` group) using a department machine, this should be done for you – talk to an HTA if an error appears referring to `LATEXML_CONFIG`.

If you are on your home computer, the next few sections will help you set up your configuration file correctly.

## 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 feature that would 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.

## 1.2 Installation on Home Computers

First, open a terminal (Linux or Mac) or Cygwin (Windows - currently do not know another option, sorry!). Create a new folder where LaTeXML will be installed (`mkdir ~/latexml`) and switch into it ( `cd ~/latexml`). Download LaTeXML using git by running the following commands:

```
git clone https://www.github.com/nschank/latexml
cd latexml
git submodule update
```

Open your .bashrc file (`~/.bashrc`) [1] and add the line:

```
export LATEXML_CONFIG="/home/login/latexml/ config/config.xml"
```

[1] In some systems, this is .bash_profile.

We also recommend adding aliases to the tools by adding the lines

```
alias 22build="python /home/login/latexml/build.py"

alias 22edit="python /home/login/latexml/edit.py"
```

Now open the config.xml file in ~/latexml/config. Change the contents of the include tag to contain the path

```
/home/login/latexml/include/simple22.sty
```

This is the location of the header file inserted into the top of built documents.

You may optionally add a directory of your preference as the problemroot – this is where 22build all and 22build doc search, but it is not necessary for single problems or for editing.

If you will be working with graphics or other resources (e.g. if you will be including an image or diagram in a problem), you will need to set the resourceroot to an existing directory, and you will need to store that image/diagram there. If you do not plan to work with resources, the resourceroot does not need to be fixed.

You should now run the tests (python tests.py) to make sure that everything is running smoothly.

# 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="3">
   <author>cjk</author>
   <year>2016</year>
   <topics>number_theory graph_theory</topics>
   <types>proof induction</types>
   <param name="modulus">3</param>
   <dependencies>tikz graphicx</dependencies>
   <resource>2016/prettypicture.jpg</resource>
   <body>
     The modulus is \modulus.
     \includegraphics{prettypicture}
   </body>
   <solution>
     Solution to problem goes \textit{here}
   </solution>
   <rubric>
     Problem rubric goes here
   </rubric>
 </version>
 <version id="2">
   ...
 </version>
 <version id="1" standalone="true">
   ...
 </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 may 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. (See the command cheat sheet for an example of how you can use this feature.)

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 more 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.

A version MAY have zero or more `resource` tags, each of which should contain a single relative path to an image or file. This path should be relative to the `resourceroot` specified in the config file[2] and it will be copied into the same temporary folder as the build. Thus, for instance, the example problem file above includes the resource in `2016/prettypicture.jpg`, and is therefore able to use the `graphicx` package (which was included as a dependency!) to show the graphic 'prettypicture'.

A version MAY have the attribute `standalone` set to "true"; "false" is also allowed, and is the default value. If a version is marked as `standalone`, then it is viewable by the build tool even though it is not the most recent version of the problem. This is highly recommended whenever an old version of a problem has very different (and interesting) parts that are no longer present in the new version.

`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[3]
- big_o
- bijections
- circuits
- counting
- equivalence_relations

[2] In the course directory: `/course/cs0220/resources`

[3] 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.

- graph_theory
- logic
- mod
- number_theory
- pigeonhole
- probability
- relations
- set_theory
- todo[4]

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!
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)

[4]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.

**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

The next two sections cover how to effectively edit and render problem files.

# 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 `/course/cs0220/pdf/everything.pdf` and searching for some keywords about your problem.

Then, you should make a new XML file in your home folder (or wherever you wish to work); you should *not* add new files directly into the `/course/cs0220/problems/` directory, since your LaTeX errors may cause builds to break. Move your problem into that directory (wherever makes the most sense to you and your topic group) once it is completed.

There is a simple tool for constructing the skeleton of a new problem XML. On a command line, run:

`22edit new` *`filename.xml`*

This command builds a new problem with one version, the current year, your login as the sole author[5], 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:

`22edit branch` *`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 `22edit 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.

[5] If you are on your local machine, and you have a different login there, you can add an `<author>` tag into your config file, and that will be the one automatically added instead.

## 3.3 Editing Topics and Types

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

`22edit edit` *problem.xml*

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

## 3.4 Validating a Problem File

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

`22edit validate` *a_problem.xml*

Validation is actually stricter (but more verbose) than the build tools: it looks closely at things like LaTeX style, and likely XML errors that can be very difficult to find (e.g. raw ampersands). You should always run validate, even if your problem builds successfully.

Validate will independently render the body, solution, and rubric of your problem, and alert you to the LaTeX errors caused by each rendering.

## 3.5 Validating a Document File

Documents can also be validated! Use

`22edit validate_doc` *an_assignment.xml*

# 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:

```
22build problems output[.pdf] problem1.xml [problem2.xml ...]
                [-s] [-m] [-r] [-k] [--title ‘‘TITLE’’] [--verbose]
```

If making only a single problem, you can use the shortcut:

```
22build single name.xml [-s] [-m] [-r] [-k] [--title ‘‘TITLE’’] [--verbose]
```

which simply names the output `name.pdf` (i.e. the same name as the problem file, except with .pdf instead of .xml).

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 unless you use the `-k` flag. The logs and auxiliary files are always removed – 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.[6]

The most recent version (by highest ID) of each problem will be built in order, with repetition if requested. 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 independently.

The `-k` flag (for *keep*) tells the tool not to delete the intermediary `.tex` file.

An invalid problem XML will be skipped silently, but the build will continue. Use the `--verbose` flag to have the tool print out what files are being added or skipped and why.

## 4.2 Building All Problems of a Type

The builder script can search all of the problems within the root problem directory[7] individually, building those that meet a set of criteria. You should run

```
22build all output[.pdf] [CRITERIA]
```

[6] As a quick tip: use the `-m` flag to more easily find what files contain LaTeX errors.

[7] This is set for you in the department machines, within `/course/cs0220/problems`; on your local machine, you can set it within the config file by changing the path set as the `problemroot`.

```
[-s] [-m] [-r] [-k] [--title ‘‘TITLE’’] [--verbose] [--all]
```

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

Alternatively, you can specify a different directory to search using the `from` option:

```
22build from problems/  output[.pdf] [CRITERIA]
            [-s] [-m] [-r] [-k] [--title ‘‘TITLE’’] [--verbose] [--all]
```

This would search everything within the `problems/` directory.

Normally, as with other build options, the tool will ignore all but the newest version of any problem. If the `--all` flag is used, the tool will also look at any old versions which are marked as `standalone`.

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.

`--authors` After this flag, include one or more logins separated by whitespace. A problem will be included only if it shares at least one author with this list.

`--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 it *is* sensitive to formatting and whitespace.

`--not-used-in` If present, only includes problems that were not used in any of the years given. Use ‘none’ to only include problems that have been used in at least one homework.

`--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 type 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` Standalone flag (i.e. does not take any arguments). 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.

`--used-in` After this flag, include one or more years separated by whitespace. If present, only includes problems that were used in any of the years given. Use 'none' to include problems that have never been used.

`--written` After this flag, include one or more years separated by whitespace. If present, only includes problems that were written in any of the years given.

The command

```
22edit list [CRITERIA] [--verbose]
```

is a quick way to see the names of all files that would be included in a build.

# 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.) Absolute paths are allowed, and relative

paths are appended to the problem root.

## 5.2 Building

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

```
22build doc assign.xml output[.pdf]
           [-s] [-m] [-r] [-k]
```

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 independently.

The `-k` flag prevents the rendered `.tex` file from being deleted.

## 5.3 Finalizing

Finalizing an assignment is simple, and has no options:

```
22edit finalize assign.xml
```

## 5.4 Private Assignments

A *private* assignment is one for which solutions are not publicly released – generally, midterms and finals. You should always mark private assignments private, by setting an attribute in the assignment tag: `<assignment private="`*true*`">`

If a problem was used in a private assignment, it is allowed to appear in searches which ask for problems not used in the same year. So, for instance, a problem used in the 2014 final may still appear when searching with `--not-used-in 2014`.

# 6 Features Being Implemented

- Move to 'templates' that documents can be built into
- Possibly figure out a way to avoid the stupid & and < things. (CDATA?)
- Finalize should render the final TeX documents for archival, so that we are no longer dependent on the problems staying in the same place.

# 7 Course Folder Organization

LaTeXML was created because the 22 course folder was a mess, and was built in the hopes of preventing that from happening again. The only way to make sure that it *doesn't* happen again is to have considerably better institutional memory, even while the staff changes every year.

These homework problems stick around for literally decades – at the time of writing, we have problems going back to more than 13 years ago. Codifying standards will allow this class to keep improving year by year, instead of constantly struggling to just put a single homework together. With that in mind, here are some rules to keep in mind when working in the course folder.

1. **Duplication is your enemy.** The number of person-hours which have been wasted by 22 TAs due to problem duplication is truly staggering. If you make a new problem, you should be SURE that there is not a nearly identical problem lying around.[8] And if you are modernizing an old problem, you should NEVER copy and paste the file – make a new version instead.

2. **Organize conceptually.** In particular, you should not organize the entire problem root by year: organizing by year will make duplication far too likely, and will bias the staff towards using more recent problems.

   Instead, we recommend the following. Break up the problems into a few directories based on topic (e.g. graph theory, number theory...). Each of these folders should have a subdirectory called `old`, which contains all problems that have not been looked at by the *current year*'s staff. When a problem is looked at and fixed up, it should be moved into the main topic directory. Conversely, at the end of the year, all problems should be put back into the `old` subdirectory of their topic.

3. **Finalize assignments.** If you finalize your assignments, later years will be able to much more easily find the problems they can use. Finalizing an assignment marks each problem as used in the correct year, and the build tool can filter out problems used too recently, e.g. `--not-used-in 2014 2015`.

   Take advantage of *private* assignments, which allow you to mark an assignment as a midterm/exam which did not have solutions released. The problems within it can be reused soon afterwards, and the finalize tool will mark the problems as used privately (allowing the build tool to not filted it out).

4. **Organize resources.** The resource directory, found at `/course/cs0220/resources`, should be kept neat and tidy. Contrary to the problems folder, the resources folder *should be organized by year*. In fact, more specifically, you should never edit or alter ANYTHING in a previous year's resources. Please instead copy old resources into your own year.

   Reasoning: images and graphics have historically been very difficult to keep straight, because everything keeps moving around. Old problems would reference some old image, but it had

[8] Use `22build --grep`!

been renamed since, and somebody edited the image too, so now the problem makes no sense. No more! In later years, old versions will reference an image which has the same filename and same contents as when the problem was written.

5. **Be lenient when adding topics.** It is much better, organizationally, if you include all topics that are relevant to your problem... even if it is not the focus of that problem. By combined usage of the `--allowed-topics` and `--required-topics` tags, you will have a much easier time finding appropriate problems.

   As an example, let's say you write a question that focuses on graph theory, but incorporates some set theory, and probability: you should DEFINITELY include all three of them as topics. If we are looking for a question on the set theory homework, it is easy to not include graph theory questions, so the inclusion of the topic does no harm. But the extra information is still useful if we are looking for a question that connects back to set theory when making a graph theory homework.

6. **Validate constantly.** Spam the validation tool. It is the most important and rapid way of making sure that your problem is up to consistently set standards, which will save time for everyone overall. (Not to mention it is fairly good at finding likely errors.) Documents can also be validated!

7. **Mark versions `standalone`.** If an old version is distinctly different from the current version, and might be useful to see later, then mark is as standalone so that it is not ignored by the build tool. This is especially recommended if:

   - The version is a list of lots of different parts that are only going to be used one at a time.
   - Two problems have been combined into one. Mark the old problems as standalone versions, and have the combination be a distinct version.

# 8 LaTeX Style

In order to keep the formatting of LaTeX consistent between problems, we have a few special macros and stylistic choices you are *required* to use.

1. You should **ALWAYS** use the environments `22itemize` and `22enumerate` rather than `itemize` and `enumerate` respectively. Specifically, these environments change the spacing between items, and use a consistent lettering scheme.

   `22enumerate` resumes numbering within a single problem if you leave the environment and then enter it again.

2. When providing a hint or a note, you **MUST** use the commands `hint` and `note`, as in

   ```
   \hint{Think about the pigeonhole principle.}
   ```

   This allows us to change how all hints and notes look without seeking them out individually.

3. The newline commands (e.g. \\) should **NEVER** be used to make paragraphs or for spacing of text. To make a new paragraph, leave a blank line between two blocks of text, as in:

   ```
   \begin{document}
     First paragraph.

     Second paragraph.
   \end{document}
   ```

   This does *not* apply to tables or other such environments; you should use \\ as you normally would.

4. All parts to a question should end with a period (or question/exclamation mark). You'd be surprised how often people forget this!

5. **NEVER** use the double-dollar-sign notation (e.g. `$$x$$`); in fact, don't do it in real life either. It is error-prone and difficult to identify using i.e. regexes. Use `\[x\]` instead.

   (Single dollar signs are still the preferred notation for inline math.)

6. **ALWAYS** use `\pmod` to write things in mod notation.

   **Example**: `3\equiv 4\pmod 5` compiles to $3 \equiv 4 \pmod 5$.

   **Wrong**: `3\equiv 4\mod 5` compiles to $3 \equiv 4 \mod 5$, has weird spacing and we wish to maintain consistency.

   **Very Wrong**: `3=4 mod 5` compiles to $3 = 4 mod 5$ and will lead to spontaneous crying among the staff.

7. Keep lines under 80 characters to remain editor friendly. The validator enforces this too!

8. Use `\setminus` to write the 'backslash' used in set algebra; the text backslash character does not provide the right spacing, and is less readable in the LaTeX.

## 8.1 22-Specific Macros

Here is a small table of other convenience macros that are available to you. We **highly recommend** them, as they cut down on (1) useless clutter, and (2) minor typos.

| Symbol | Command |
|---|---|
| $\mathbb{N}$ | `\N` |
| $\mathbb{Z}$ | `\Z` |
| $\mathbb{R}$ | `\R` |
| $\mathbb{Q}$ | `\Q` |
| $\mathrm{E}[X]$ | `\E[X]` |
| $\mathrm{V}(X)$ | `\V(X)` |
| $\{x \mid x \in X\}$ | `\setbuilder{x}{x\in X}` |
| $\mathcal{P}$ | `\Pow` |
| $\Pr[\,]$ | `\Pr[ ]` |

Note that the `setbuilder` command does not deal with multiple line equations.

## 8.2 LaTeX Tips

Here are some tips to writing LaTeX more effectively.

- The commands `\left` and `\right` can be used with delimiters like {}, `()`, and `[]` to make them grow with their contents. So, for instance, `[\binom{n}{r}]` looks like

$$[\binom{n}{r}]$$

  while `\left[\binom{n}{r}\right]` looks like

$$\left[\binom{n}{r}\right]$$

  The matching command `\middle` can be used with the pipe character | to produce effective set builder notation (and that's exactly how `\setbuilder` does it!)

- Trying to write implies statements or iff statements? You are looking for `\Rightarrow` ($\Rightarrow$) and `\Leftrightarrow` ($\Leftrightarrow$).

- Be careful with XML special characters &, <, and >! See the section on special characters to avoid confusing XML parse errors. (Thankfully, XML color coding should make these errors relatively easy to spot.)

## 8.3 Adding to This List

Contact Nick and offer more tips or stylistic decisions, if you think they are important! We are very willing to add useful commands into our built-in macros.

# 9 Command Cheat Sheet

| Command | Result |
|---|---|
| `22edit new problem.xml -i` | Create a new problem file called `problem.xml`, and specify its topics and types using the `edit` tool |
| `22edit edit problem.xml` | An interactive tool for changing the topics or types of a problem |
| `22edit branch problem.xml -c` | Adds a new version to the problem, copied from the previous version. Omit the `-c` to add an empty version. |
| `22edit validate problem.xml` | Finds common mistakes and ensures a file conforms to the style guide. All problems you write should successfully validate before you put them into the problem folder. |
| `22build single problem.xml -rms` | Renders a single problem (with its metadata, rubric, and solution) into `problem.pdf` |
| `22build all` *`output.pdf`* `--required-topics logic counting -rms` | Renders all problems in the problem root that have either logic or counting as one of their topics. |
| `22build all` *`output.pdf`* `--authors` `` `whoami` `` | Renders all problems in the problem root that you helped write. |
| `22build all` *`output.pdf`* `--authors` `` `whoami` `` `--todo -rms` | Renders all problems in the problem root that you wrote, and that still need a rubric or a solution. |
| `22build all` *`output.pdf`* `--grep foo bar -rms` | Renders all problems in the problem root which contain the text 'foo' and the text 'bar' (case insensitive). |
| `22build all` *`output.pdf`* `--written 2016` | Renders all problems in the problem root which were written this year. |
| `22build from .` *`output.pdf`* `*.xml` | Renders all problems in the current directory, without going into subdirectories |
| `22build list --written 2016 --verbose` | Quick way of seeing what recent problems are not compiling. |