MathML Parser Design

# Pattern Database

The MathML parser prepares for parsing by reading a pattern database that contains all the patterns that the parser needs to recognize. The grammar of a pattern is the following:

variable [categories] = mathml<attribute="regular expression", ...>(...) | variable | [category] | "regular expression" | ? | + | # -> 'output {1} to {2} of {3}, {4} ...'

A pattern seeks to collapse a collection of elements into a single entity with a single output. The elements it collects can be used to help in the creation of the final output.

# Overview of Flow from MathML to Prose

MathML must somehow be fed into the parser, from which then it outputs the correct prose. It does this using a number of steps:

The MathML is converted to a data structure that is comparable to a pattern parsed from the database

For each pattern

1. The pattern is first compared with the main MathML tree
2. All matches are labeled and marked
3. Then, matches that are marked are removed and replaced with the appropriate speech object
4. All speech object’s children are then pattern matched like the main MathML tree
5. The final structure is checked to make sure it has collapsed into a single speech object successfully
6. A single string is extracted by evaluating every speech object and its dependencies recursively

# Common Pattern Tree

There must be a datatype that can successfully compare a MathML tree and a pattern tree in the same manner. From now on, this will be called a **PatternTree**.

For each node in a **PatternTree**, it can be one of many things:

* Variable, referring to a previously matched pattern
* Category, referring to a category of matched patterns
* XML, the tags that make up MathML
* Text, regular expressions that match the text inside of tags
* Collector, a wildcard that accumulates and matches both XML and Variables, such as \*?\*, \*+\*, and \*#\*

Initially, the MathML will be converted to a **PatternTree** that has solely XML elements. For the pattern that will be used to match up with the MathML, it will be made up of Variable, Collector, and XML elements. The eventual Variable that will replace the matched pattern is **not** a part of the pattern's tree.

# How to Match Patterns

The match process essentially just marks where all of the matches are by marking one node for replacement, and then the rest for removal. The replacement is a separate portion of the algorithm.

The pattern's first element is constantly tested against the MathML tree by iterating through its elements using depth-first traversal. Whenever the first element is found, 2 things happen:

* The iterator is copied in case we need to backtrack if pattern matching fails
* A list is created for all of the nodes that are in the matched pattern. This list will be used later to mark the elements for replacement or removal

Determining whether a pattern matches a given element in the MathML tree is based on multiple criteria. There is no real order, but here is a list of all possible criteria:

* If pattern element is a Variable, it only matches a Variable
* If pattern element is a Category, it matches Variables that have Category in its list of categories
* If pattern element is a XML, it only matches an XML
* If pattern element is a Text, it only matches a Text with same content
* If pattern element is a Collector, multiple things can happen:
  + If it's a +, it matches all elements up to element that matches with the +'s sibling. If + has no siblings, then it matches all of them
  + If it's a #, same deal as a + with a different means to store them
  + If it's a ?, it matches any 1 element

# How Matches are Stored

If a pattern matches and element successfully, the match is stored in the following way:

* If pattern element is a Variable, it is stored as ('variable', [element])
* If pattern element is a Category, it is stored as ('category', [element])
* If pattern element is a XML, it is stored as ('xml', [element])
* If pattern element is a Text, nothing happens
* If pattern element is a Collector, multiple things can happen:
  + If it's a +, it is stored as ('+', [elements])
  + If it's a #, it is stored as ('#', [elements])
  + If it's a ?, it is stored as ('?', [element])