Eberhard Karls Universität Tübingen
Mathematisch-Naturwissenschaftliche Fakultät
Wilhelm-Schickard-Institut für Informatik

Master's Thesis in Media Informatics

UnicodeMath ⥅ MathML

Implementation of a UnicodeMath to MathML Translator and its Integration Into Markdeep

Noah Doersing

November 30, 2019

Reviewers:	Prof. Dr. Torsten Grust	Prof. Dr. Thomas Walter
	Database Systems Research Group	Arbeitsbereich Informationsdienste & Direktor des ZDV
	Wilhelm-Schickard-Institut für Informatik	Wilhelm-Schickard-Institut für Informatik
	Universität Tübingen	Universität Tübingen

ABSTRACT

While it is centered around the titular UnicodeMath to MathML translation (abbreviatory *UnicodeMathML*), the work presented in this thesis is composed of multiple parts. They are listed here in descending order of importance:

- Conversion of *UnicodeMath* input into *MathML* for rendering (*e.g.*, in Web browsers).

  UnicodeMath is a linear encoding of math formulas that, in contrast to the more well-known LaTeX, takes full advantage of Unicode's plethora of math symbols. For example, the UnicodeMath expression a⃗ⁿ (composed of the three Unicode characters `U+0061 LATIN SMALL LETTER A`, `U+20D7 COMBINING RIGHT ARROW ABOVE`, and `U+207F SUPERSCRIPT LATIN SMALL LETTER N`) encodes $\vec{a}^n$.

  MathML is a W3C standard that defines an XML-based encoding of math formulas. Think *HTML, but for math*. Some browsers can render it natively, others need a readily available JavaScript polyfill.

- Integration of this JavaScript-based translator into Morgan McGuire's open-source Markdown engine *Markdeep* to make UnicodeMath immediately usable in day-to-day document authoring.

  The guiding principle of this work was to only carry out minimally invasive changes to the Markdeep source code.

  In addition, the integration of the translator into arbitrary HTML documents was successfully explored.

- Implementation of a web-based UnicodeMathML *playground*.

  Originally intended as a parser development aid, this tool enables writing of UnicodeMath expressions with instant preview, character info, control word substitution, math font selection, and other features.

- During the thesis period, I have developed other Markdeep-adjacent tools. Although not strictly related to the UnicodeMath to MathML translation, they were instrumental in producing both this document and a presentation on the subject:
  - `markdeep-thesis` for generating undergraduate and graduate theses such as this one using Markdeep and Bindery (a JavaScript library for creating printable documents with HTML and CSS),
  - `markdeep-slides` for building presentation slides with Markdeep and presenting them in the user's browser, and
  - `markdeep-diagram-drafting-board` for user-friendly authoring of ASCII art diagrams that Markdeep then converts to SVG drawings.

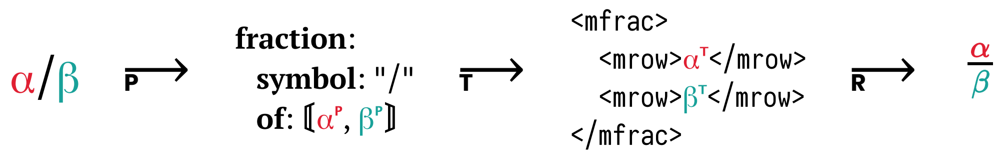
NOTES

This document was generated using the custom-built tool `markdeep-thesis` (the implementation of which is touched upon in Section 6.2.1), which makes use of Markdeep[1] and Bindery[2].

Body text is set in PT Serif[3], headings utilize Poppins[4], code shines with Iosevka[5], references to footnotes, figures, and sections are indicated with PT Sans Narrow[6], and Markdeep's `.fancyquotes` are made appropriately fancy using Aleo[7].

Some somewhat noteworthy notes on notation:

- Unicode code points – along with their names – such as U+1F4A9 PILE OF POO, are set in the tall, narrow font Teko[8] as they can get rather long (the longest happens to be U+FBF9 ARABIC LIGATURE UIGHUR KIRGHIZ YEH WITH HAMZA ABOVE WITH ALEF MAKSURA ISOLATED FORM).
- Paths of files, such as ☁`/thesis/thesis.md.html`, are usually denoted beginning with U+2601 CLOUD. This symbol signifies the root directory of the `unicodemathml` repository.
- All 154 mathematical expressions in this document have been written in UnicodeMath, automatically translated to MathML by the translator discussed in this thesis, and rendered using MathJax.
- When referencing "Section ⋯ of the tech note", I'm referring to Murray Sargent's Unicode Technical Note "UnicodeMath: A Nearly Plain-Text Encoding of Mathematics" [Sargent16]. Similarly, "page ⋯ of the digital typography book" refers to Donald Knuth's anthology of essays, articles, talks, and Q&A session transcripts released as "Digital Typography" [Knuth99].
- Instead of in typical inference rule notation, many kinds of transformations – such as parsing (indicated by **P** below the arrow), translation and pretty-printing (**T**) and rendering (**R**) – are expressed as follows. More details in Chapter 4.



[1] See `https://casual-effects.com/markdeep/`.

[2] See `https://evanbrooks.info/bindery/`.

[3] See `https://fonts.google.com/specimen/PT+Serif`.

[4] See `https://fonts.google.com/specimen/Poppins`.

[5] See `https://typeof.net/Iosevka/`.

[6] See `https://fonts.google.com/specimen/PT+Sans+Narrow`.

[7] See `https://fonts.google.com/specimen/Aleo`.

[8] See `https://fonts.adobe.com/fonts/teko`.

CONTENTS

1 INTRODUCTION

“ you gotta hand it to the unicode consortium, no one quite opens a can of worms like they do ”

— Brendan Hennessy[1]

Although the above comment is about the politics of Unicode's flag emoji encoding and operating system vendors' implementations thereof, it rings true even for such mundane subjects as the encoding of mathematical formulas: Were it not for the valiant standardization efforts of the Unicode consortium, this thesis could not have been written and I would've saved myself a whole bunch of work.

Math typesetting and, by extension, the encoding of mathematical expression on the path from the author's imagination to a rendered image on a screen or a piece of paper has been an area of active development since the invention of the printing press. Among the many approaches developed in recent decades (see Section 2.2), UnicodeMath stands out:

Spearheaded by Murray Sargent, a physics professor emeritus[2] and more recently a “Microsoft [senior] software design engineer [...] who was [...] a key participant in the implementation of mathematics support in Microsoft products” [Beeton16] whose SCROLL language, developed in the late '60s, was “the first language capable of ‘typesetting’ mathematical equations on a computer” [Sargent06], UnicodeMath makes liberal use of Unicode

[1] See `https://twitter.com/bphennessy/status/1147903246977179650`.

[2] See `https://unicode.org/iuc/iuc29/bios.htm`.

symbols (see Section 2.1) to encode mathematical concepts.

For example, the rendering equation, one of the fundamental equations in computer graphics – it formally describes the light transport through surface points in three-dimensional scenes – is written as

$$I(x,x')= g(x,x')\left[\varepsilon(x,x')+\int_S \rho(x,x',x'')I(x',x'')\,dx''\right]$$

in the paper [Kajiya86] that introduced it. It can be expressed in UnicodeMath fairly human-readably:

```
I(x,x') = g(x,x') [ε(x,x') + ∫_S▒ρ(x,x',x'')I(x',x'')ⅆx'']
```

When translated into Donald Knuth's (La)TeX, which is the most popular *linear format* for math in use today, the same equation is a bit more verbose:

```
I(x,x') = g(x,x') \left[\epsilon(x,x') + \int_S \rho(x,x',x'')I(x',x'')
                          \,dx''\right]
```

And in MathML, an XML-based encoding of mathematics developed for the web (more details in Section 2.5), it's become completely unreadable for most people (but trivially easy to parse for computers!):

```
<math xmlns="http://www.w3.org/1998/Math/MathML" display="block">
  <mi>I</mi>
  <mfenced open="(" close=")">
    <mrow>
      <mi>x</mi>
      <mo>,</mo>
      <msup>
        <mi>x</mi>
        <mo>′</mo>
      </msup>
    </mrow>
  </mfenced>
  …
</math>
```

(That was just the bit before the equals sign – it doesn't get any less verbose later on in the equation, either.)

“UnicodeMath is used for keyboard entry of mathematical expressions in Microsoft Word, PowerPoint, OneNote and Excel.” [Sargent16] Through the work detailed in this thesis, UnicodeMath has also become applicable for encoding mathematical expressions in Markdeep (see Section 2.6) and HTML documents. This work is divided into multiple aspects, contextualized in Figure 1 on the next page, which will, as you progress through this document, hopefully make more sense than it may presently.

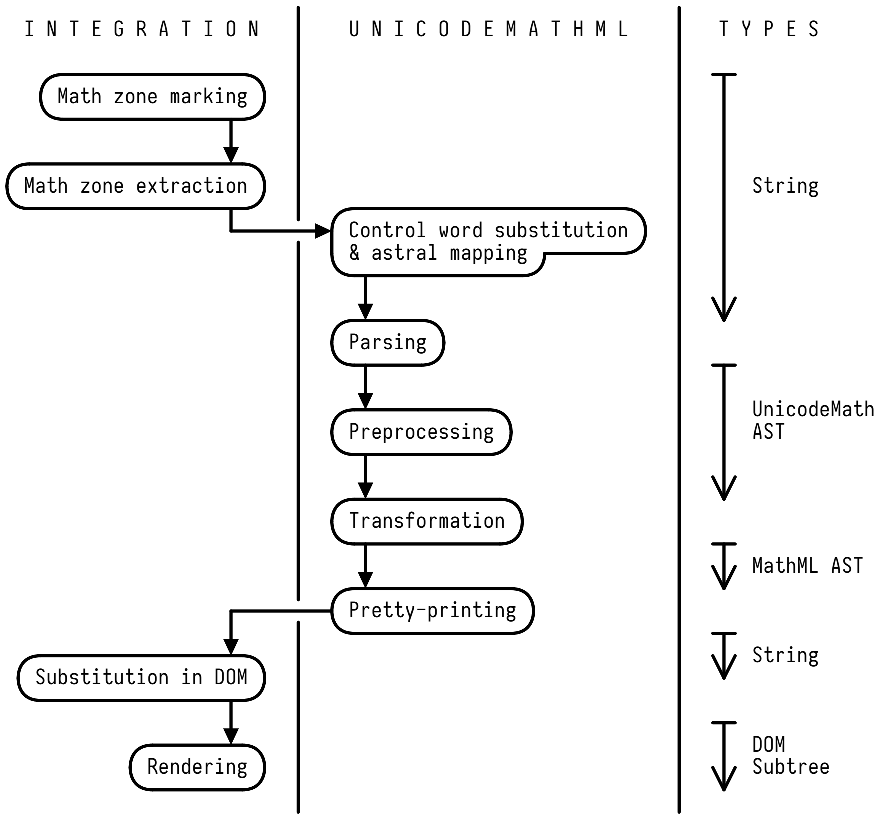


Figure 1: *An overview of the UnicodeMathML pipeline.*

This thesis is divided into several chapters:

First, I will attempt to bring you up to speed on the technologies I relied on in building UnicodeMathML: Unicode, MathML, HTML, JavaScript, MathJax, Markdeep, and the parser generators PEG.js and ANTLR. In this **background** chapter, an initial review of UnicodeMath and a short history of linear math encodings are given, as well. Finally, prior and related work is discussed.

Before getting to the meat of this thesis, I will swiftly discuss the **implementation architecture** – basically the center and right columns of Figure 1. It helps to get an overview, plus there are some aspects of it that are important, but can be ignored during parsing and/or translation.

UnicodeMath's syntax is explored in greater detail in a chapter partially focused on **parsing UnicodeMath**. Its details are explained in lockstep with a run through the PEG.js grammar I wrote in order to generate a UnicodeMath parser. In the follow-up [Siracusa11] sections, I will go over the abstract syntax tree (AST) that's built up during parsing, shining the light on one UnicodeMath construct at a time. Just after I handle the parsing of each UnicodeMath construct, it's all about **translating it into MathML** – it makes more sense to do this in an interleaved fashion rather than explaining parsing first, translating second. As part of this, I will go into various MathML details and quirks as I describe the transformation process.

With this main contribution of the thesis out of the way, I will next write about the UnicodeMathML pipeline's **integration into Markdeep and HTML** documents – there happen to be multiple ways of going about this, so I explored two of them. The left column of Figure 1 provides an overview of the final implementation.

During the implementation of UnicodeMathML, I have developed a web app that eases input of UnicodeMath expressions, with instant preview of the MathML translations of UnicodeMath expressions, as well as Unicode character information, control word substitution, math font selection, and other features. This UnicodeMathML playground is detailed in a chapter on **ancillary work**, along with an overview of several Markdeep-based tools I have built during the thesis period: a typesetting tool for theses such as this one, a tool that turns a Markdeep document into presentation slides, and a diagram drawing aid. Finally, a set of Python utilities that implement minor Unicode-related data transformations is discussed.

In the following chapter, I will give an **evaluation** of my work – I'll write about aspects that were challenging to me before analyzing the performance of the UnicodeMath to MathML translator.

Finally, some musings on potential **future work** – some smallish todos that I will likely tackle in the coming weeks, along with big-ticket items that would be interesting to explore and/or nice to have, but which I might just leave up for grabs.

2 BACKGROUND

This chapter serves to introduce the technologies used in this thesis, from Unicode all the way to the parser generator ANTLR. For most of them, I will briefly summarize relevant historical aspects, along with introductory examples where appropriate. A section on prior and related work concluded this chapter.

2.1 UNICODE

“We began Unicode with a simple goal: to unify the many hundreds of conflicting ways to encode characters, replacing them with a single, universal standard.”

— Mark Davis, President of the Unicode Consortium [Davis06]

In the 1980s, computers became smaller, less expensive, more usable for the layperson, and thus more widespread. Combined with the proliferation of full-grown networking, this presented some issues as different software stacks would try to talk to each other: A textual message, encoded using whichever encoding Alice's computer used, might come out garbled and utterly unreadable on Bob's end if his computer didn't support the original encoding and instead used its own preferred decoding scheme to interpret the message.

For example, the Swedish word “Smörgås”, when encoded using the ISO 8859-1 standard but interpreted as Mac Roman (the default encoding of the classic Mac OS), would decode to “Smˆrgås”.

This phenomenon is known as Mojibake[1] (文字化け, “character transformation”) – one can imagine that Asian languages, due to their higher complexity[2] of encoding, were more affected by this than languages that predominantly rely on the Latin alphabet.

The Unicode project aims to address this set of issues by providing a standard mapping from *code points*, *i.e.*, numbers, to characters. Since Unicode can be encoded using UTF-8, UTF-16, and other encoding schemes, this does not entirely solve the Mojibake problem and related issues (which caused a significant road block in this thesis, see Section 7.1.1), but delivers a baseline for digital representation of textual data.

Among the ~7000 living languages[3] and plenty more dead languages Unicode supports, many of the symbols, character variations, and notational subtleties native to mathematical expressions are present. This enables “[support for] a wide variety of math usage on computers, including in [...] languages like TeX, in math markup languages like MathML and OpenMath, in internal representations of mathematics for applications like Mathematica, Maple, and MathCAD, in computer programs, and in plain text.” [Beeton17]

Table 2.2 of “Unicode Technical Report #25: Unicode Support for Mathematics” (cited in the previous paragraph and co-authored by Murray Sargent) provides a good overview of where in Unicode's 17 planes the code points most relevant for math are located. A plane is a continuous group of 2^{16} code points.

- Most of the world's writing systems are encoded in Plane 0, the *Basic Multilingual Plane* (abbreviatory BMP), as are most symbols relevant in mathematics. A subset of Plane 0 is a designated “Private Use Area” whose code points may be used for application-specific purposes – and indeed, UnicodeMathML makes use of them, as described in Section 3.2.2.
- Plane 1, the *Supplementary Multilingual Plane* (SMP), contains the rest of Unicode's presently assigned code points[4] – including the “Mathematical Alphanumeric Symbols” block, which encodes variations of Latin and Greek letters and Arabic numerals[5] replicating common font styles, such as bold, italic, script, Fraktur, sans-serif, monospace, and combinations thereof[6].

---

[1] See `https://en.wikipedia.org/wiki/Mojibake`.

[2] In fact, when looking at a chart of all assigned Unicode code points, it appears to be dominated by Chinese and Korean characters.

[3] As a side note, this “variety of scripts, directions, and input methods [...] impose[s] tricky (and in some cases, unsolved) problems on any [text] editor”: `https://lord.io/blog/2019/text-editing-hates-you-too/`

[4] Except for a set of so-called *CJK Ideographs* in Plane 2 and, some control characters in Plane 14 and a large *Private Use Area* encompassing Planes 15 and 16, none of which are relevant in the context of this thesis.

[5] See `https://en.wikipedia.org/wiki/Mathematical_Alphanumeric_Symbols`.

[6] Interestingly, specific variations of some characters are missing in this block. This is because they had been previously encoded in the BMP, and the Unicode consortium avoids encoding the same symbol twice or breaking backwards compatibility. An example is h, which one would expect to locate at U+1D455, sandwiched between U+1D454 MATHEMATICAL ITALIC SMALL G and U+1D456 MATHEMATICAL ITALIC SMALL I, but is actually encoded as U+210E PLANCK CONSTANT. See also: `https://stackoverflow.com/questions/47206070/why-are-there-holes-in-the-unicode-table`. Similarly, Unicode subscripts and superscripts can be found all over the place, see `https://stackoverflow.com/a/17909597` and `https://en.wikipedia.org/wiki/Unicode_subscripts_and_superscripts`.

Planes 1 through 16 are jokingly related to as *astral* planes[1] since the most commonly used characters are located in the BMP, so the higher planes see limited usage in most contexts. This goes hand in hand with sometimes questionable software support for characters from these planes.

The 17 Unicode planes are visualized in Figure 2.

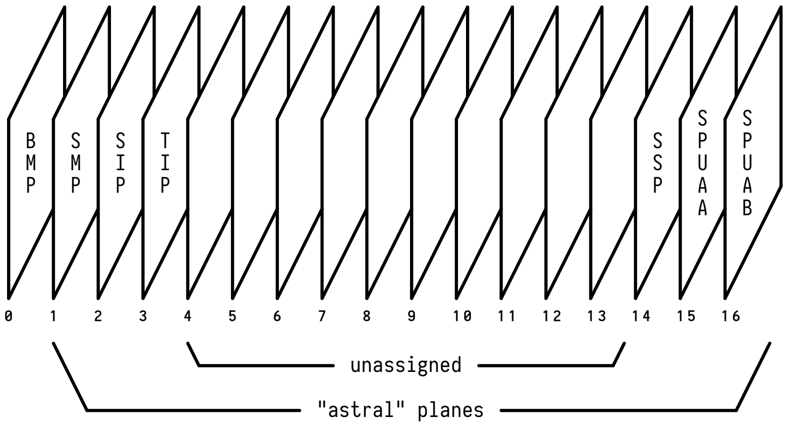


Figure 2: *The 17 Unicode planes.*

Each Unicode character belongs to one of the 29 Unicode character categories[2], for example "Lowercase Letter" (abbreviatory "Lu"), "Modifier Letter" (Lm), "Spacing Mark" (Mc), "Decimal Number" (Nd), or "Math Symbol" (Sm).

There likely is no piece of software that can display all Unicode characters correctly. When a font contains no *glyph* for a specific code point, a replacement symbol such as □ – jokingly referred to as "tofu" because it's a while block[3] – is displayed instead. You can observe this in the abstract of this thesis: U+20D7 COMBINING RIGHT ARROW ABOVE is not supported by the specific font rendering stack used in this instance.

In my work on this thesis, I relied in a variety of tools to navigate the depths and shallows of Unicode's planes, categories, blocks, and code points:

- `http://www.fileformat.info/info/unicode/index.htm` provides pre-rendered example images even for code points not supported by most fonts.
- `https://unicode-search.net` allows its user to search Unicode characters by name – this comes in handy, for example, when looking for different variants of integrals, returning everything from U+222B INTEGRAL to U+2A0C QUADRUPLE INTEGRAL OPERATOR to U+2A17 INTEGRAL WITH LEFTWARDS ARROW WITH HOOK.
- `https://www.compart.com` shows all characters related to a given character.

[1] See `http://opoudjis.net/unicode/unicode_astral.html`.

[2] See `https://www.compart.com/en/unicode/category`.

[3] See `https://threadreaderapp.com/thread/1194628388473819137.html`.

2.2 (LINEAR) MATH ENCODINGS

While simple mathematical formulas (or, in more general terms, expressions) such as Einstein's iconic mass-energy equivalence formula

$$E = \mathrm{mc}^2$$

can more or less be written from left to right, more complicated expressions consist of elements that are on top of other elements, or generally find themselves in a complicated spatial alignment to each other. As an example, consider the product of binomial coefficients[1]:

$$\prod_{k=0}^{n} \binom{n}{k} = \frac{H^2(n)}{(n!)^{n+1}} = \frac{\prod_{h=0}^{n} h^h}{(n!)^{n+1}}$$

Since commonly used text-based formats are to be understood as linear sequences of bytes, there is a need to express mathematical terms in a *linear* fashion.

As related in one of his many interesting blog posts, UnicodeMath's own Murray Sargent developed *SCROLL*, "the first language capable of 'typesetting' mathematical equations on a computer" [Sargent06] in the late 1960s. Around a decade later, "another physicist, Mike Aronson, [...] suggested that the input format should resemble real linearized math as in the C language rather than the Polish prefix format used in SCROLL. So [Sargent] wrote a translator to accept a simplified linear format, the forerunner of the linear format [used] in Office 2007" [Sargent06].

Around the same time, Donald Knuth, dissatisfied[2] with the phototypesetting process used for the second edition of the first volume of his work "The Art of Computer Programming", began developing TeX. On page 27 of the digital typography book, in a transcript of a lecture given in 1978, he compares the linear math notations supported by

- "commercially available systems [then] used to typeset mathematical journals" [Knuth99], where the term θ^2 was encoded as `*gq"2` (where `*g` marks the next character as being Greek and `q`, in this mode, means θ),
- a system [Kerninghan75] "developed at Bell Laboratories [which] has been used to prepare several books and articles" [Knuth99], where the same term was written `theta sup 2`, with
- a primitive version of TeX, which at the time would have denoted the term as `\theta↑2`.

Current versions of TeX and LaTeX instead understand `\theta^2`, of course, as does

[1] See `http://mathworld.wolfram.com/BinomialCoefficient.html`.

[2] See `https://en.wikipedia.org/wiki/TeX#History`. On page 482 of the digital typography book, Knuth's diary entry from that fateful day is reproduced, and the subsequent pages provide insight into the initial design decisions that went into TeX.

AsciiMath[1], a more recent linear format that uses ASCII characters to represent mathematical notation in a deliberately human-readable way. For example, a fraction $\frac{a}{b}$ is represented as `a/b` in AsciiMath (and indeed in UnicodeMath), which is arguably more intuitive than LaTeX's `\frac{a}{b}`.

Stephen Wolfram, when not busy playing[2] with cellular automata, also dabbled[3] in linear formats: "Unlike with ordinary human natural language, it is actually possible to take a very close approximation to familiar mathematical notation, and have a computer systematically understand it. That's one of the big things that we did [around 1995] in the third version of Mathematica. And at least a little of what we learned from doing that actually made its way into the specification of MathML." [Wolfram00]

After this brief history of linear math formats (of which there are many more that I'm unaware of) and before moving on to UnicodeMath, I will briefly introduce a linear format that sets itself apart from all others.

Quoting from Section 4.9 of the tech note (with links converted into footnotes), "[t]he 6-dot Nemeth braille encoding was created by Abraham Nemeth for mathematical and scientific notation. It's general enough to encode almost all of UnicodeMath. He started working on his encoding in 1946 and it was first published in 1952 by the American Printing House for the Blind. As such it's the first math linear format. It's a little like UnicodeMath in that [...] it's a globalized notation, so localization isn't needed except for embedded natural language. Also both formats strive to make simple things easy and concise at the cost of additional syntax rules. But because a mere 64 codes are used to encode virtually all of math notation [...], the semantics of the codes depend heavily on their contexts. This level of complexity contrasts with UnicodeMath which has the luxury of the exhaustive Unicode math symbol set. Accordingly, encoding math expressions can become quite tricky as revealed in the full specification[4]. [...] Nemeth recounts some history in this 1991 interview[5]." [Sargent16]

---

1 See `http://asciimath.org/`.

2 As I, too, am wont to do: `https://twitter.com/sundryautomata`

3 This work led to the addition of five special, double-struck characters to Unicode – see `https://blogs.msdn.microsoft.com/murrays/2010/08/30/linear-format-notations-for-mathematics/` and Section 4.19.

4 See `https://www.nfb.org/images/nfb/documents/pdf/nemeth_1972.pdf`.

5 See `https://www.nfb.org/images/nfb/publications/fr/fr28/fr280110.htm`.

2.3 UNICODEMATH

“Bertrand Russell once wrote, “A good notation has a subtlety and suggestiveness which at times make it seem almost like a live teacher [...] and a perfect notation would be a substitute for thought.” [UnicodeMath] certainly isn't a substitute for thought, but it is a more mathematically natural notation than previously available on computers.”

— Murray Sargent [Sargent10]

A successor to Sargent's previous work on linear math notation, the linear format used within Microsoft's products today is called UnicodeMath. It “uses the Unicode math symbol set [...], resembles real mathematical notation the most closely of all math linear formats, and handles almost every mathematical notation. Since Unicode characters are global by nature, UnicodeMath doesn’t need localization” [Sargent16a] in contrast to formats like LaTeX and AsciiMath, which are based on control words.

For example, the integral

$$\int_0^{20} \sqrt{x}\, dx$$

can be written as

$$∫_0^{20}\ √x\ ⅆx$$

in UnicodeMath whilst a number of control words (along with some manual spacing adjustment) are required in LaTeX:

```
\int_0^{20} \sqrt{x} \ dx
```

A different expression, written as `sin θ=(e^iθ-e^-iθ)/2i` and rendered as shown below, exemplifies how “[o]perators and operator precedence are used to delimit arguments. A binary minus has lower precedence than the superscript operator `^` and the fraction operator `/`, but a unary minus has higher precedence than `^`. This approach contrasts with LaTeX and AsciiMath which require that arguments consisting of more than one element be enclosed in {} or (), respectively.” [Sargent16c]

$$\sin\theta = \frac{e^{i\theta} - e^{-i\theta}}{2i}$$

“A really neat feature of this notation is that the linear text is, in fact, a legitimate mathematical notation in its own right, so it's relatively easy to read.” [Sargent16b] “In addition to being the most readable linear format, UnicodeMath is the most concise. It represents the simple fraction, one half, by the 3 characters `1/2`" [Sargent16a] – in contrast, LaTeX requires 11 characters, and “typical MathML takes 62 characters” [Sargent16a]. “Another advantage of

UnicodeMath over MathML and [a Microsoft Office-specific variant thereof] is that UnicodeMath can be stored anywhere Unicode text is stored. When adding math capabilities to a program, XML formats require redefining the program's file format and potentially destabilizing backward compatibility, while UnicodeMath does not.”

“ *And it's delightful that the operator characters look like the operators they represent, while control words do not.* ”

— Murray Sargent, in Section 3.2 of the tech note [Sargent16]

Refer to table 1 for a set of basic examples where UnicodeMath is both highly concise and more human-readable than LaTeX. UnicodeMath's syntax will be explained in more detail in lockstep with a scenic tour through my UnicodeMath parser in Chapter 4.

Expression	UnicodeMath	LaTeX
$\frac{1}{2}$	`1/2`	`\frac{1}{2}`
$\sqrt{2}$	`√2`	`\sqrt 2`
$\delta_1 \cdot \rho_1$	`δ₁·ρ₁`	`\delta_1 \cdot \rho_1`
$a \neq b$	`a≠b`	`a \neq b`
$\widehat{a+b}$	`(a+b)^`	`\widehat{a+b}`

Table 1: *Different mathematical expressions formulated in UnicodeMath and LaTeX.*

At the time of writing, UnicodeMath version 3.1 is the latest[1] release. A formal specification is not available – instead, Murray Sargent has published a Unicode Technical Note on UnicodeMath, in which he explains the notation in a more colloquial manner, largely based on examples. A simplified grammar is given, however it has not proven to be entirely accurate, as I will discuss in Section 4.1. In the tech note, Sargent also outlines an algorithm for recognizing undelimited mathematical expressions within text and muses about how UnicodeMath could be used to make math-heavy code more readable.

Sargent describes UnicodeMath's syntax in lockstep with how it used in Microsoft's products for equation input, which made it difficult to determine what's actually supposed to be part of the syntax and what's the post-parsing, pre-rendering transformations applied to expressions during “build-up”, *i.e.*, the successive conversion of UnicodeMath into a rendered representation[2] as the user is typing. I've incorporated small bits and pieces that I think are actually related to these input methods into the grammar wherever I was confident that this wouldn't break any otherwise legal syntax.

[1] Note that the changes between version 1 and 3.1 are fairly insignificant, as far as changes from versions 1 to versions 3 go. See Section “Version differences” of the tech note for details.

[2] Sargent himself demonstrates this in a video, see `https://www.youtube.com/watch?v=yyvJwNeUALY` – also, equation build-up and its inverse is patented: `https://patents.google.com/patent/US20060059217`

Two other points worth mentioning before moving on:

- UnicodeMath is designed with an eye on accessibility, which is outlined in a slide deck[1] on the subject.
- Windows provides an API[2] for rendering of UnicodeMath expressions.

2.4 HTML, CSS, JAVASCRIPT, AND BROWSERS

Before continuing with the more interesting stuff, I'll briefly define some jargon and mention several points relating to the standard web stack that both Markdeep and my work are based on.

- When opened in a web browser, an HTML document is transformed into an internal representation that may be queried and modified via JavaScript via a standard interface. This interface is called **DOM**, short for *Document Object Model*. It facilitates traversal of the tree-shaped document structure – nodes represent HTML tags/elements and their attributes, parent-child relationships describe the contents of tags. Integrating UnicodeMath into Markdeep makes use of the DOM in this way.
- DOM manipulations extend to style modifications. The appearance of HTML elements is otherwise defined using CSS[3].
- JavaScript is (in the context of this thesis) solely executed client-side, *i.e.*, within the user's browser. It's worth noting that JavaScript "is executed in a single thread, that is, two [functions] cannot run at same time[, and this] thread also maintains a queue, which has asynchronous tasks queued to be executed one by one [so that a] long running queue task can [block] the execution of all other queue tasks and the main script" [Prusty15]. This means that during a long, uninterrupted computation, no processing of new input or creation of new output can take place. This behavior is one of the reasons why a naïve integration of UnicodeMath into Markdeep (outlined in Section 5.1) didn't cut it.
- JavaScript has not remained static throughout time – its specification, called ECMAScript[4], experiences an update every year. UnicodeMathML, including the playground and the Markdeep integration, makes use of features introduced as recently as in ECMAScript 2017, however the core is written in a more conservative subset of features.
- JavaScript strings are encoded in UTF-16, an encoding based on 16-bit *code units*. Although UTF-16 is capable of encoding all Unicode code points, only the BMP's code points fit into a single code unit. A surrogate pair – the concatenation of two specially reserved code units – is used to encode astral plane characters: 𝒩 (U+1D4A9 MATHEMATICAL SCRIPT CAPITAL N), which is encoded in the SMP, is represented as the surrogate pair U+D835 U+DCA9 in JavaScript strings. This property of the UTF-16 used here complicates many

[1] See `https://www.unicodeconference.org/presentations/S9T1-Sargent.pdf`.

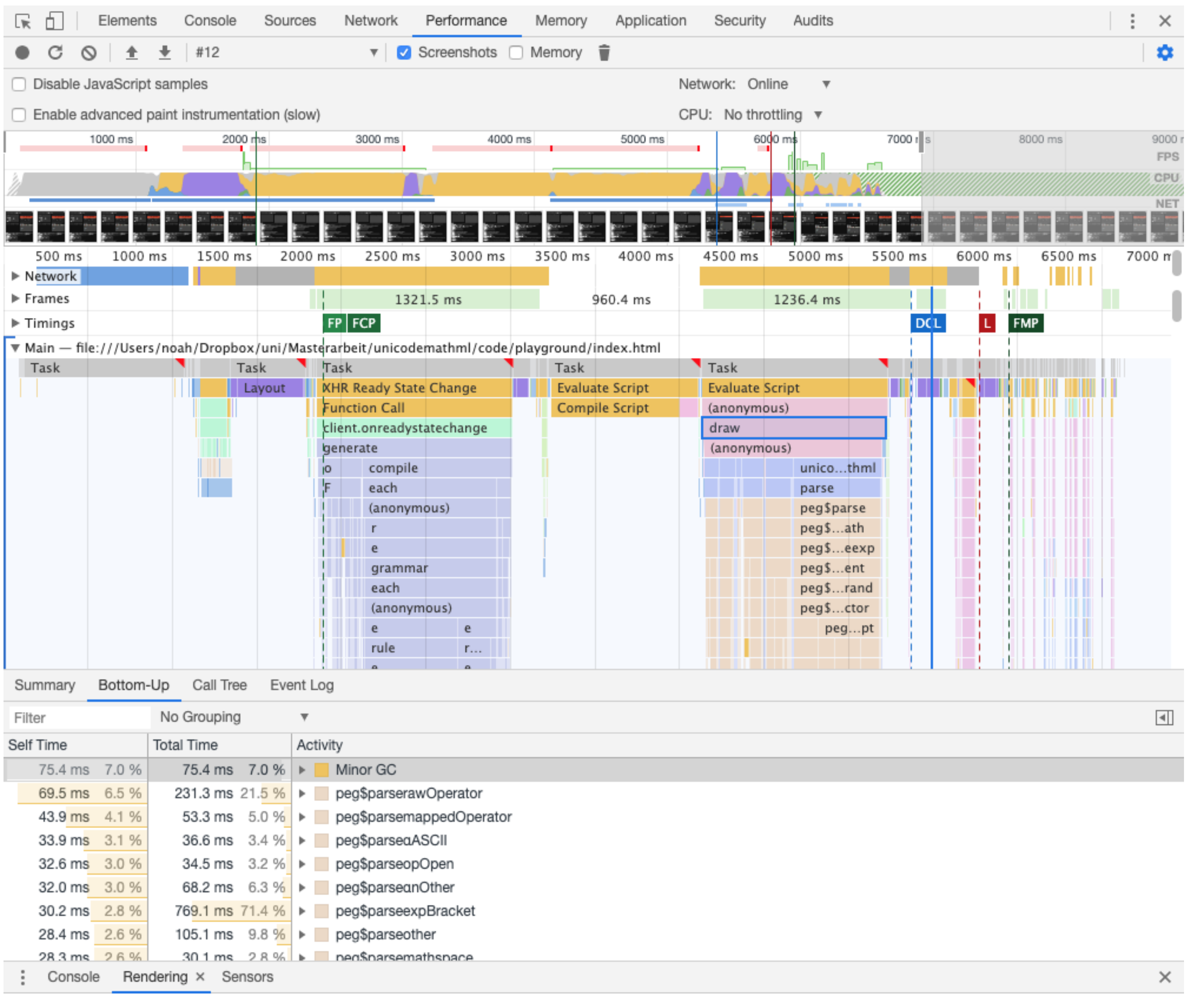
[2] See `https://docs.microsoft.com/en-us/previous-versions/windows/desktop/legacy/hh780445%28v%3Dvs.85%29`.

[3] See `https://developer.mozilla.org/en-US/docs/Web/CSS`.

[4] See `https://en.wikipedia.org/wiki/ECMAScript`.

operations on strings containing such characters, which – combined with the fact that the vast majority of commonly used symbols is encoded in the BMP – is why correct surrogate pair handling is not implemented in many JavaScript libraries.

- Because JavaScript is the primary client-slide programming language on the web, powerful debugging and profiling tools have been built around it. As an example, I will highlight the profiler built into Google Chrome. It can be accessed by opening the developer tools (View > Developer > Developer Tools) and switching the the “Performance” tab. There, a profile can be recorded, either immediately or in conjunction with a page reload. As presented in Figure 3, once a profile has been recorded, a flame graph of function calls and other events is shown. A click on one of these events – here, a call of the `draw` function – reveals which of its component actions have taken up the bulk of its processing time. This tool was tremendously useful in optimizing UnicodeMathML (see Section 7.1.5).



[✓ Screenshots] [Memory]
[Disable JavaScript samples]
[Enable advanced paint instrumentation (slow)]
Network: Online
CPU: No throttling

Main — file:///Users/noah/Dropbox/uni/Masterarbeit/unicodemathml/code/playground/index.html

Self Time		Total Time		Activity
75.4 ms	7.0 %	75.4 ms	7.0 %	Minor GC
69.5 ms	6.5 %	231.3 ms	21.5 %	peg$parserawOperator
43.9 ms	4.1 %	53.3 ms	5.0 %	peg$parsemappedOperator
33.9 ms	3.1 %	36.6 ms	3.4 %	peg$parseαASCII
32.6 ms	3.0 %	34.5 ms	3.2 %	peg$parseopOpen
32.0 ms	3.0 %	68.2 ms	6.3 %	peg$parseαnOther
30.2 ms	2.8 %	769.1 ms	71.4 %	peg$parseexpBracket
28.4 ms	2.6 %	105.1 ms	9.8 %	peg$parseother
28.3 ms	2.6 %	30.1 ms	2.8 %	peg$parsemathspace

Figure 3: *A typical session in Google Chrome's JavaScript profiler. Here, the performance of the UnicodeMath playground's* `draw` *function, which takes a number of UnicodeMath expressions, translates them to MathML, and presents the results to the user, is scrutinized.*

2.5 MATHML

MathML, an XML-based markup language for mathematical expressions, was first released as a W3C recommendation in 1998.[1] Initially supported by major browsers around 2011[2], MathML 3.0 became an ISO standard[3] in 2016.

Two flavors of MathML exist:

- *Content MathML* is designed to provide semantics-preserving encoding. “The intent of the content markup in the Mathematical Markup Language is to provide an explicit encoding of the underlying mathematical structure of an expression, rather than any particular rendering for the expression.” [Carlisle03]
- *Presentation MathML* is instead focused on visual expressiveness, much like other popular math notations. This difference has led to higher adoption of this branch of the standard. “Although Presentation MathML doesn’t have all of the semantics of Content MathML, it does have more semantics than [La]TeX” [Sargent07], which is reflected in UnicodeMath's design.

UnicodeMathML translates UnicodeMath expressions into Presentation MathML for rendering – Content MathML is thus not relevant within the scope of this thesis. All further references to MathML relate to Presentation MathML only.

For example, Einstein's mass-energy equivalence formula can be expressed with the following (annotated) MathML markup.

```
<math xmlns="http://www.w3.org/1998/Math/MathML" display="block">
  <mrow>
    <mi>E</mi>      <!—— identifier ——>
    <mo>=</mo>      <!—— operator ——>
    <mrow>          <!—— grouping, similar to <span> in HTML ——>
      <mi>m</mi>
      <msup>        <!—— subscript ——>
        <mi>c</mi>
        <mn>2</mn>  <!—— number ——>
      </msup>
    </mrow>
  </mrow>
</math>
```

The `<math>` tag's `display="block"` attribute[4] switches MathML renderers into a mode equivalent to LaTeX's `displaystyle`, *i.e.*, with looser constraints on vertical space: for example, fractions may be displayed in a larger font, *n*-ary operators such as integral or sum signs may grow, or the subscript of $\lim_{a\to\infty}$ may move below $\lim$ (that last example is actually

[1] See `https://www.w3.org/1999/07/REC-MathML-19990707/toc.html` – it happened to the be the first W3C standard built upon XML's formal underpinnings.

[2] See `https://developer.mozilla.org/en-US/docs/Web/MathML#Browser_compatibility`.

[3] See `https://www.iso.org/standard/58439.html`.

[4] See `https://www.w3.org/TR/MathML3/chapter3.html#presm.scriptlevel`.

implemented in my translator since some MathML renderers don't support an attribute required for this). `displaystyle` mode is indended for stand-alone expressions. Conversely, `display="inline"` corresponds to `\textstyle` and should be used for expressions occurring within a paragraph of text.

To give you a feel for how MathML tastes, as it were: Other MathML tags include

- `<mfrac>` for fractions,
- `<msubsup>` for combinations of subscripts and superscripts,
- `<mmultiscripts>` for subscripts and/or superscripts shown to the left of their base,
- `<mfenced>` for expressions within delimiters that grow to match their content,
- `<msqrt>` for square roots,
- `<mroot>` for n-th roots, and
- `<mspace>` for various spaces, including line breaks.

Three additional points I'd like to make before moving on:

- The MathML renderers I've tested appear to use a box model akin to TeX's. Page 640 of the digital typography book says: “Each letter is inside a box, and we glue boxes together to make a line, and each line is viewed as a box, and the boxes are fitted together to form a paragraph” [Knuth99] or an equation. The results in artifacts where, for example, in $\sqrt{a} + \sqrt{b}$, the square root symbol surrounding a is shorter than the one around b due to the properties of these letters.
- MathML, due to its structure-encoding properties, is well-suited for making mathematics available to vision-impared people who use screen readers. In a GitHub comment thread on whether the web-based LaTeX renderer KaTeX should drop MathML as an output format, a comment[1] notes that “screen readers do [the following] with presentation MathML: They produce speech; They produce Nemeth code and potentially other braille math codes (there are several used around the world) on an attached refreshable braille display; They allow navigation/exploration of the expression.”
- Murray Sargent notes that “MathML has been designed for machine representation of mathematics and is useful for interchange between mathematical applications as well as for rendering the math in technical documents. While very good for these purposes, MathML is awkward for direct human input. Hence it’s desirable to have more user friendly ways of inputting mathematical expressions and equations” [Sargent07], which this thesis hopefully provides.

2.5.1 BROWSER SUPPORT

As is the case for any client-side web technology, browser adoption is critical to MathML being usable in practice. At the time of writing, browsers with MathML support account for around 22%[2] of the global installed base. Currently, only

[1] See `https://github.com/KaTeX/KaTeX/issues/593#issuecomment-271067131`.

[2] See `https://caniuse.com/#search=mathml`.

- Firefox (including Firefox for Android) and
- Safari (including on iOS)

support a significant subset of MathML natively, with

- an ongoing push[1] by Igalia[2] to implement MathML[3] rendering in Chromium (and thus Google Chrome) and
- Microsoft listing[4] MathML support as being "in development" for their Chromium-based variant of Edge, "not currently planned" for EdgeHTML-based Edge releases and "not supported" in Internet Expolorer.

How well Igalia's initative pans out remains to be seen in the coming months. In spite of the presently poor state of MathML support in major browsers, it is usable in practice thanks to a polyfill[5] provided by MathJax.

2.5.2 MATHJAX

MathJax is a self-described "JavaScript display engine for mathematics that works in all browsers"[6]. It currently accepts three different input formats – LaTeX, AsciiMath and MathML. MathJax can convert either of them into one of the three supported output formats – HTML+CSS, SVG or MathML.

The UnicodeMathML playground, when opened in any browser other than Firefox or Safari, uses MathJax for rendering of translation results. MathJax is also built into Markdeep. `markdeep-thesis`, which this thesis is typeset with (see Section 6.2.1), uses MathJax to transform UnicodeMathML's MathML output into SVG graphics.

MathJax was very useful to me in figuring out how best to express certain mathematical constructs in MathML (or get a "second opinion", at least): I simply needed to formulate them in LaTeX and then configure MathJax to convert this notation to MathML.

I am not aware of any alternatives for rendering MathML on the web in a browser-agnostic fashion. For LaTeX math, however, there are several tools, including jsMath[7] and KaTeX[8], which is referred to as the "fastest math typesetting library for the web" on its website and employs a hand-coded[9] parser.

---

[1] See `https://mathml.igalia.com/`.

[2] Igalia was previously instrumental in improving MathML rendering in Webkit, the HTML rendering engine Safari is built around: `https://webkit.org/blog/6803/improvements-in-mathml-rendering/`

[3] Or rather, what they call MathML Core – "a necessary and rigorous definition of the fundamental subset of MathML features which are widely developed, deployed and used in practice." See Section 8.1 for details.

[4] See `https://developer.microsoft.com/en-us/microsoft-edge/platform/status/mathml/`.

[5] See `https://en.wikipedia.org/wiki/Polyfill`.

[6] See `https://www.mathjax.org/`.

[7] See `http://www.math.union.edu/~dpvc/jsMath/`.

[8] See `https://katex.org/`.

[9] See `https://github.com/KaTeX/KaTeX/blob/master/src/Parser.js`.

Before concluding this section on MathML, it is worth noting that no MathML renderer I'm aware of supports MathML "without asterisks".

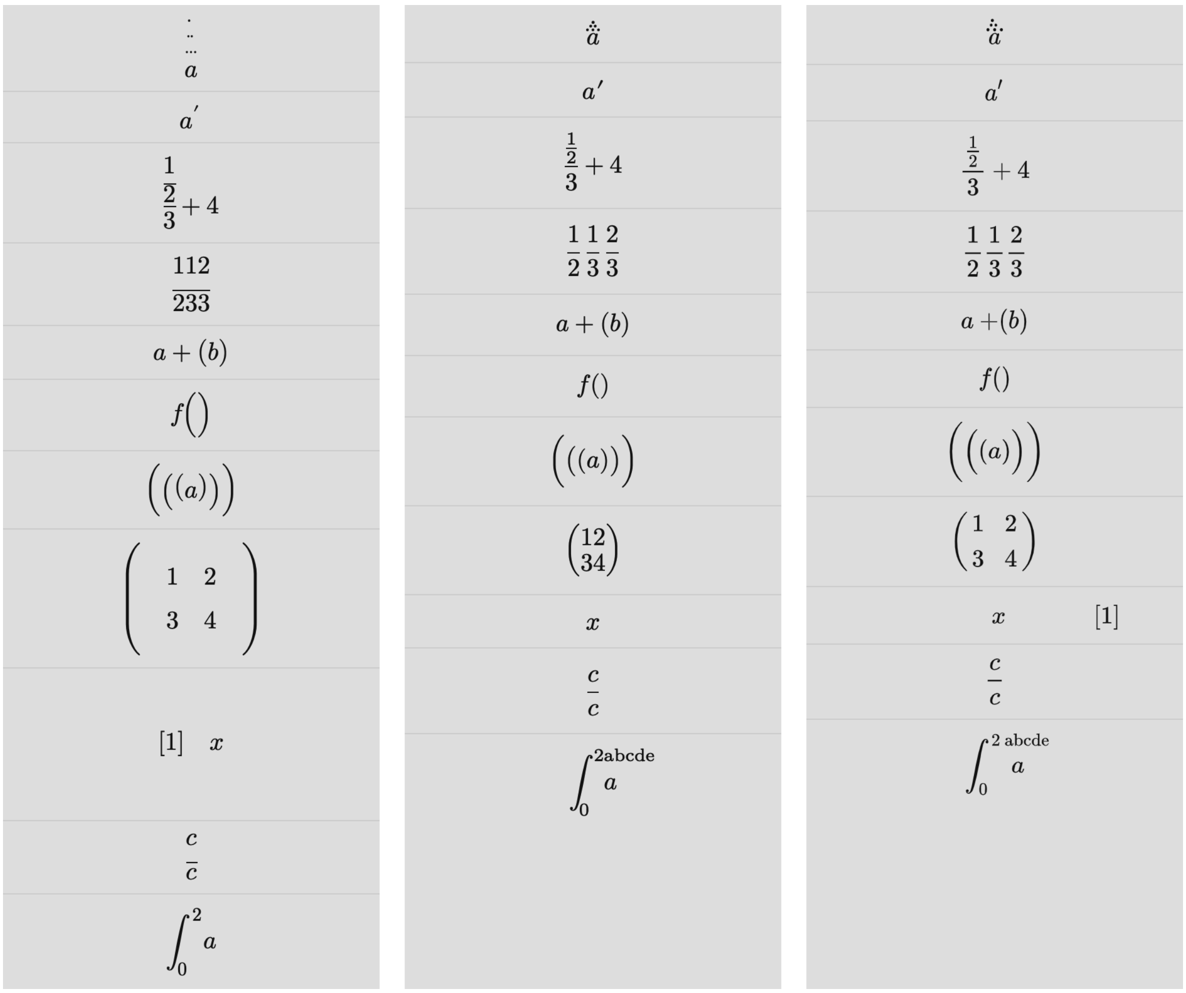


Figure 4: *A number of MathML expressions (translated from UnicodeMath) rendered with (from left to right) Safari 13.0.2, Firefox 69.0, and MathJax 2.7.5 with SVG output in Google Chrome 77.0.3865.90. Note that these are worst-case examples, specifically chosen to display the differences between renderers.*

As shown in Figure 4, different renderers have different quirks. Some MathML elements such as `<merror>` (for error reporting) and `<malignmark>` (for horizontally aligning multiline formulas within equation arrays[1]) appear to be wholly unsupported.

Mozilla has published[2] a "MathML Torture Test" which compares XeTeX-rendered formulas from Donald Knuth's TeXbook with equivalent MathML representations as rendered by the user's browser.

[1] The absence of any renderer support for equation arrays means that UnicodeMath's notation for horizontal alignment of multiple equations (detailed in Section 3.23 of the tech note), as well as equation arrays (Section 3.19), is effectively useless within the context of UnicodeMathML. I have implemented equation arrays anyway, so they might start rendering properly at some point in the future. See Section 7.1.4, as well.

[2] See `https://developer.mozilla.org/en-US/docs/Mozilla/MathML_Project/MathML_Torture_Test`.

Igalia has published an article showcasing the differences across various MathML renderers, concluding that there are currently “three different [native] MathML implementations [in browsers] but work is in process to improve interoperatility *[sic]* between them and fix bugs” [Igalia19].

In my experience, most of the time, MathJax's SVG output format looks better than the what the renderers built into Firefox and Safari produce. MathJax, however, comes with a significant page-load performance penalty, so the UnicodeMathML playground, for example, by default only utilizes MathJax in other browsers.

2.6 MARKDEEP

Basend on John Gruber's Markdown[1], a text markup format “intended to be as easy-to-read and easy-to-write as is feasible” [Gruber04] that “should be publishable as-is, as plain text, without looking like it’s been marked up with tags or formatting instructions” [Gruber-04], Morgan McGuire's JavaScript-based Markdown renderer Markdeep[2] extends this philosophy to diagrams and other elements.

> *Markdeep is a technology for writing plain text documents that will look good in any web browser, whether local or remote. It supports diagrams, calendars, equations, and other features as extensions of Markdown syntax.*
>
> — Morgan McGuire [McGuire19]

I assume that you are basically familar with Markdown's syntax. If this is not the case: GitHub has published a guide at `https://guides.github.com/features/mastering-markdown/`.

Markdown supports rendering of LaTeX, AsciiMath, and MathML equations via MathJax. This thesis aims to expand the range of supported math formats to include UnicodeMath. This integration of UnicodeMathML into Markdeep is discussed in Chapter 5.

In contrast to most other Markdown renderers – which require a server component in order to serve rendered Markdown content on the web – Markdeep is implemented fully client-side: Any Markdown file, when opened in a browser, can be rendered automatically if

1. the `.md` file extension is changed to `.md.html`[3], which convinces browsers to treat the file as an HTML document instead of displaying its contents verbatim, and
2. the following line is included at the bottom of the file:

```
<script src="https://casual-effects.com/markdeep/latest/markdeep.min.js"></script>
```

[1] See `https://daringfireball.net/projects/markdown/`.

[2] See `https://casual-effects.com/markdeep/`.

[3] Just `.html` would work, too – however, since the file contains Markdown instead of HTML, `.md.html` is semantically superior.

This lends itself to modifications of, and tools built upon, Markdeep – indeed, I have modified Markdeep to add UnicodeMath support as described in Chapter 5, and I've built several Markdeep-based tools that I'll briefly outline in Section 6.2.

2.7 PARSER GENERATORS

Early on in the design of UnicodeMathML, the decision was made to use a parser generator instead of implementing a parser by hand.

My expectation was that this would accelerate development (which, in hindsight, is highly debatable – see Section 7.1.1).

Another benefit hinged on the fact that even though Sargent includes a basic UnicodeMath grammar in Appendix A of the tech note, "[t]his grammar is simplified compared to the model in the text" [Sargent16] and does not give an indication of the refinements required to accurately model the abundance of edge cases and syntactical sugar present in UnicodeMath. My thought was that this way, it would be easier to modifiy the parser as the full grammar emerged (in this respect, the plan was solid – there were many changes throughout the process, including significant departures from the basic grammar given in the tech note).

There are many parser generators that emit JavaScript code. Within the scope of this project, I worked with two of them: PEG.js and ANTLR.

2.7.1 PEG.JS

I initially settled on PEG.js[1] due to

- the high performance[2] of the parsers it generates,
- how readable its grammar format is,
- its support for semantic actions that can be used to build up a custom AST during parsing,
- the choice of generating parsers either in the form of a JavaScript object or relatively compact JavaScript source code[3] which does not require a separate runtime,
- its fairly responsive maintainer and community[4] and because it is still being actively developed, and
- because it is written in plain JavaScript, requiring virtually no setup.

As its name suggests, PEG.js generates JavaScript code implementing a *recursive descent*

[1] See `https://pegjs.org/`.

[2] SAP engineers, during development of the parser building toolkit Chevrotain, have built a page that compares the performance of various JavaScript-emitting parsing libraries – PEG.js performs well on this benchmark: `https://sap.github.io/chevrotain/performance/`

[3] As a point of reference: the finished UnicodeMath parser weighs in at around 300 KB, and this is with PEG.js configured to optimize for parsing speed instead of code size.

[4] See `https://github.com/pegjs/pegjs/issues/586`.

parser based on a *parsing expression grammar* (abbreviatory PEG). PEGs differ from the more well-known *context-free grammars* (CFGs) in one important respect, as Bryan Ford's 2004 paper which introduced PEGs notes: Although "[t]he ability of a CFG to express ambiguous syntax is an important and powerful tool for natural languages [...], this power gets in the way when we use CFGs for machine-oriented languages that are intended to be precise and unambiguous. [...] PEGs are stylistically similar to CFGs with [regular expression]-like features added [...]. A key difference is that in place of the unordered choice operator '|' used to indicate alternative expansions for a nonterminal [...], PEGs use a prioritized choice operator '/'. This operator lists alternative patterns to be tested in order, unconditionally using the first successful match. The EBNF rules 'A → a b | a' and 'A → a | a b' are equivalent in a CFG, but the PEG rules 'A ← a b / a' and 'A ← a / a b' are different. The second alternative in the latter PEG rule will never succeed because the first choice is always taken if the input string to be recognized begins with 'a'. A PEG may be viewed as a formal description of a top-down parser." [Ford04] This grammar semantics lends itself to the generation of fast, linear-time parsers.

Importantly, PEG.js supports (and encourages use of) *semantic actions*, code snippets attached to grammar rules that respectively are executed each time the parser recognizes a match for that rule. While the UnicodeMath grammar I wrote uses semantic actions to build up an AST corresponding to the input expressions, semantic actions *can* be used to directly compute an output.

2.7.1.1 EXAMPLE

For example, Listing 1 contains a PEG.js grammar[1] that recognizes simple arithmetic expressions and directly evaluates them via semantic actions.

```
{
  function makeInteger(o) {
    return parseInt(o.join(""), 10);
  }
}

start = expression

expression
  = head:element tail:(("+" / "-") element)* {
      return tail.reduce(function(result, element) {
        if (element[0] === "+") { return result + element[1]; }
        if (element[0] === "-") { return result - element[1]; }
      }, head);
    }

element
  = left:factor "*" right:element { return left * right; }
  / factor

factor
```

[1] Based on a grammar given in the PEG.js documentation: https://pegjs.org/documentation#grammar-syntax-and-semantics

```
22    = integer
23    / "(" expr:expression ")" { return expr; }
24
25  integer = num:[0-9]+ { return makeInteger(num); }
```

Listing 1: *A simple PEG.js grammar that parses and evaluates arithmetic expressions such as `40+2`, `2*3+2*2` or `2*(3+2)*2`. You're encouraged to take it for a spin on `https://pegjs.org/online`.*

Lines 1–5 of Listing 1 are not strictly part of the grammar – instead, PEG.js makes any functions defined in this area callable from semantic actions.

By default, PEG.js begins parsing from the first rule it encounters, which in this case is located in line 7 and appropriately named `start`. No semantic action is attached to this entry rule, so whatever its right-hand side returns is simply returned as the parser's output.

The `expression` rule defined in lines 9–15 exemplifies how grammar rules can be defined in a regex-like[1] fashion: it matches `element`, `element ("+" / "-") element`, `element ("+" / "-") element ("+" / "-") element`, and so on. Top-level components of the *parsing expression*, *i.e.*, the right-hand side, can be named; these names can be referred to in semantic actions. Parentheses may be used to create what amounts to an anonymous grammar rule within a parsing expression, as was done for `(("+" / "-") element)` here.

The `element` rule of lines 17-19 shows that multiple alternative parsing expressions, separated by `/`, may be assigned to a rule. PEG semantics dictate that the generated parser will successively try out these alternatives in order, progressing with the first one that matches. Thus, the *order* of alternatives is crucial – if one were to switch them here, the grammar would cease to recognize multiplicative subexpressions. In other words, “PEG parsers are greedy algorithms. They accept the first sequence [of grammar rules] that matches the input and backtraces *[sic]* if the sequence doesn't reach an accepting state before it hits a mismatch.”[2] This PEG.js behavior is reflected in my UnicodeMath grammar.

Another point worth making is that rewriting this rule to

```
element
  = left:element "*" right:element { return left * right; }
  / factor
```

would lead to an error during parser generation as this rule is now left-recursive[3] – PEG.js detects left recursion, but does include functionality to eliminate it.

The `integer` rule defined in line 25 displays regex-like character range matching, which is

[1] Note that “PEG does not backtrack through [these] repetition operators” when matches within them fail, as noted on `https://groups.google.com/forum/#!topic/pegjs/maCYLZG_gCk`. “The expression ‘a* a’ for example can never match any string.” [Ford04]

[2] See `https://groups.google.com/forum/#!topic/pegjs/i1jsGNbyggk`.

[3] See `https://en.wikipedia.org/wiki/Parsing_expression_grammar#Indirect_left_recursion`.

significantly more efficient than `0 / 1 / ⋯ / 9` would be. Also, the previously-defined function `makeInteger` is called in this rule's semantic action.

Operator precedence is encoded within the structure of the grammar – PEG.js does not provide syntax for defining it any other way. This was a minor issue with regard to replicating UnicodeMath's semantics in UnicodeMathML's parser, but I managed to solve it without too much trouble. Note that Listing 1 demonstrates most[1] of the PEG.js features used in the UnicodeMath grammar, which is located at `/code/src/unicodemathml.pegjs`.

2.7.1.2 CONFIGURATION

PEG.js allows the programmer to set a number of options once it's time to build a parser from a grammar:

- The `output` format – by default, PEG.js returns a parser in the form of a JavaScript object, but generating a static copy of the parser that take the form of JavaScript source code is supported, as well.
- Whether the parser should `cache` already-successfully-parsed subexpressions as it tries out different alternatives instead of starting anew every time, "avoiding exponential parsing time in pathological cases but making the parser slower", according[2] to the PEG.js documentation. However, for my UnicodeMath grammar (even after applying a number of manual optimizations), this makes things about 20x faster on average than sans caching (this is further discussed in Section 7.1.5).
- Whether to include the bits and pieces required to `trace` the parser's progress – if enabled, the parser keeps a log as it enters and exits grammar rules, which can be helpful when debugging the grammar. This is somewhat akin to compiling C++ code with debug symbols.
- More options (that are less relevant in a UnicodeMathML context) are explained[3] in PEG.js's documentation.

[1] In addition to the `*` and `+` quantifiers, PEG.js implements a zero-or-one quantifier `?`. What's more, PEG.js features a lookahead mechanism which came in handy in some places: `! expression` performs negative lookahead (it matches if `expression` does not match and does not consume any input), and `& expression` analogously implements positive lookahead.

[2] See `https://pegjs.org/documentation`.

[3] See `https://pegjs.org/documentation`.

2.7.2 ANTLR

After I ran into Unicode-related trouble with PEG.js (see Section 7.1.1), I experimented with ANTLR[1] ("ANother Tool for Language Recognition"), a widely-used[2] parser generator that

- is written in Java,
- has been around for a long time[3],
- can generate parsers in various languages[4] (these *targets* include Java, Python, JavaScript, C++, and Swift, among others), and
- is generally known to generate fast[5] parsers.

Parsers generated by ANTLR 4, the most recent generation, make use of the ALL(∗) (pronounced "allstar", short for *Adaptive LL(∗)*) parsing algorithm. ALL(∗) parsers "combine the simplicity of deterministic top-down parsers with the power of a GLR-like mechanism to make parsing decisions. Specifically, LL parsing suspends at each prediction decision point (nonterminal) and then resumes once the prediction mechanism has chosen the appropriate production to expand. The critical innovation is to move grammar analysis to parse-time; no static grammar analysis is needed [and] ALL(∗) is $O(n^4)$ in theory but consistently performs linearly on grammars used in practice" [Parr14], according to the paper introducing the ALL(∗) algorithm.

Parsers generated by ANTLR 4's JavaScript target require a runtime[6] in order to be executable in a browser, which makes them larger than PEG.js-generated parsers. Further, Webpack[7] must[8] be used to integrate runtime and parser.

Combined with performance issues I experienced after implementing a subset of the UnicodeMath grammar in ANTLR (located in ☁ `/antlr-experiment/`, refer to ☁ `/antlr-experiment/README.md` for help with setting things up), this convinced me to switch back to PEG.js. This decision and what led to it is explored in more detail in Section 7.1.1.

1 See `https://www.antlr.org/`.

2 See `https://en.wikipedia.org/wiki/ANTLR#Projects`.

3 The initial commit to `https://github.com/pegjs/pegjs` was performed on March 7, 2010, but ANTLR was released in February 1992.

4 See `https://github.com/antlr/antlr4/blob/master/doc/targets.md`.

5 In SAP's previously mentioned benchmark of JavaScript-emitting parsing libraries, the ANTLR-generated parser's performance is not far behind its PEG.js equivalent.

6 See `https://www.npmjs.com/package/antlr4`.

7 See `https://webpack.js.org/`.

8 See `https://github.com/antlr/antlr4/blob/master/doc/javascript-target.md`.

2.8 PRIOR AND RELATED WORK

Much related work has already been discussed in the previous sections:

- AsciiMath's syntax is similar to UnicodeMath's. Its canonical implementation, developed in conjunction with the language itself, uses a hand-coded parser[1] to translate AsciiMath into MathML.
- A variant of UnicodeMath ships as part of Microsoft's office suite, where it is referred to as “linear format”. This implementation differs from what's specified in the tech note in minor respects, but was useful in clearing up a number of ambiguities.

Several folks have implemented parsers for subsets of UnicodeMath, but none of the ones I could find seem to be more than half-baked:

- `https://github.com/appcypher/unicode-math` (stalled at an early stage)
- `https://github.com/jipsen/UnicodeMath` (narrow subset, handwritten parser)

Other things that may be of interest:

- The Sublime Text plugin “UnicodeMath”[2], which allows input of UnicodeMath expressions. Combined with UnicodeMathML's Markdeep integration, this makes UnicodeMath-enabled document authoring a breeze.
- The XeTeX and LuaTeX package “unicode-math”[3] allows the use of Unicode symbols in LaTeX documents – simple UnicodeMath expressions can be used in LaTeX this way.
- The Java application “MathToWeb”[4] translates LaTeX math into MathML.

[1] See `https://github.com/asciimath/asciimathml/blob/master/ASCIIMathML.js`.

[2] See `https://github.com/mvoidex/UnicodeMath`.

[3] See `https://ctan.org/pkg/unicode-math`.

[4] See `http://www.mathtoweb.com/cgi-bin/mathtoweb_home.pl`.

3

IMPLEMENTATION ARCHITECTURE

This chapter is designed to remove complexity from the next. Put differently: I will now discuss some details of the UnicodeMathML pipeline – the set of transformations applied to a mathematical expression on its path through UnicodeMathML – with the goal of later focusing on UnicodeMath parsing and its translation to MathML *only*, blissfully ignoring any non-core aspects. Recall Figure 1 from the introduction for a high-level visual overview of the pipeline.

3.1 UNICODEMATHML.PEGJS

The UnicodeMath grammar (explained in Section 4.1) is kept in `/code/src/unicodemathml.js`. I've previously discussed PEG.js's grammar syntax, however I'd like to present another example to exemplify how the UnicodeMath AST is built up during parsing – it's a standard JavaScript object:

```
// terms enclosed in rectangles, circles, etc.
enclosed
    = "▭(" m:bitmask "&" o:exp ")" {
        return {enclosed: {mask: m, symbol: null, of: o}};
    }
    / e:opEnclosure o:operand {
        return {enclosed: {mask: null, symbol: e, of: o}};
    }
```

(Enclosures will be discussed in Section 4.10)

When using the UnicodeMathML playground (more about it in Section 6.1), PEG.js automatically converts the grammar into an in-memory parser as the playground loads.

```
var grammar = '<grammar source="" code,="" fetched="" using="" an="" xmlhttprequest="">';
ummlParser = peg.generate(grammar, {cache: ummlConfig.caching, trace: ummlConfig.tracing});
```

The playground allows the user to toggle caching and/or tracing (see Section 2.7.1) on or off.

Alternatively, opening /code/utils/generate-parser.html in any browser generates a static parser (with caching enabled) that will show up wherever that browser stores downloads. It takes the form of a reasonably lightweight and transparently-structured[1] JavaScript file: Even ahead of possible postprocessing steps such as minimization and/or gzipping, it weighs around 315 KB. When included into an HTML document, it provides a function `ummlParser.parse` which – as the name suggests – takes a UnicodeMath expression, parses it and returns the resulting UnicodeMath AST.

3.2 UNICODEMATHML.JS

The `ummlParser.parse` function is called from within /code/src/unicodemathml.js, which does the rest of the work, most notably containing the implementation of the transformation step. It provides the `unicodemathml` function which funnels an input expression through the entire UnicodeMathML pipeline, returning the resulting MathML code along with intermediate data structures such as the UnicodeMath AST, a variant of the UnicodeMath AST that's been preprocessed ahead of the transformation step, and the MathML AST.

Before the UnicodeMath AST is available, three steps are performed in conjunction with parsing.

3.2.1 CONTROL WORD SUBSTITUTION

If the variable `ummlConfig.resolveControlWords` is both set and `true`, any *control words* (see Section 4.6 and Appendix B of the tech note) found in the input expression are replaced with their corresponding Unicode character. This is done via a basic lookup in the dictionary

```
var controlWords = {
    '\\above': '2534',
    '\\acute': '0301',
    '\\aleph': '2135',
    …
}
```

[1] See https://github.com/pegjs/pegjs/issues/620.

which maps backslash-preceded control words (in JavaScript strings, backslashes must be escaped with backslashes) to Unicode code points. This step is optional since it's firmly planted in input method territory – it's enabled by default in the playground, but disabled by default in the Markdeep and HTML integrations.

In my implementation, control words in the input are expected to be terminated by a non-alphabetic character or a space (which is removed during control word substitution).

3.2.2 ASTRAL MAPPING

As I will explore in Section 7.1.1, PEG.js has trouble with Unicode characters that lie outside of the BMP, *i.e.*, *astral* code points. The vast majority of symbols relevant for math are not affected by this, however the *Mathematical Alphanumeric Symbols* block, which contains various variants of Latin and Greek letters, is part of SMP, as are some emoji.

To get around this limitation, the `astralPrivateMap` variable defines a bijective mapping between

- the *Mathematical Alphanumeric Symbols* block and
- a number of code point ranges[1] corresponding to SMP-based emoji

and the BMP's *Private Use Area*. Here, code points are specified as hexadecimal numbers.

```
var astralPrivateMap = [
    {astral: {begin: 0x1D400, end: 0x1D7FF}, private: {begin: 0xE000, end: 0xE3FF}},
    {astral: {begin: 0x1F004, end: 0x1F004}, private: {begin: 0xE400, end: 0xE400}},
    {astral: {begin: 0x1F0CF, end: 0x1F0CF}, private: {begin: 0xE401, end: 0xE401}},
    …
    {astral: {begin: 0x1FA90, end: 0x1FA95}, private: {begin: 0xE803, end: 0xE808}}
];
```

Between control word substitution and parsing, the function `mapToPrivate` receives the input and replaces any characters[2] which fall into the range of any of the `astralPrivateMap` entries with the corresponding *Private Use Area* code point.

3.2.3 PARSING

After control word substitution and astral mapping has been applied to the input, the UnicodeMath parser is let loose on it via a call of `ummlParser.parse`. I do believe that it is sufficiently explained in the next chapter, so I won't get into any details here.

Note that the grammar rules for emoji and the the *Mathematical Alphanumeric Symbols* block contain the *Private Use Area* ranges assigned during astral mapping.

[1] Determined by a Python script, see Section 6.3.

[2] There are many way to split a JavaScript string `s` into characters. Conventionally, a programmer would likely write `s.split("")`, however this method is unaware of the meaning of surrogate pairs (described in Section 2.4), splitting a string into 16-bit substrings. Instead, `Array.from(s)` is Unicode-aware and thus functions correctly here.

If the parser throws an error, an appropriate error message is immediately returned[1] to the user.

3.2.3.1 TRACING

If tracing, described in Section 2.7.1, has been enabled[2] by the user, the `ummlParser.parse` function is called with an additional argument: a reference to a *tracer*. This is an object exposing a `trace` function which the parser, whenever a grammar rule is entered, a match succeeds, or a match fails, calls with an appropriate event object.

I've implemented a basic[3] tracer, called `SimpleTracer`, which simply keeps a log of these events. It is equipped with a function `traceLogHTML` which can format the log as HTML code where matches and failures are highlighted – this is used to present traces in the UnicodeMathML playground.

3.2.3.2 INVERSE ASTRAL MAPPING

If parsing was successful, the inverse of the previously discussed astral character replacement function is mapped over the AST. It really pays that the AST is simply a JavaScript object solely composed of lists, strings (as well as numbers, `null`s and such – but they're not relevant here) and further objects with the same constraints – the AST mapping function `astMapFromPrivate` is only a few lines long.

Inverse astral mapping is also applied to any error messages produced during parsing.

3.2.4 PREPROCESSING

I've been prototyping LaTeX code generation (more in Section 8.1) shortly before submission of this thesis, which has prompted me to separate a number of desugarings and minor AST transformations which are required no matter whether MathML, LaTeX or any other output format is desired, from the main translation step.

This preprocessing step is mapped over the UnicodeMath AST before the actual translation commences. It has been implemented in the function `preprocess` and structured analogously to the upcoming translation step.

In the next chapter, I will treat actions performed during this preprocessing step as being part of the translation.

---

[1] Note that both the original UnicodeMath expression as well as the variant where control words have been substituted are returned if they are not equal, in case the user has accidentally used the wrong control word somewhere. (It also allows the user to replace all control words in an expression with their meanings: append a slash to the expression, which yields a syntax error, and copy the thusly Unicode-fied expression from the error message.)

[2] This is indicated by the `ummlConfig.tracing` boolean – it can be set via the UnicodeMathML playground's interface.

[3] Fancier tracers exist: `https://github.com/okaxaki/pegjs-backtrace`

3.2.5 TRANSLATING

The function `mtransform` is responsible for transforming the UnicodeMath AST into a MathML AST (see the next section) and thus does most of the work translating UnicodeMath to MathML. Since the UnicodeMath AST is a standard JavaScript object, this function function can be written as sketched below:

```
1  function mtransform(dsty, puast) {
2      if (Array.isArray(puast)) {
3          return {mrow: noAttr(puast.map(e => mtransform(dsty, e)))};
4      }
5
6      var key = Object.keys(ast)[0];
7      var value = Object.values(ast)[0];
8
9      switch (key) {
10         ...
11         case "expr":
12             return mtransform(dsty, value);
13         case "operator":
14             return {mo: noAttr(value)};
15         case "negatedoperator":
16             return {mo: noAttr(value + "/")};
17         ...
18     }
19 }
```

If the input AST takes the form of a list (which it can, for example, in the recursive call of line 12), Lines 2-4 recursively call `mtransform` on each list element. The results are emitted in an `mrow` node with `noAttributes` in order to group them, since a MathML element further up the call stack may expect a fixed number of child nodes.

The input variable `dsty` is short for `displaystyle` – as you may recall from Section 2.5, MathML renderers make more liberal use of vertical space when rendering `displaystyle` expression. Not all of them support it sufficiently, though, so the translation step does a bit of extra work for some constructs depending on the truthiness of `dsty`.

Line 16 exemplifies how negated operators[1] are translated into MathML: a U+0338 COMBINING LONG SOLIDUS OVERLAY is appended to the operator symbol, it will be superimposed over it during rendering. The resulting string is returned as an `mo` AST node corresponding to MathML's `<mo>` element. (Of course, not all transformations are this simple.)

[1] During preprocessing, any `negatedoperators` for which a dedicated Unicode symbol exist have been replaced with an `operator` with the corresponding negated symbol.

3.2.6 PRETTY-PRINTING

The MathML AST directly corresponds to MathML code. Rather than explaining it, I will show a slightly condensed example...

```
{"math": {
  "attributes": {
    "class": "unicodemath",
    "xmlns": "http://www.w3.org/1998/Math/MathML",
    "display": "block"},
  "content": {
    "mrow": {
      "attributes": {}, "content": [
        {"mrow": {
          "attributes": {}, "content": {
            "mrow": {
              "attributes": {}, "content": {
                "mi": {"attributes": {}, "content": "a"}}}}}},
        {"mo": {"attributes": {}, "content": "+"}},
        {"mrow": {
          "attributes": {}, "content": {
            "mrow": {
              "attributes": {}, "content": {
                "mi": {"attributes": {}, "content": "b"}}}}}}]}}}}
```

...and the MathML code generated from it:

```
<math class="unicodemath" xmlns="http://www.w3.org/1998/Math/MathML" display="block">
  <mrow>
    <mi>a</mi>
    <mo>+</mo>
    <mi>b</mi>
  </mrow>
</math>
```

Notice that the MathML code contains fewer `<mrow>` tags than the MathML AST. since `<mrow>` is predominantly used for grouping, similar to curly braces `{}` in LaTeX: in `a^{b}`, the braces can be omitted because they contain only a single character. In the same manner, `<mrow>` tags that contain only a single child are superfluous, which is why my pretty-printer strips them out.

3.3 NOTES ON MODULARIZATION

Towards the end of the thesis period, I considered splitting `/code/src/unicodemathml.js` into a number of files, each of which would be responsible for a portion of the pipeline (*e.g.*, one file for parsing-related functionality, another for preprocessing, etc.). This could have been done

- naïvely, by essentially moving each step into its own file and requiring the user to import all files into an HTML document before being able to use UnicodeMathML, or
- in a more sophisticated manner by using the

module system[1] introduced in ECMAScript 6. While fairly new, this feature is widely supported. Modules must be defined in `.mjs` files, any functions defined within them may be made available to other modules using by prepending `export` to the functions' signatures or via a single `export` statement:[2]

```
…
export {a, b, c};
```

Importing functions from a different module works as follows:

```
import {d, e, f} from './modules/def.mjs';
```

Such modules can be loaded (or defined) using a normal `&script>` tag in HTML:

```
<script type="module" src="./modules/main.mjs"></script>
<script type="module">
  …
</script>
```

Splitting UnicodeMathML into modules would obviously be the proper way of doing things, and it would allow leaving out parts of the pipeline as desired by the user (*e.g.*, the LaTeX code generation I prototyped, see Section 8.1). However, I eventually decided against it for the following reasons:

- `unicodemathml.js` is not that large – before minimization and/or gzipping, it weighs only around 100 KB. Thus, any loading speed gains stemming from leaving any presently-unneeded parts of the pipeline out would be negligible. What's more, the network overhead from loading n modules[3] instead of one file would exceed any savings.
- ES6 modules do not work when served from the local file system in most browsers, so one would need to spin up a local server (described at the top of Section 6.1), which would make certain use cases more cumbersome. Further, modules are cached by most browsers, and any changes to them might only take effect after telling the browser from the server or emptying the browser's cache, which increases development complexity.
- Keeping everything in one file makes changes and basic refactorings easier.

Another idea: It's possible to "hide" all functions except for `unicodemathml` as shown below – this is how parsers generated by PEG.js avoid polluting the global namespace. This makes various debugging tasks a bit more cumbersome, though, so I decided against it for now. I will likely do it once I release UnicodeMathML as free software (see Chapter 8), though.

[1] See `https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Modules`.

[2] More elaborate examples can be found here: `https://github.com/mdn/js-examples/tree/master/modules`

[3] Especially since modules referenced not from the main document, but from another module, can only begin loading once their parent modules have partially been loaded – browsers simply don't know about them before.

```
(function(root) {
    …
    root.unicodemathml = unicodemathml;
})(window);
```

Note that I've kept the code such that it could be adjusted to match either of the approaches discussed in this section without much trouble.

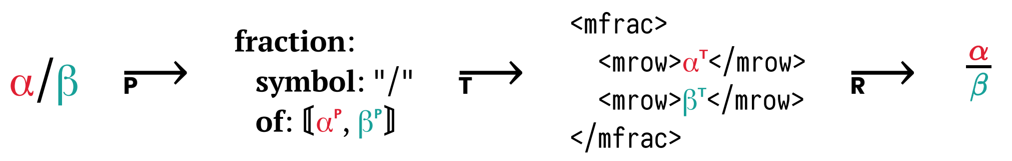
With the architectural details now out of the way, let's focus on the interesting stuff: parsing and translating.

4 PARSING UNICODEMATH AND TRANSLATING IT INTO MATHML

In this chapter, I will first present a simplified variant of my PEG.js-based UnicodeMath grammar. Then, I will take you on a stroll though UnicodeMaths's constructs, explain their syntax and how they are transformed into a UnicodeMath AST during parsing, then translated into a MathML AST, and finally pretty-printed as MathML code, ready for rendering.

In the context of this thesis, implementing a complete grammar for UnicodeMath that generates a fast parser presented the greatest difficulty to me, the reasons for which I will detail in Section 7.1.5 – translating it into MathML was the easy part, since UnicodeMath maps to MathML relatively cleanly.

As I've already alluded to in the “Notes” section before the table of contents, I will be using a bespoke notation for various transformations. The example atop the next page documents how a basic UnicodeMath fraction is parsed into a UnicodeMath AST.



$\alpha/\beta \xrightarrow[\mathbf{P}]{}$

fraction:
  symbol: "/"
  of: ⟦α^{P}, β^{P}⟧

$\xrightarrow[\mathbf{T}]{}$

```
<mfrac>
  <mrow>αᵀ</mrow>
  <mrow>βᵀ</mrow>
</mfrac>
```

$\xrightarrow[\mathbf{R}]{} \frac{\alpha}{\beta}$

Here, α and β can be any substrings that the grammar permits in the numerator and denominator positions of a fraction.

The letter **P** below the arrow indicates that parsing is the step performed next. Accordingly, α^{P} (read this as "alpha parsed") and β^{P} are the UnicodeMath AST subtrees that result from parsing the previously named substrings, assuming no error has occurred (any error stops the parse in its tracks). The AST is shown in a compact format which makes use of indentation to indicate nesting in lieu of braces. ⟦⋯⟧ denotes a list (*i.e.*, a JavaScript array) – note that lists split across multiple lines use line breaks instead of commas as item separators. JavaScript `null` values are shown as ␀ (U+2400 SYMBOL FOR NULL).

Next, the **T** arrow indicates the translation[1] and pretty-printing steps. The resulting MathML code (shown without the enclosing `<math>` tag to save space) again contains appropriately transformed variants α^{T} and β^{T} of the named subexpressions.

Finally, a rendered representation of the MathML code is presented towards the right of the **R** arrow. For better correspondence to the input, named subexpressions are shown as α instead of α^{R}.

4.1 GRAMMAR

The notation and level of detail of the following simplified and heavily commented grammar is roughly based[2] on the simplified grammar given in Appendix A of the tech note. In some places, natural language is used to describe rules that are conceptually simple, but awkward to read when expressed with formal grammar syntax. Some edge cases, as well as markers for where arbitrary whitespace[3] is allowed, have been omitted. Note that operator precedence (see Section 3.22 of the tech note) is implicitly encoded within the grammar.

Semantic actions are not included here – how they construct an AST based on the UnicodeMath syntax encoded in this grammar is explained in the following sections, as previously noted. Accordingly, you might prefer to skim this grammar for now, referring to it as required while reading on.

[1] Note that the only *context* during transforming is the `displaystyle` variable, which is not shown in these diagrams to avoid visual clutter.

[2] Although the ordering is different – in the tech note's grammar, the entry rule is found at the bottom, while the grammar shown here *begins* with it. (This is simply because I found myself mentally traversing Sargent's grammar from bottom to top more often than not.)

[3] My grammar allows whitespace wherever possible, *i.e.*, wherever one might be inclined to insert white space to aid readability of expressions in plain text. Note that this differs from how whitespace is treated in Section 2.3 of the tech note, but that description is more related to equation build-up as implemented in Microsoft's programs.

Note that the full, unabridged grammar can be found at ☁/`code/src/unicodemathml.pegjs`.

Entry rule (discussed after this grammar)

unicodemath	←	(**exp** "\n"?)+ **eqnumber**?
eqnumber	←	"#" followed by arbitrary characters, until a line break

exp	←	(**element** / **operator**)+ ↪ *The tech note specifies this rule as "element (operator element)∗", but in my testing, this had no advantages, but it disallows some reasonable expressions such as* `-a+b`.

Operators (discussed in Section 4.2).

operator	←	**mathspaces**
	/	**basicoperator** with any kind of script (see further down) attached
	/	**basicoperator**
basicoperator	←	**negatedoperator** / **rawoperator**
negatedoperator	←	**rawoperator** preceded by a slash or followed by U+0338 COMBINING LONG SOLIDUS OVERLAY ↪ *In spite of not being mentioned in the tech note, negating operators by attaching the diacritical mark* U+0338 COMBINING LONG SOLIDUS OVERLAY *looks better in plain text: compare* /+ *with* +̸.
rawoperator	←	any character that's not recognized by the "atom" rule or used as a special operator in any of the other rules

Constructs built around low-precedence operators – they are likely to be near the outermost level of an expression imagined in tree form, and are thus – and because the greediness of PEG parsers requires this – the object of one of the first parsing decisions.

element	←	**array** / **matrix** / **nary** / **phantom** / **smash** / **fraction** / **atop** / **operand**

Equation arrays and matrices (discussed in Section 4.3).

array	←	"█(" **arows** ")"
arows	←	**arow** ("@" **arow**)∗
arow	←	**exp**? ("&" **exp**)∗

matrix	←	"■(" **mrows** ")"
	/	"⒨(" **mrows** ")" ⇝ *syntactic sugar for parenthesized matrices*
mrows	←	**mrow** ("@" **mrow**)∗
mrow	←	**exp** ("&" **exp**)∗

n-*ary operations such as sums of integrals (discussed in Section 4.4). The bit mask may be used to modify the layout of the integration limits. "▒" links the integral to its integrand.*

nary	←	**opNary bitmask**? (**script** / **abscript**)? "▒" **element**
opNary	←	[∑∏∫∬∭⨌∮⋯]
bitmask	←	[0-9]+

↪ *Bit masks are used in several places throughout the grammar, each time with different meanings. The tech note mentions that a graphical user interface should provide visual tools to generate these bit masks.*

Phantoms and smashes (discussed in Section 4.5).

phantom ← "⟡(" **bitmask** "&" **exp** ")"
/ **opPhantom** "(" **exp** ")"

opPhantom ← "⟡" ⇝ *standard phantom*
/ "⬄" ⇝ *horizonal phantom, like LaTeX's* `\hphantom`
/ "⇳" ⇝ *vertical phantom, like* `\vphantom`

smash ← **opSmash** "(" **exp** ")"

opSmash ← "⬍" ⇝ *standard smash*
/ "⬆" ⇝ *ascenders smash*
/ "⬇" ⇝ *descenders smash*
/ "⬌" ⇝ *horizontal smash*

Various kinds of fractions (discussed in Section 4.6).

fraction ← [↉½⅓⅔¼¾⅕⅖⅗⅘⅙⅚⅐⅛⅜⅝⅞⅑]
↪ *Note that this is non-standard – the tech note does not mention Unicode's built-in vulgar fractions, but I think it's neat to support these as syntactical sugar.*
/ (**operand opFraction**)+ **operand**

opFraction ← "/" ⇝ *yields a normal fraction*
/ U+2044 FRACTION SLASH ⇝ *skewed fraction,* e.g., $^1/_2$
/ U+2215 DIVISION SLASH ⇝ *linear fraction,* e.g., *1/2*
/ U+2298 CIRCLED DIVISION SLASH ⇝ *small numeric fraction*

atop ← (**operand** "¦")+ **operand** ⇝ e.g., $\genfrac{}{}{0pt}{}{1}{2}$
/ **operand** "⒞" **operand**
↪ *Syntactic sugar for binomial coefficients.*

Medium-precedence constructs, comprising constructs which may occur inside *scripts (and outside of any script context, of course) as well as the three kinds of scripts themselves.*

operand ← **factor**+

factor ← **preScript**
/ **subsupScript**
/ **abovebelowScript**
/ **sfactor**
↪ *Covers various other constructs that can also occur within script exponents (and outside of any script context, of course).*
/ **entity**

*Subscripts, superscripts and combinations thereof (discussed in Section 4.7). These are rather complicated (or, rather, convoluted) because they are composed of 1. LaTeX-style scripts and 2. scripts made up from subscripted and/or superscripted Unicode characters. These two types may be combined under certain circumstances. Additionally, some of the rules in this block are reused for prescripts and *n*-ary operations.*

subsupScript	←	**subsupSubsup**
	/	**subsupSubscript**
	/	**subsupSuperscript**
script	←	**subsup / sub / sup** ⇝ *subscripts and superscripts without a base*
scriptU	←	**subsupU / subU / supU** ⇝ *base-less Unicode scripts only*
subsupSubsup	←	**scriptbase subsup**
subsup	←	**subU** (**supU / supL**)
	/	**supU** (**subU / subL**)
	/	**subsupL**
		↪ *This rule parses a subscript and a superscript (or* vice versa*), each script can be a Unicode or LaTeX-style script.*
subsupU	←	**subU supU**
	/	**supU subU**
		↪ *Analogous to the previous rule, but with Unicode scripts only.*
subsupL	←	**subL supL**
	/	**supL subL**
		↪ *Analogous to the previous rule, but with LaTeX-style scripts only.*
subsupSubscript	←	**scriptbase sub**
sub	←	**subU / subL**
subU	←	**unicodeSub**
subL	←	(`"_"` **soperand**)+ ⇝ *allows for nested subscripts*
subsupSuperscript	←	**scriptbase sup**
sup	←	**supU / supL**
supU	←	**unicodeSup**
supL	←	(`"^"` **soperand**)+ ⇝ *allows for nested superscripts*

Unicode subscripts and superscripts. Though not apparent here, their semantic actions ensure that the generated AST matches LaTeX-style scripts. (In `unicodemathml.pegjs`*, these rules are in a different section of the grammar since they essentially constitute sub-parsers,* i.e.*, they don't refer to other rules.)*

nUnicodeSub	←	(`[₊₋]`?) `[₀₁₂₃₄₅₆₇₈₉]`+ ⇝ *numbers with an optional sign*
opUnicodeSub	←	`"₊₋"` ⇝ *translated into* ± (U+00B1 PLUS-MINUS SIGN)
	/	`"₋₊"` ⇝ *translated into* ∓ (U+2213 MINUS-OR-PLUS SIGN)
	/	`[₊₋₌]`
factorUnicodeSub	←	**nUnicodeSub**
	/	`"₍"` **unicodeSub** `"₎"`
elementUnicodeSub	←	**factorUnicodeSub**+
unicodeSub	←	(**elementUnicodeSub / opUnicodeSub**)+
nUnicodeSup	←	(`[⁺⁻]`?) `[⁰¹²³⁴⁵⁶⁷⁸⁹]`+
atomsUnicodeSup	←	`[ⁱⁿ]`+
		↪ *These letters are only available as superscripts because their subscripted variants are not within the code point ranges given in Section 3.12 of the tech note.*

opUnicodeSup	←	"⁺⁻" / "⁻⁺" ⇝ *same deal as for subscripts*
	/	[⁺⁻⁼]
factorUnicodeSup	←	**atomsUnicodeSup**
	/	**nUnicodeSup**
	/	"⁽" **e:unicodeSup** "⁾"
elementUnicodeSup	←	**factorUnicodeSup**+
unicodeSup	←	(**elementUnicodeSup** / **opUnicodeSup**)+

Prescripts, i.e.*, subscripts and superscripts preceding an expression (discussed in Section 4.8). These essentially come for free due to how regular scripts have been defined.*

preScript	←	"(" **script** ")" **operand**
	/	**scriptU operand** ↪ *Unicode-only scripts can be directly adjacent to their base, with no need for parentheses or a space to disambiguate. Note that this is not specified in the tech note, but I thought it would be worthwhile – ₂x₂ is more clear than ₂ x₂, especially when surrounded by more math.*
	/	**script** " " **operand** ↪ *Mixed or LaTeX-style-only scripts do require a space between prescript and base (in fact, if the script were to end with a Unicode script, no space would be strictly required from a disambiguation perspective – but that would be a bit of a mess to integrate into the grammar.*

Scripts above or below their base (discussed in Section 4.9). Note that due to syntactical similarities, this part of the grammar looks somewhat similar to the section responsible for subscripts and superscripts.

abovebelowScript	←	**abovebelowAbovebelow**
	/	**abovebelowAbove**
	/	**abovebelowBelow**
abscript	←	**abovebelow** / **above** / **below** ↪ *Above/below scripts without a base.*
abovebelowAbovebelow	←	**scriptbase abovebelow**
abovebelow	←	"┬" **soperand** "┴" **soperand** ⇝ *"below ··· above ···"*
	/	"┴" **soperand** "┬" **soperand** ⇝ *"above ··· below ···"*
abovebelowAbove	←	**scriptbase above**
above	←	"┴" (**abovebelowAbove** / **soperand**) ⇝ *allows for nesting*
abovebelowBelow	←	**scriptbase below**
below	←	"┬" (**abovebelowBelow** / **soperand**) ⇝ *allows for nesting*

This rule parses the bases of all three kinds of scripts.

scriptbase	←	"|" ⇝ *enables using pipes as script bases*
	/	**primed**
	/	**primedbase**

Contents of scripts – note that this is related to the "operand" rule, but excludes most kinds of scripts. If "operand" were used instead, PEG.js's greediness would make it impossible to parse expressions with both a subscript and a superscript. Similar caveats are the reason for the order of alternatives in other rules.

soperand ← **sfactor**+
/ **basicOperator sfactor**+ ⇝ *enables expressions like* `∫_-∞^+∞`
/ **basicOperator**+ ⇝ *enables expressions like* `ℕ^+`

High-precendence constructs – this includes many of UnicodeMath's basic constructs. Due to the operator precedence encoded within this grammar, they can all appear in fraction numerators and denominators, or in scripts.

sfactor ← **enclosed**
/ **abstractbox**
/ **hbrack**
/ **root**
/ **function**
/ **text**
/ **sizeOverride**
/ **colored**
/ **comment**
/ **tt**
/ **scriptbase scriptU** ⇝ *enables expressions like* `W_δ₁ρ₁σ₂`
/ **primed**
/ **factorial**
/ **entity**

Expressions enclosed in rectangles, circles, or similar shapes, and abstract boxes used to adjust spacing and other parameters (discussed in Section 4.10). The bit mask can be used to assemble custom enclosures from predefined components.

enclosed ← "▭(" **bitmask** "&" **exp** ")"
/ **opEnclosure operand**
opEnclosure ← "▭" / "○" / "▁" / "□" / ⋯

abstractbox ← "□(" **bitmask** "&" **exp** ")"
↪ *Note that the abstract box operator* □ *is* U+25A1 WHITE SQUARE, *visually similar to yet distinct from the enclosure operator* ▭ (U+25AD WHITE RECTANGLE).

Stretchy horizontal brackets (discussed in Section 4.11) that may be placed above or below terms.

hbrack ← [⎴⏝⏞⋯] **operand**
↪ *You might as well stop reading now:* ⎴⏝⏞ *is by far the cutest three-character slice of this thesis. It's only downhill from here on out.*

Roots of various degrees (discussed in Section 4.12).

root ← "√(" **operand** "&" **exp** ")" ⇝ *roots with arbitrary degree*
/ "√" **exp** "▒" **operand** ⇝ *alternate notation for the same construct*
/ "√" **operand**

	/	"∛" **operand**
	/	"∜" **operand**

"Built-in" functions (discussed in Section 4.13).

function	←	**functionName funcApply** (**script** " ")? **operand**
		↪ *The space is necessary to separate LaTeX-style scripts from the function argument.*
	/	**functionName operand**
		↪ *For simple cases like* `sin(x)`.
functionName	←	`"sin"` / `"log"` / ⋯ / `"lim sup"`
		↪ *There's 61 of these predefined function names.*
funcApply	←	U+2061 FUNCTION APPLICATION / "⁡"

Plain text zones (discussed in Section 4.14). Roughly equivalent to LaTeX's `\text{⋯}`.

text	←	a double quotation mark, followed by any number of non-quotation-mark characters (but quotation marks escaped by a backslash are permitted), followed by a terminating double quotation mark
		↪ *The formal rule is a mess of quotation marks and escapes.*

Font size adjustments (discussed in Section 4.15).

sizeOverride	←	"Ⅎ" [A-D] (**operand** / **basicOperator**)

Non-standard extensions: colors, comments and typewriter font (discussed in Section 4.16).

colored	←	"✎(" **color** "&" **exp** ")"
	/	"☁(" **color** "&" **exp** ")"
color	←	any string that does not contain "&"
		↪ *I've left it up to the MathML renderer to interpret these. Any color names or notations legal in CSS appear to work in Safari, Firefox, and with MathJax.*
comment	←	"⫷" any string not containing ⫸ unless backslash-escaped "⫸"
tt	←	"ￗ(" any string not containing a closing parenthesis unless backslash-escaped ")"
		↪ *Read* ￗ *as "tt", even though it's actually* U+FFD7 HALFWIDTH HANGUL LETTER YU.

Identifiers and other constructs with one or many primes (discussed in Section 4.17).

primed	←	**primedbase prime**+
primedbase	←	**entity**
	/	**basicOperator**
	/	**opNary**
prime	←	"'" (U+0027 APOSTROPHE)
	/	"′" (U+2032 PRIME)
	/	"″" (U+2033 DOUBLE PRIME)
	/	"‴" (U+2034 TRIPLE PRIME)
	/	"⁗" (U+2057 QUADRUPLE PRIME)

Single and double factorials (discussed in Section 4.18).

factorial	←	**entity opFactorial**
opFactorial	←	"‼" (U+203C DOUBLE EXCLAMATION MARK) / "!!" / "!"

Highest-precedence constructs – including delimiter pairs, which are high-precedence operators with regard to what's surrounding them, but low-precedence operators with respect to their contents (which are recursively defined to be entire UnicodeMath expressions).

entity	←	**atoms**
	/	**doublestruck**
	/	**number**
	/	**expBracket**

Identifiers: single characters or words, potentially with diacritical marks (discussed in Section 4.19).

atoms	←	**atom**+
atom	←	**diacriticized**
	/	`"\" .` ↪ *Preceding any character with a backslash, in its function as the "literal operator", removes any UnicodeMath-specific meaning from that character and renders it as an operator.*
	/	**αn**
	/	**mathspaces** ↪ *Spaces can be used as null arguments, for example in script bases, to steer kerning. An example of this usage can be found towards the end of Section 3.16 of the tech note.*

Diacriticized characters, numbers, and expressions.

diacriticized	←	**diacriticbase diacritics**
diacriticbase	←	**αn** ⇝ *single character*
	/	**nn** ⇝ *single-digit number*
	/	`"("` **exp** `")"` U+00A0 NO-BREAK SPACE? ↪ *The optional non-breaking space may be used to visually decouple diacritics (which are usually merged with the preceding character by text renderers – even in code-focused text editors) from the closing parenthesis.*
diacritics	←	**diacritic**+
diacritic	←	`[\u0300-\u036F\u20D0-\u20FF]` ↪ *Unicode's "Combining Diacritical Marks" and "Combining Diacritical Marks for Symbols" blocks.*

αn	←	**αnMath / αnOther / emoji**
αnMath	←	`[\uE000-\uE3FF\u2102-\u2131\u2133\u2134]` ↪ *Math variants of (mostly) Latin and Greek letters, some of which have been mapped into the BMP's "Private Use Area" block as described in Section 3.2.2.*
αnOther	←	Latin and Greek letters ↪ *Note that ideally, all code points in Unicode's L∗ categories should be included here, but JavaScript's implementation of regular expressions provides no built-in method for matching specific Unicode categories.*
emoji	←	a giant regular expression matching all currently-defined emoji ↪ *Some emoji have been mapped into the BMP's "Private Use Area" block. Note that whether a character is an emoji sometimes depends on context, as discussed in Section 7.1.2.*

Math spaces, see Section 3.16 of the tech note.

```
mathspaces       ←   mathspace+
mathspace        ←   U+200B ZERO WIDTH SPACE   ⟿ 0/18 em
                 /   U+200A HAIR SPACE   ⟿ 1/18 em
                 /   twice U+200A HAIR SPACE   ⟿ 2/18 em
                 /   U+2009 THIN SPACE   ⟿ 3/18 em
                 /   …
                 /   U+2002 EN SPACE   ⟿ 1/2 em
                 /   U+2003 EM SPACE   ⟿ 1 em
                 /   U+2007 FIGURE SPACE   ⟿ digit-width space
                     ↪ Note that there is no pre-defined digit width in MathML. The only font-
                     agnostic way of producing a digit-width space in MathML is to take a digit
                     and wrap it within a phantom – so that's what I implemented.
                 /   U+00A0 NO-BREAK SPACE   ⟿ space-width space
```

Numbers, e.g., `1.2` *or* `3` *or* `.4`.

```
number           ←   opDecimal digits
                 /   digits opDecimal digits
                 /   digits
digits           ←   nn+
nn               ←   [0-9]
                     ↪ Note that ideally, all code points in Unicode's Nd category should be in-
                     cluded here, but JavaScript's implementation of regular expressions provides
                     no built-in method for matching specific Unicode categories.
```

Differential operators (Euler's and Leibnitz's notation), Euler's number, and imaginary units.

```
doublestruck     ←   [ⅅⅆⅇⅈⅉ]
```

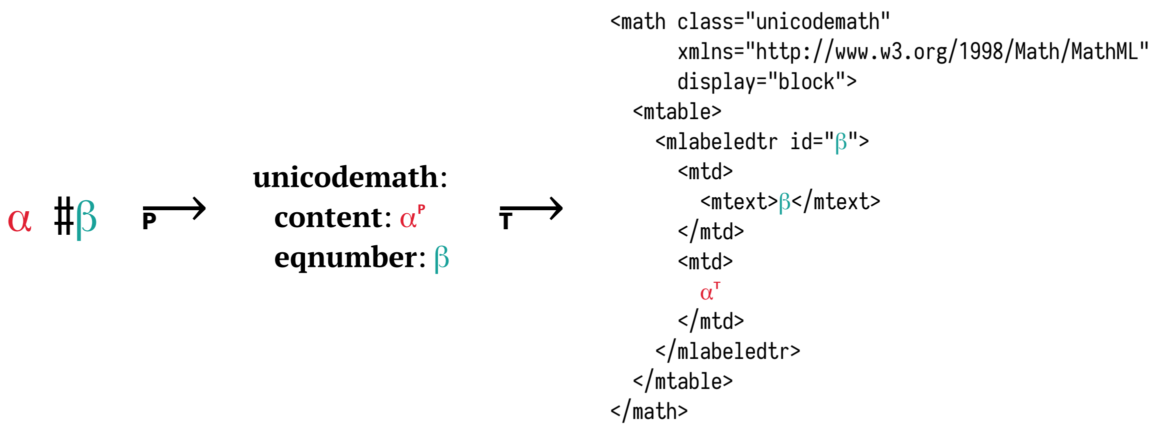
Delimited or bracketed expressions (simplified compared to the real grammar, discussed in Section 4.20).

```
expBracket       ←   ("||" / "‖") exp ("||" / "‖")   ⟿ vector norm
                 /   "|" exp "|"   ⟿ absolute value
                 /   op:expBracketOpen (exp / " "+) cl:expBracketClose
                 /   "Ⓒ(" arows ")" {   ⟿ syntactic sugar for cases
expBracketOpen   ←   "〖"   ⟿ invisible, used for grouping
                 /   opOpen
                 /   "├" bitmask? (opOpen / opClose / "|" / "‖")   ⟿ bracket size
                     override
opOpen           ←   [([{⟨⌈⌊]
expBracketClose  ←   "〗"   ⟿ invisible, used for grouping
                 /   opClose
                 /   "┤" bitmask? (opOpen / opClose / "|" / "‖")
opClose          ←   [)}⟩⌉⌋]]
```

Now that the structure of the grammar should be clear (or as clear as eight pages of a flattened, mostly topologically sorted graph can be), I will explain how the UnicodeMath AST is built up in the grammar's myriad semantic actions and translated to MathML.

This explanation will be based on a non-exhaustive set of examples – non-interesting cases, or ones that are similar to previously discussed cases, will not be separately discussed. Further, certain implementation details will be omitted simply because they are not particularly interesting.

First, consider the entry rule **unicodemath**. In Section 3.21 of the tech note, quoth Sargent: "To represent an equation number flushed right of the equation in UnicodeMath, [the user must] enter the equation followed by a # [...] followed by the desired equation number text." [Sargent16]



```
<math class="unicodemath"
      xmlns="http://www.w3.org/1998/Math/MathML"
      display="block">
  <mtable>
    <mlabeledtr id="β">
      <mtd>
        <mtext>β</mtext>
      </mtd>
      <mtd>
        αT
      </mtd>
    </mlabeledtr>
  </mtable>
</math>
```

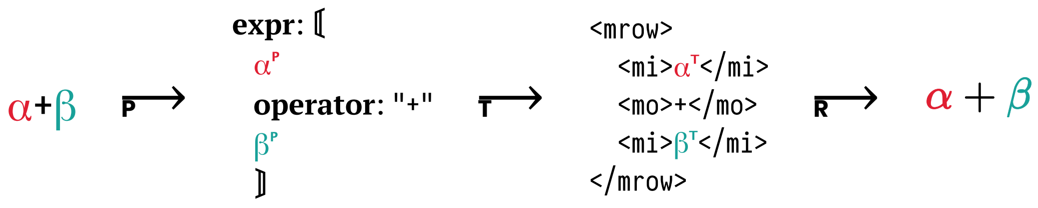
MathML provides no direct means of annotating an expression with an equation number, so the expression must be inserted into an `<mtable>`, with the equation number wrapped in an `<mlabeledtr>` tag. My translator sets a cleaned version of β as the value of the `id` attribute – while nothing of the sort is specified in the tech note, this makes the equation uniquely identifiable via JavaScript or other means, in case the user desires to apply further transformations to it.

Rendered by MathJax (neither Safari nor Firefox support them properly), an equation number defined as `⋯#[42]` looks like this:

$$\hat{f}(\xi)= \int_{-\infty}^{\infty} f(x)e^{-2\pi i x\xi}\,dx \qquad [42]$$

4.2 OPERATORS

A **rawoperator** like `+` is translated as shown below. Note that some operator sequences are mapped to Unicode characters that more closely resemble their meaning – for example, `+-` is mapped to ± (U+00B1 PLUS-MINUS SIGN), U+002A ASTERISK is turned into the vertically centered U+2217 ASTERISK OPERATOR, and the ASCII character U+002D HYPHEN-MINUS becomes U+2212 MINUS SIGN.



```
<mrow>
  <mi>α^T</mi>
  <mo>+</mo>
  <mi>β^T</mi>
</mrow>
```

If a “native” Unicode version of a **negatedoperator** (refer to grammar for how they can be input) exists, it is converted accordingly during preprocessing (*e.g.*, `/∃` becomes U+2204 THERE DOES NOT EXIST). Otherwise, the diacritical mark U+0338 COMBINING LONG SOLIDUS OVERLAY is superimposed over the operator.

4.3 EQUATION ARRAYS & MATRICES

The grammar rules for equation arrays and matrices are identical – except for their leading operator, which is █ (U+2588 FULL BLOCK) for arrays and ■ (U+25A0 BLACK SQUARE) for matrices – but their semantics differ:

UnicodeMath matrices consist of `@`-separated rows, each of which contains `&`-separated values. For example, the UnicodeMath expression `(■(1&2&3@4&5&6@7&8&9@10))` renders as

$$\begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \\ 10 & & \end{pmatrix}.$$

Since such parenthesized matrices are common, UnicodeMath includes syntactic sugar for them. The given example could also have been formulated as `⒨(1&2@3&4)`, whose operator is U+24A8 PARENTHESIZED LATIN SMALL LETTER M.

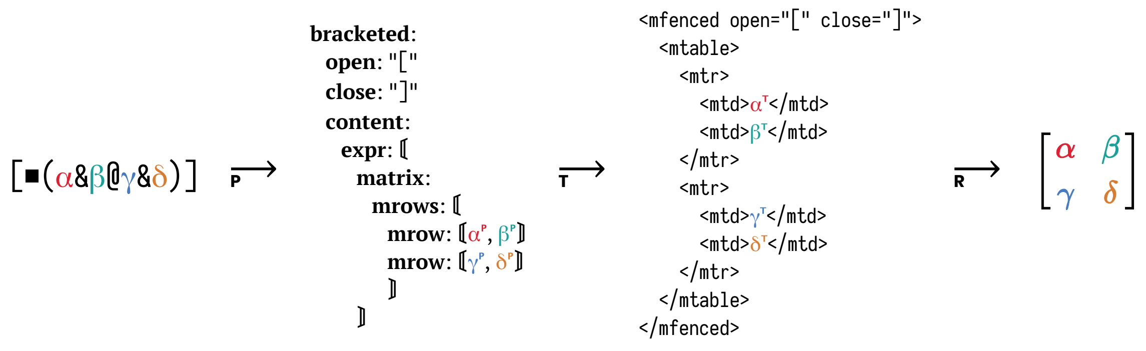
In MathML, matrices can be expressed using `<mtable>`s which, analogously to HTML `<table>`s, contains rows within `<mtr>` elements and column values in `<mtd>` tags. Note that all MathML renderers display tables where the column count differs among rows “correctly”, *i.e.*, missing columns are implicitly treated as being empty (instead of resulting in an error), as is the case in the example above.

Equation arrays also consist of `@`-separated rows, but their rows have semantics inspired by LaTeX's `align` environment, as noted in Section 3.19 of the tech note: “Here the meaning of the ampersands alternate between *align* and *spacer*, with an implied *spacer* at the start of the line. So every odd `&` is an alignment point and every even `&` is a place where space may be added to align the equations.” [Sargent16] In my implementation, this division of ampersands into alignment points and spacers is performed in the preprocessing stage. For example, the UnicodeMath expression `█(10&x+&3&y=2@3&x+&13&y=4)` should render as follows:

$$\begin{array}{rr} 10x + & 3y = 2 \\ 3x+ & 13y = 4 \end{array}$$

(Note that MathJax inserts too much space between the two columns – Firefox's built-in renderer does better.) However, the visual example has been aligned manually (with a bunch of varying-width space): In MathML, the `<malignmark>` element is intended[1] to be used for such alignment purposes. Sadly, no renderer seems to implement it – as a result, equation arrays are rendered like matrices, *i.e.*, the alignment points are ignored.

Here are two basic examples of matrices and arrays:



```
[■(α&β@γ&δ)]
```

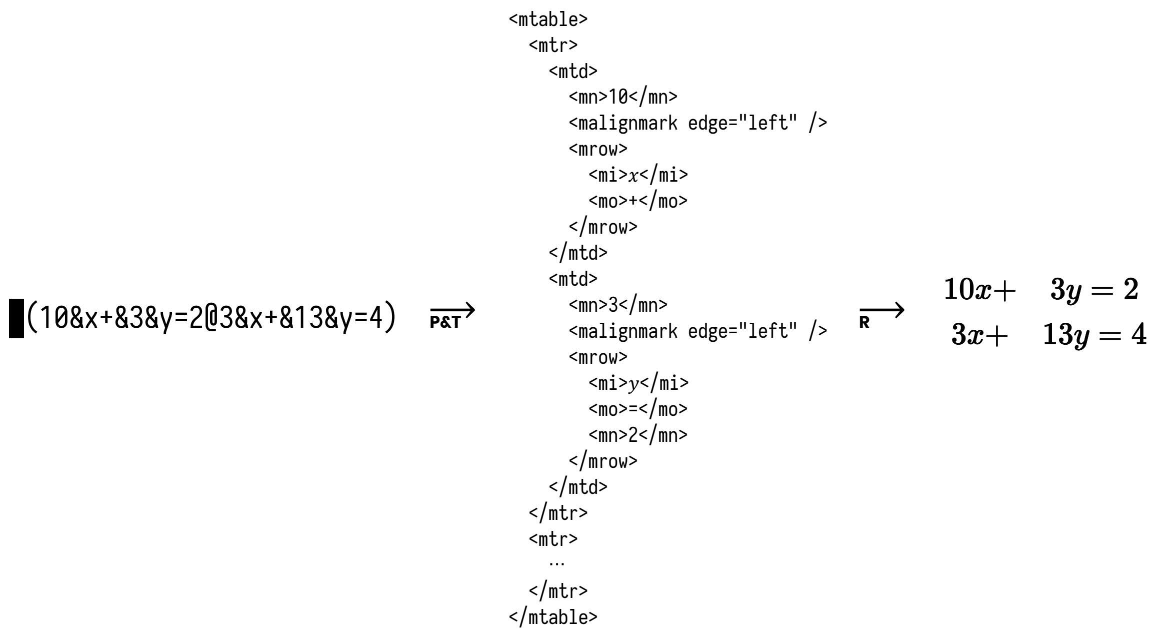
$\xrightarrow{P}$

```
bracketed:
  open: "["
  close: "]"
  content:
    expr: ⟦
      matrix:
        mrows: ⟦
          mrow: ⟦αᴾ, βᴾ⟧
          mrow: ⟦γᴾ, δᴾ⟧
        ⟧
    ⟧
```

$\xrightarrow{T}$

```
<mfenced open="[" close="]">
  <mtable>
    <mtr>
      <mtd>αᵀ</mtd>
      <mtd>βᵀ</mtd>
    </mtr>
    <mtr>
      <mtd>γᵀ</mtd>
      <mtd>δᵀ</mtd>
    </mtr>
  </mtable>
</mfenced>
```

$\xrightarrow{R}$ $\begin{bmatrix} \alpha & \beta \\ \gamma & \delta \end{bmatrix}$



```
█(10&x+&3&y=2@3&x+&13&y=4)
```

$\xrightarrow{P\&T}$

```
<mtable>
  <mtr>
    <mtd>
      <mn>10</mn>
      <malignmark edge="left" />
      <mrow>
        <mi>x</mi>
        <mo>+</mo>
      </mrow>
    </mtd>
    <mtd>
      <mn>3</mn>
      <malignmark edge="left" />
      <mrow>
        <mi>y</mi>
        <mo>=</mo>
        <mn>2</mn>
      </mrow>
    </mtd>
  </mtr>
  <mtr>
    ...
  </mtr>
</mtable>
```

$\xrightarrow{R}$ $\begin{array}{rr} 10x+ & 3y = 2 \\ 3x+ & 13y = 4 \end{array}$

[1] See `https://www.w3.org/TR/MathML3/chapter3.html#presm.malign`.

4.4 *N*-ARY OPERATIONS

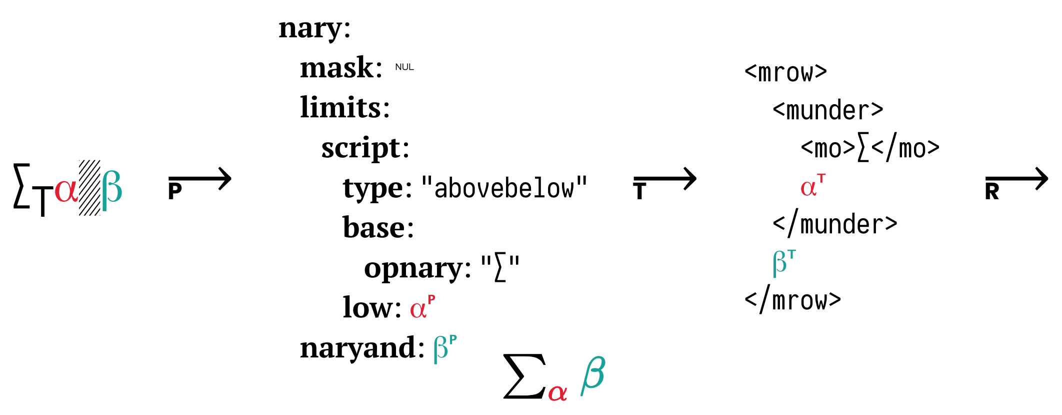
Integrals and sums are the most common n-ary operations – but n-ary products, set unions, logical conjunctions and others[1] also exist. As shown in the grammar a couple of pages ago, UnicodeMath uses the special operator ▒ (U+2592 MEDIUM SHADE) to link the n-ary operation with its operand (dubbed n-aryand by Sargent). Further, a bit mask may be input between the n-ary operator and the script defining its limits.

Consider the binomial theorem, which can be expressed as `(a + b)^n = ∑1_(k=0)^n▒(n¦k) a^k b^(n-k)` in UnicodeMath:

$$(a+b)^n = \sum_{k=0}^{n} \binom{n}{k} a^k b^{n-k}$$

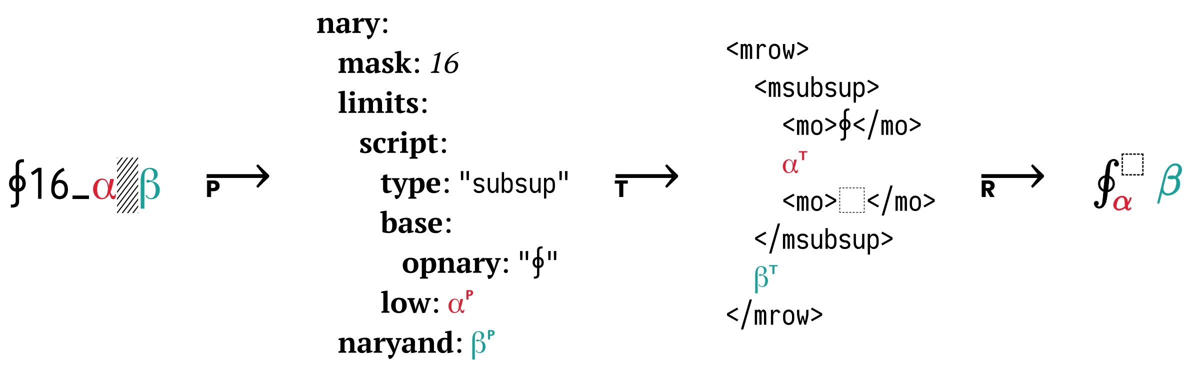
The 1 in `⋯∑1_(k=0)^n▒⋯` ensures that the limits are shown above and below the operator, even though they are specified as subscript and superscript. Other bit mask values can instead force the limits to display as subscript and superscript (or just treat the upper limit this way) or they can show placeholders when a script is missing. Section 3.4 of the tech note gives an overview of the possible bit masks.

(Note that MathJax renders abovescripts and belowscripts (see Section 4.9) as subscripts and superscripts if 1. the base is an n-ary operator and 2. `textstyle` mode is active, which happens to be the case for these examples for technical reasons.)



```
nary:
  mask: NUL
  limits:
    script:
      type: "abovebelow"
      base:
        opnary: "∑"
      low: α^P
  naryand: β^P
```

```
<mrow>
  <munder>
    <mo>∑</mo>
    α^T
  </munder>
  β^T
</mrow>
```

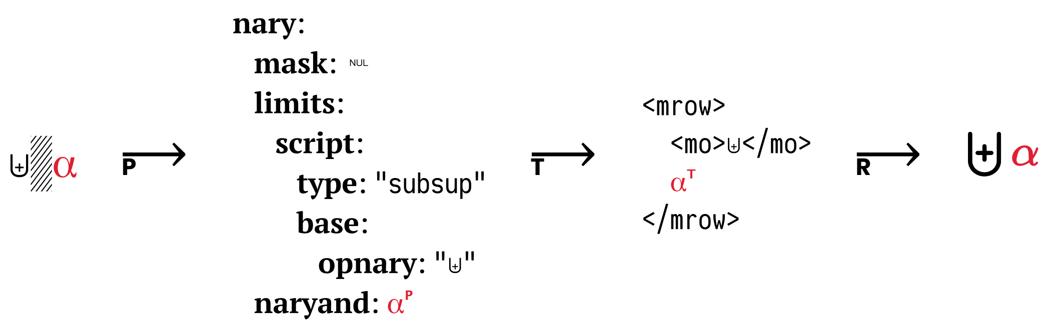


```
nary:
  mask: 16
  limits:
    script:
      type: "subsup"
      base:
        opnary: "∮"
      low: α^P
  naryand: β^P
```

```
<mrow>
  <msubsup>
    <mo>∮</mo>
    α^T
    <mo>⬚</mo>
  </msubsup>
  β^T
</mrow>
```

[1] See `https://www.cs.bgu.ac.il/~khitron/Equation%20Editor.pdf`.

If no script is defined, a dummy script is created during parsing and – if no placeholder-inserting bit mask like in the previous example is specified – removed upon translation to MathML:



```
nary:
  mask: ␀
  limits:
    script:
      type: "subsup"
      base:
        opnary: "⊎"
  naryand: α^P
```

```
<mrow>
  <mo>⊎</mo>
  α^T
</mrow>
```

4.5 PHANTOMS & SMASHES

Although they are not the most frequently used tools, phantoms and smashes come in handy when “one wants to obtain horizontal and/or vertical spacings that differ from the normal values” [Sargent16]. For example, in one of Murray Sargent's blog posts, he presents[1] a variant of the mode locking equation

$$\frac{1}{2\pi}\int_0^{2\pi}\frac{d\theta}{a+b\sin\theta}=\frac{1}{\sqrt{a^2-b^2}},$$

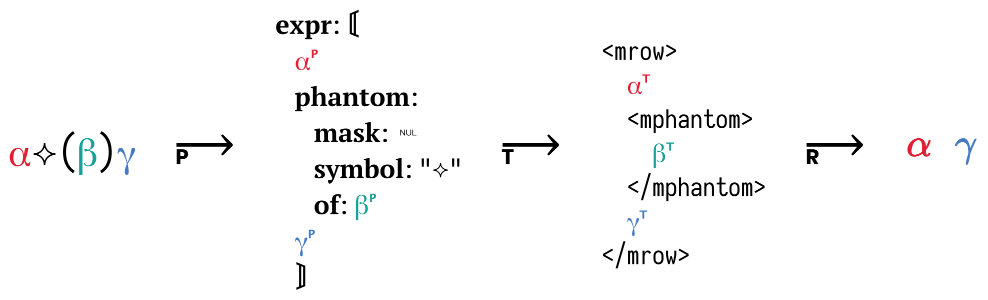
noting that “you can see that there's a little too much room between the 2π and the integrand. You can pull the integrand to the left under the π by ‘smashing’ its width. Inside the upper limit, you type `[2⬌(π)]`, and the π displays, but has no horizontal width” [Sargent11] as in

$$\frac{1}{2\pi}\int_0^{2\smash{\pi}}\frac{d\theta}{a+b\sin\theta}=\frac{1}{\sqrt{a^2-b^2}}.$$

In MathML, simple phantoms are represented using the `<mphantom>` elements. Horizontal and vertical phantoms also require the `<mpadded>`[2] element, as do smashes. In this section, I will show three examples: A simple **phantom** that does not render but takes up space, a vertical phantom that does not render and takes up only vertical space, and a horizontal **smash** that renders its argument but makes it take up no space.

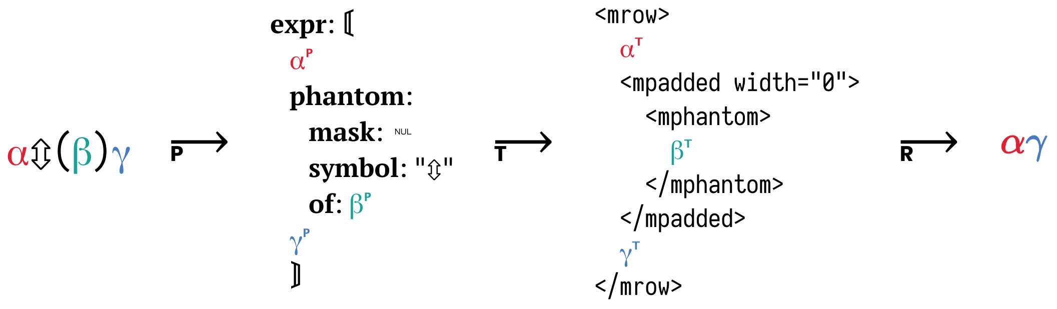
[1] In the same post, he writes about how “some colleagues and I had the good fortune to spend an extraordinary afternoon with Donald Knuth, the primary author of TeX, at his home on the Stanford University campus. Among many things, Donald showed us how he uses TeX to typeset his computer-science papers and books exactly the way he wants them to look. In particular, he applies special tweaks to achieve perfection, such as ‘smashing the descender’ on one radicand to make a sum of square roots line up in a pleasing way, and such as shimming characters to place them more beautifully in a formula.” [Sargent11]

[2] Note that Safari presently does not support `<mpadded>` properly: If its `width` and `height` attributes are set to `0`, the content of the element is not visible, even though it should be.



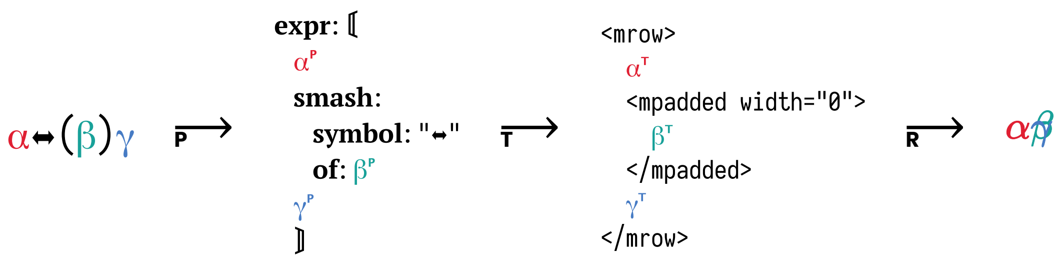
```
expr: ⟦
  α^P
  phantom:
    mask: NUL
    symbol: "✧"
    of: β^P
  γ^P
  ⟧
```

```
<mrow>
  α^T
  <mphantom>
    β^T
  </mphantom>
  γ^T
</mrow>
```



```
expr: ⟦
  α^P
  phantom:
    mask: NUL
    symbol: "⇳"
    of: β^P
  γ^P
  ⟧
```

```
<mrow>
  α^T
  <mpadded width="0">
    <mphantom>
      β^T
    </mphantom>
  </mpadded>
  γ^T
</mrow>
```



```
expr: ⟦
  α^P
  smash:
    symbol: "⬌"
    of: β^P
  γ^P
  ⟧
```

```
<mrow>
  α^T
  <mpadded width="0">
    β^T
  </mpadded>
  γ^T
</mrow>
```

Bit masks allow equation authors to build their own phantom-smash combinations – this is explored in Section 3.17 of the tech note.

4.6 FRACTIONS

UnicodeMath defines four types of fractions:

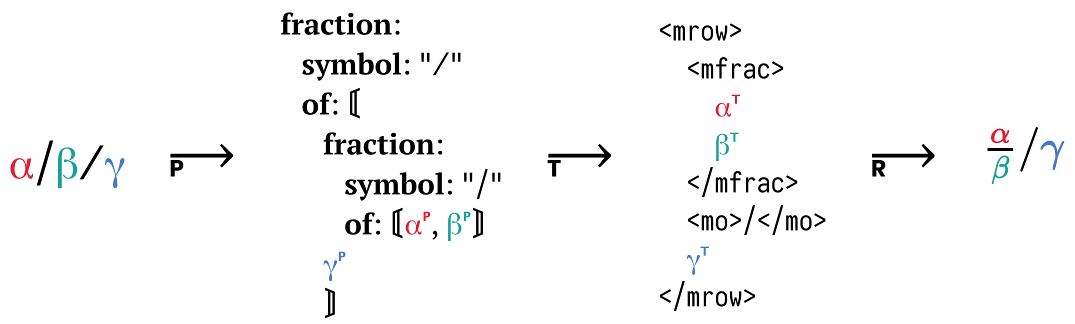
- Normal fractions, denoted by a standard forward slash (U+002F SOLIDUS) and rendered as $\frac{\alpha}{\beta}$
.
- Skewed fractions, whose operator is U+2044 FRACTION SLASH, and which render as ${}^{\alpha}/_{\beta}$.
- Linear fractions, written with U+2215 DIVISION SLASH, and rendered as α/β. Alternatively, they can be entered by preceding a standard forward slash with a backslash – this way, the size of the slash can be adjusted as required (see Section 4.15 for a discussion of size overrides). For example, the fraction slash in the following formula, taken from page 57 of the digital typography book, is denoted as ⅃B\/ in UnicodeMath.

$$\sum_k (-1)^k z_k f(t-k) \Big/ \sum_k (-1)^k f(t-k)$$

- Small numeric fractions, written using `U+2298 CIRCLED DIVISION SLASH`, and rendered a bit smaller than normal fractions: compare $\frac{\alpha}{\beta}$ and $\frac{\alpha}{\beta}$.

In all four cases, the outermost parentheses of numerator and denominator are removed during translation to enable intuitive grouping. This is described in Section 2.1 of the tech note. Also note that "fraction operators have left-to-right associativity as in common programming languages like C/C++/C#." [Sargent16]

MathML provides the `<mfrac>` element which must contain two children and renders as a normal fraction by default. If the optional `bevelled` attribute is set to `true`, a skewed fraction is displayed instead. UnicodeMathML handles linear fractions by inserting `<mo>/</mo>` between numerator and denominator. Here's an example:



α/β/γ $\xrightarrow{P}$

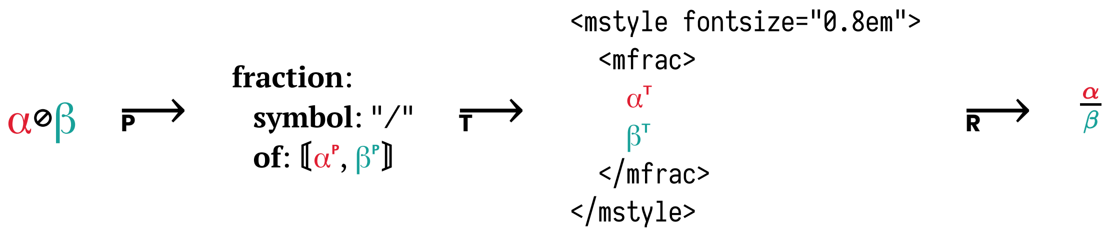
```
fraction:
  symbol: "/"
  of: ⟦
    fraction:
      symbol: "/"
      of: ⟦α^P, β^P⟧
    γ^P
  ⟧
```

$\xrightarrow{T}$

```
<mrow>
  <mfrac>
    α^T
    β^T
  </mfrac>
  <mo>/</mo>
  γ^T
</mrow>
```

$\xrightarrow{R}$ $\frac{\alpha}{\beta}/\gamma$

The translation of small numeric fractions into MathML is achieved by emitting an `<mfrac>` nested inside an `<mstyle>` that locally modifies the font size:



α⊘β $\xrightarrow{P}$

```
fraction:
  symbol: "/"
  of: ⟦α^P, β^P⟧
```

$\xrightarrow{T}$

```
<mstyle fontsize="0.8em">
  <mfrac>
    α^T
    β^T
  </mfrac>
</mstyle>
```

$\xrightarrow{R}$ $\frac{\alpha}{\beta}$

4.7 SUBSCRIPTS & SUPERSCRIPTS

“ *A mathematician spends a lot of time choosing notations for things, and one of the things we try to avoid in mathematics is double subscripts. I read one French Ph.D. thesis where the author had five levels of subscripts [laughter] – he kept painting himself into a corner. He started out with a set* $\{x_1, \ldots, x_n\}$*, so then when he talked of a subset, it had to be* $\{x_{i_1}, \ldots, x_{i_m}\}$*, and then he wanted to take a subset of this; finally he had a theorem that referred to ‘* $x_{j_{ik1}} \ldots x_{i_{j_{kr}}}$*’. [laughter]* ”

— Donald Knuth in a Q&A session, from page 643 of the digital typography book [Knuth99]

I'm self-aware enough to realize that exhaustively explaining how subscripts and superscripts are parsed might make you want to murder me, so instead, I will briefly describe the reasoning behind the general approach before presenting a couple of examples.

“Scripted” expressions as recognized by my UnicodeMath parser can have

- one script: a superscript *or* a subscript, this case is not very interesting either way;
- two scripts: a superscript *and* a subscript or *vice versa*, this case is more interesting; or
- many scripts: a string of superscripts *or* subscripts, *or* a string of superscripts *and* a string of subscripts or *vice versa*; this case is an extension of the previous one.

There are two kinds of subscripts and superscripts which can be used interchangeably in some contexts, but not in others. The first uses the notation known from LaTeX: “[W]e introduce a subscript by a subscript operator, which we display as the ASCII underscore _ as in TeX. Similarly, superscripts are introduced by a superscript operator, which we display as the ASCII ^ as in TeX.” [Sargent16] The outermost set of parentheses within a script is eliminated here in the same manner as it is done for fractions, but parentheses around the base of a script remain untouched – if they are needed to ensure correct grouping when formulating the base expression in UnicodeMath, they are likely also helpful to the reader as they disambiguate the “scope” of the script, as in the following example taken from page 43 of the digital typography book (it's part of a derivation of curves used for letter drawing in Knuth's Metafont):

$$f'(t) = 8\left(\frac{1-\cos\frac{\theta}{2}}{1+\cos\frac{\theta}{2}}\sin\frac{\theta}{2}\right)^2 (t-1)t(2t-1)\big(6t^2-6t+1\big)$$

The second kind of subscripts and superscripts takes advantage of the “Unicode” in “UnicodeMath”: “Unicode contains a small set of mostly numeric superscripts [such as U+00B9 SUPERSCRIPT ONE, U+00B2 SUPERSCRIPT TWO, and U+00B3 SUPERSCRIPT THREE] and a similar set of subscripts [...] that should be rendered the same way that scripts of the corresponding script nesting level would be rendered.” [Sargent16] “The numeric subscripts and superscripts are often used and can streamline the look of technical plain text.” [Sargent16b]

Taking a closer look at the “two scripts” case from above,

- if the first script is a Unicode script, the second script can be another Unicode script *or* a LaTeX-style script, but
- if the first script is a LaTeX-style script, the second script must be one too.

This is not an arbitrary limitation: It enables the use of Unicode scripts *within* LaTeX-style scripts. For example, the UnicodeMath expression `W_δ₁ρ₁σ₂^3β` should render as

$$W_{\delta_1\rho_1\sigma_2}^{3\beta}$$

instead of

$$W_{\delta_1}\rho_1\sigma_2^{3\beta}.$$

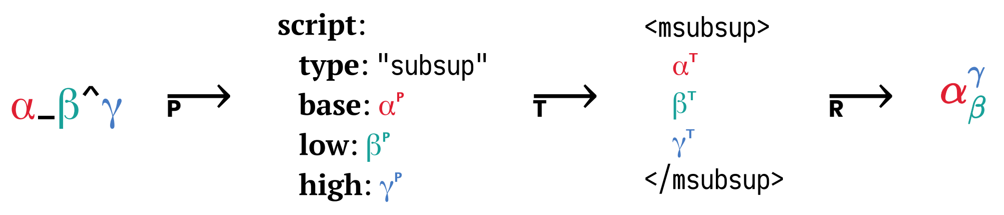
Note that this mini-algorithm is not given in the tech note, I was required to come up with it myself in order to correctly render several examples. It does not allow expressions like `xⁱ^2`, but such expressions are ambiguous, anyway. Refer to Table 2 for a number of notationally non-trivial examples and how they are rendered by UnicodeMathML and MathJax.

UnicodeMath	Rendered MathML
`a^*`	a^*
`a_b^c`	a_b^c
`a₁^b`	a_1^b
`a^b₁`	a^{b_1}
`a^1_2_3_4`	$a^1_{2_{3_4}}$
`a^(1_2)_3_4`	$a^{1_2}_{3_4}$
`a₉^+-b₁`	$a_9^{\pm b_1}$
`W_δ₁ρ₁σ₂^3β`	$W_{\delta_1\rho_1\sigma_2}^{3\beta}$
`mⁿ₋₃₌₍₂₋₅₎`	$m^n_{-3=(2-5)}$

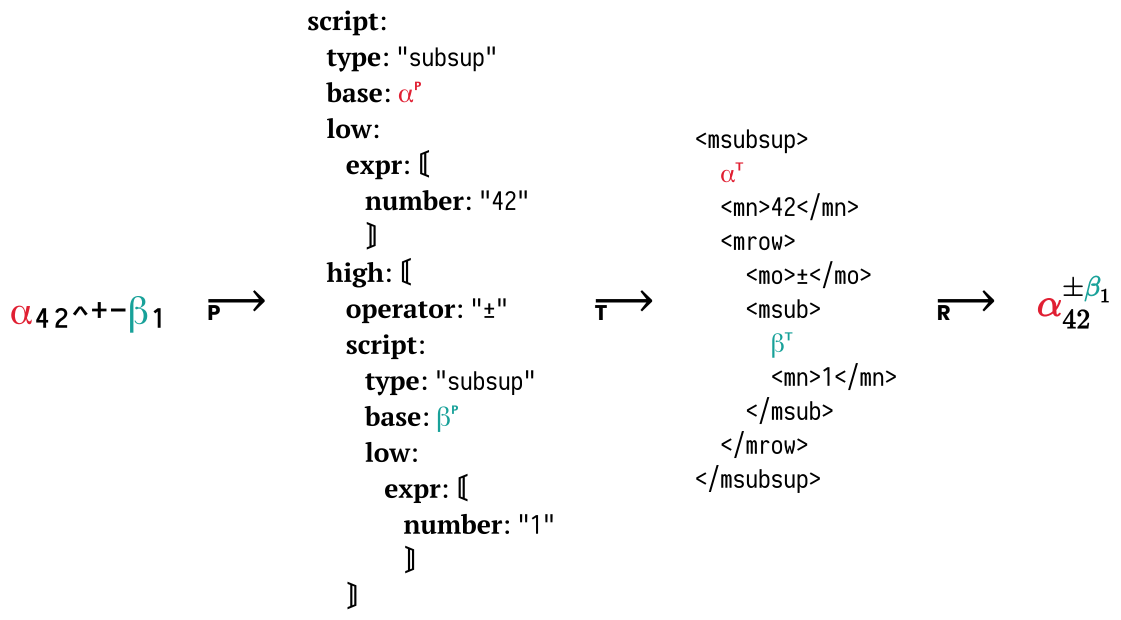
Table 2: *Some superscripts and subscripts.*

In MathML, superscripted expressions can be expressed with the `<msup>` tag, which must have two children: the base first, then the exponent. `<msub>` functions equivalently for subscripts, and `<msubsup>` similarly accepts the base first, the expression to be subscripted next, and finally the exponent.

Since the grammar rules responsible for recognizing subscripts and superscripts are re-used for prescripts and *n*-ary operations (and must sometimes be converted to or from above-/belowscripts, as mentioned in Section 4.4 and Section 4.9), the AST built up during parsing has been kept purposefully clean. Also, Unicode scripts yield the same AST as the equivalent LaTeX-style script.



```
α_β^γ  →P  script:                   →T  <msubsup>       →R  α_β^γ
             type: "subsup"                 α^T
             base: α^P                      β^T
             low: β^P                       γ^T
             high: γ^P                    </msubsup>
```



```
α₄₂^+-β₁  →P  script:                    →T  <msubsup>            →R  α_42^{±β_1}
                type: "subsup"                  α^T
                base: α^P                       <mn>42</mn>
                low:                            <mrow>
                  expr: ⟦                         <mo>±</mo>
                    number: "42"                  <msub>
                  ⟧                                 β^T
                high: ⟦                             <mn>1</mn>
                  operator: "±"                   </msub>
                  script:                       </mrow>
                    type: "subsup"            </msubsup>
                    base: β^P
                    low:
                      expr: ⟦
                        number: "1"
                      ⟧
                ⟧
```

4.8 PRESCRIPTS

Prescripts are subscripts and/or superscripts attached to the left side of an expression. In UnicodeMath, their syntax is equivalent to the subscript/superscript syntax, which enables reuse of the existing grammar rules for subscripts and superscripts, but also requires them to be separated from their base

- either by a space, as in `_α^β γ`,
- or by being surrounded with a set of parentheses, as in `(^β_α)γ`;

both variants render as ${}_{\alpha}^{\beta}\gamma$. Without a separator, the base would in most cases be indistinguishable from the script. Note that if *only* Unicode scripts are present, no separator is needed: $^1{}_2 3$ renders as ${}_{2}^{1}3$.

MathML's `<mmultiscripts>` element can be used to denote prescripts:

$\_\beta\hat{}\gamma\ \ \alpha$ $\xrightarrow[\mathsf{P}]{}$

script:
- **type**: `"pre"`
- **base**: α^{P}
- **prelow**: β^{P}
- **prehigh**: γ^{P}

$\xrightarrow[\mathsf{T}]{}$

<pre>
<mmultiscripts>
  α^T
  <mprescripts />
  β^T
  γ^T
</mmultiscripts>
</pre>

$\xrightarrow[\mathsf{R}]{}$ ${}_{\beta}^{\gamma}\alpha$

"Variables can have both prescripts and postscripts (ordinary subscripts and superscripts)" [Sargent16], which, while conceptually obvious, requires prescripts and postscripts to be merged into a single script – a *multiscript* (hence the name of the MathML element).

$(\_\beta\hat{}\gamma)\alpha\_\delta\hat{}\varepsilon$ $\xrightarrow[\mathsf{P}]{}$

script:
- **type**: `"pre"`
- **base**:
  - **script**:
    - **type**: `"subsup"`
    - **base**: α^{P}
    - **low**: δ^{P}
    - **high**: ε^{P}
- **prelow**: β^{P}
- **prehigh**: γ^{P}

$\xrightarrow[\dots]{}$

script:
- **type**: `"pre"`
- **base**: α^{P}
- **prelow**: β^{P}
- **prehigh**: γ^{P}
- **low**: δ^{P}
- **high**: ε^{P}

$\xrightarrow[\mathsf{T}]{}$

<pre>
<mmultiscripts>
  α^T
  δ^T
  ε^T
  <mprescripts />
  β^T
  γ^T
</mmultiscripts>
</pre>

$\xrightarrow[\mathsf{R}]{}$ ${}_{\beta}^{\gamma}\alpha_{\delta}^{\varepsilon}$

Without such a merging step, the UnicodeMath expression `^1_2 n^3_4` would render as

$${}_{2}^{1}n_4^3 \quad \text{or} \quad {}_{2}^{1}{n_4}^{3} \quad \text{instead of} \quad {}_{2}^{1}n_{4}^{3}.$$

Note that one could, if so inclined, represent the Wilhelm scream[1] in UnicodeMath with a prescript and a string of subscripts:

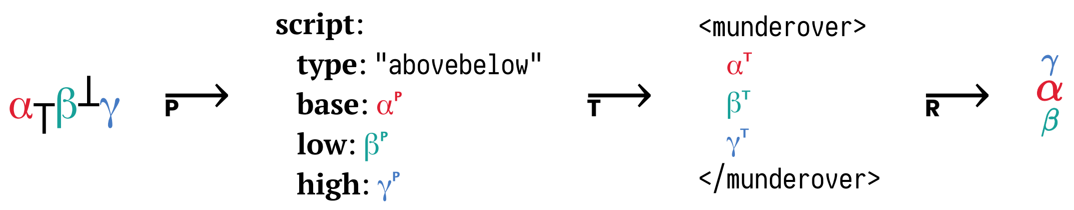
$${}_{a}a_{a_{a_{a_{a_{a_{a_{a_{u_{g_{h}}}}}}}}}}$$

[1] See `https://www.youtube.com/watch?v=9ua2RWL5big`.

4.9 ABOVE/BELOW SCRIPTS

The third kind of script supported by UnicodeMath comprises belowscripts and abovescripts, which are represented by the line drawing operators ┬ (U+252C BOX DRAWINGS LIGHT DOWN AND HORIZONTAL) and ┴ ___(U+2534). They can be repeated and nested in the same manner as LaTeX-style subscripts and superscripts.

The MathML elements `<mover>`, `<munder>` and `<munderover>` are the above/below script equivalents of `<msup>`, `<msub>` and `<msubsup>`.



In accordance with Section 3.3 of the tech note, if the `displaystyle` flag is set, subscripts attached to a function apply operator that follows the function names "det, gcd, inf, lim, lim inf, lim sup, max, min, Pr, and sup" [Sargent16] are transformed[1] into belowscripts. For example, the UnicodeMath expression `lim▒_(n→∞) a_n` renders as $\lim_{n\to\infty} a_n$ within a paragraph, but as

$$\lim_{n\to\infty} a_n$$

when it stands on its own.

Many of Unicode's arrow characters, when placed over or under an expression, will stretch to fit its width, provided they are wrapped in an `<mo stretchy="true">` tag. It works the other way around, too. For example, he nonsensical expression `123┴↔ + ↔┴123` renders as

$$\overset{\longleftrightarrow}{123} + \overset{123}{\leftrightarrow}.$$

One could, if so inclined, replicate the Unicode character ≝ (U+225D EQUAL TO BY DEFINITION) with the UnicodeMath expression `=┴"def"`:

$$≝ \approx \overset{\text{def}}{=}$$

4.10 ENCLOSURES & ABSTRACT BOXES

Sometimes, it's useful to highlight a subexpression by circling it, underlining it or drawing a rectangle around it. UnicodeMath addresses this demand with enclosures, which can be

[1] If only a subscript is present, this is done by changing the `type` attribute of its AST node. If a superscript is included, subscript and superscript are separated into two AST nodes first, with the new belowscript node nested around the function name and its superscript.

summoned by using one of two syntaxes.

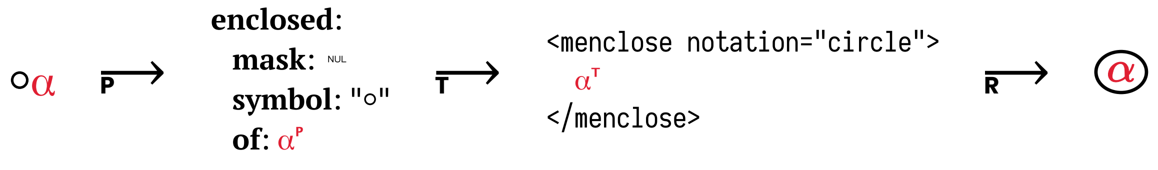
When using the first syntactic variant, one of the enclosure operators – among which are ▭ (U+25AD WHITE RECTANGLE), ○ (U+25CB WHITE CIRCLE), ▁ (U+2581 LOWER ONE EIGHTH BLOCK), and ▢ (U+25A2 WHITE SQUARE WITH ROUNDED CORNERS) – must precede an operand. This results in an enclosure matching the shape of the operator placed around the operand. If the operand is a parenthesized expression, the outermost parentheses are removed in the process. For example, ▢(a+b) renders as

$$\boxed{a+b}.$$

Alternatively, the other variant allows the standard rectangle operator U+25AD WHITE RECTANGLE to be paired with a bit mask. This allows the construction of user-defined enclosures as follows:

- ▭(1&α) yields $\overline{\alpha}$,
- ▭(2&α) yields $\underline{\alpha}$,
- ▭(4&α) yields $|\alpha$,
- ▭(8&α) yields $\alpha|$,
- ▭(16&α) yields $\enclose{horizontalstrike}{\alpha}$,
- ▭(32&α) yields $\enclose{verticalstrike}{\alpha}$,
- ▭(64&α) yields $\bcancel{\alpha}$, and
- ▭(128&α) yields $\cancel{\alpha}$.

Note that these flags can be combined (*i.e.*, added up) as desired, *e.g.*, ▭(19&α) yields $\enclose{top bottom horizontalstrike}{\alpha}$. No matter which notation is used, MathML's `<menclose notation="⋯">` element is the tool for the job – which enclosure is drawn during rendering depends on the value[1] assigned to the `notation` attribute.



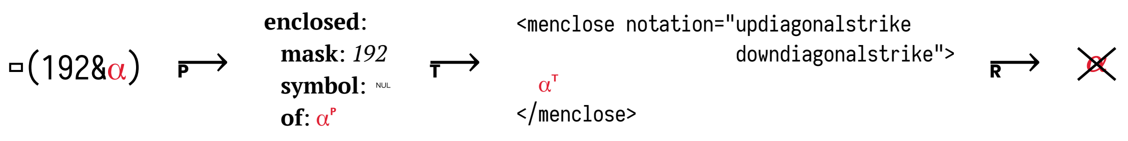
$$○\alpha \xrightarrow{P}$$

enclosed:
  mask: NUL
  symbol: "○"
  of: α^P

$\xrightarrow{T}$

```
<menclose notation="circle">
  α^T
</menclose>
```

$$\xrightarrow{R} \enclose{circle}{\alpha}$$



$$▭(192\&\alpha) \xrightarrow{P}$$

enclosed:
  mask: *192*
  symbol: NUL
  of: α^P

$\xrightarrow{T}$

```
<menclose notation="updiagonalstrike
                    downdiagonalstrike">
  α^T
</menclose>
```

$$\xrightarrow{R} \xcancel{\alpha}$$

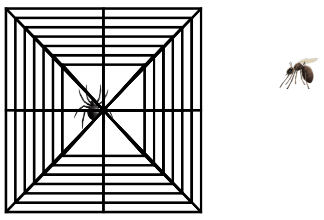
Note that enclosures need not be composed of black lines – they inherit the current text color, which can be set as explained in Section 4.16.

Briefly mentioned in Section 3.7 of the tech note, "an abstract box can be put around an expression [...] to change alignment, spacing category, size style *[sic]*, and other properties." [Sargent16] The syntax of abstract boxes is equivalent to the bit mask variant of

[1] See https://developer.mozilla.org/en-US/docs/Web/MathML/Element/menclose#Attributes.

enclosures, except the operator □ (U+25A1 WHITE SQUARE) is used. I have not fully implemented abstract boxes for the reasons detailed in Section Section 7.1.3.

Apropos of nothing: here's an enclosure-powered drawing of a spider about to have a snack.



4.11 HORIZONTAL BRACKETS

Introduced in Section 3.8 of the tech note, stretchy horizontal brackets may be used to group...

$$\underparen{a_1+b_1}+\underparen{a_2+b_2}+\underparen{a_3+b_3}$$

...or annotate expressions. For example, overbraces can be used to indicate how many times something repeats:

$$a_1+a_2+\cdots+a_{i-1}+a_i+\overbrace{a_{i+1}+\cdots+a_{n-1}+a_n}^{n-i\text{ times}}$$

These brackets come in two flavors:

- Those that render above the expression they precede: ⏜ (U+23DC TOP PARENTHESIS), ⏞ (U+23DE TOP CURLY BRACKET), ⏠ (U+23E0 TOP TORTOISE SHELL BRACKET), ⎴ (U+23B4 TOP SQUARE BRACKET), and ¯ (U+00AF MACRON).
- Ones that display below their argument: ⏝ (U+23DD BOTTOM PARENTHESIS), ⏟ (U+23DF BOTTOM CURLY BRACKET), ⏡ (U+23E1 BOTTOM TORTOISE SHELL BRACKET), and ⎵ (U+23B5 BOTTOM SQUARE BRACKET).

If the expression following the bracket does *not* possess a script, the bracket is simply placed above or below it (sans outermost parentheses) via MathML's `<mover>` or `<munder>`. In order to convince MathML renderers to stretch the bracket along the expression, it needs to be wrapped in an `<mo stretchy="true">` tag.

⏜α $\xrightarrow[\text{P}]{}$

hbrack:
  bracket: "⏜"
  of: α^{P}

$\xrightarrow[\text{T}]{}$

```
<mover>
  αᵀ
  <mo stretchy="true">⏜</mo>
</mover>
```

$\xrightarrow[\text{R}]{}$ $\overparen{\alpha}$

When such a lower bracket is followed by a subscripted (or belowscripted) expression, the subscript is converted into a belowscript and the bracket is inserted *between* it and its base. Note that the outermost level of parentheses around the base is removed as part of this transformation. (Upper brackets are treated analogously.)

⏟α_β $\xrightarrow{P}$

```
hbrack:
  bracket: "⏟"
  of:
    script:
      type: "subsup"
      base: α^P
      low: β^P
```

$\xrightarrow{T}$

```
<munder>
  <munder>
    <mi>α^T</mi>
    <mo stretchy="true">⏟</mo>
  </munder>
  <mi>β^T</mi>
</munder>
```

$\xrightarrow{R}$ $\underbrace{\alpha}_{\beta}$

4.12 ROOTS

UnicodeMath comes with five kinds of root-related notation:

- square roots √α which render as $\sqrt{\alpha}$,
- nth-degree roots √(n&α), which render as $\sqrt[n]{\alpha}$
- another notation for nth-degree roots √n▒α, which renders in the same way,
- cube roots ∛α, which I implemented as syntactic sugar that resolves to √(3&α), and
- fourth roots ∜α, which are analogously sweet.

Apart from the removal of outermost parentheses from the argument, there's no trickery or special cases required when translating them into MathML, so they are not particularly interesting – here's how two examples make it through the pipeline:

√α $\xrightarrow{P}$ **sqrt**: α^P $\xrightarrow{T}$

```
<msqrt>
  α^T
</msqrt>
```

$\xrightarrow{R}$ $\sqrt{\alpha}$

√(δ&α) $\xrightarrow{P}$

```
root:
  degree: δ^P
  of: α^P
```

$\xrightarrow{T}$

```
<mroot>
  α^T
  δ^T
</mroot>
```

$\xrightarrow{R}$ $\sqrt[\delta]{\alpha}$

4.13 FUNCTIONS

Section 2.5 of the tech note prescribes that "[m]athematical functions such as trigonometric functions like 'sin' should be recognized as such and not italicized. [...] In addition it's desirable to follow them with the Invisible Function Apply operator [U+2061 FUNCTION APPLICATION] [or ▒ (U+2592 MEDIUM SHADE)]. This is a special binary operator and the operand that follows it is the function argument. [...] If the Function Apply operator is immediately followed by a subscript or superscript expression, that expression should be applied to the function name and the Function Apply operator moved passed *[sic]* the modified name to bind the operand that follows as the function argument."

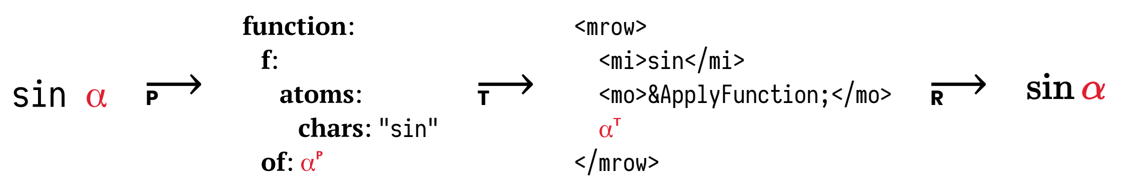
One case where a function conventionally carries a script is in the derivative of the tangent:

$$\frac{d(\tan x)}{dx} = \frac{1}{\cos^2 x}$$

In summary, functions are denoted by

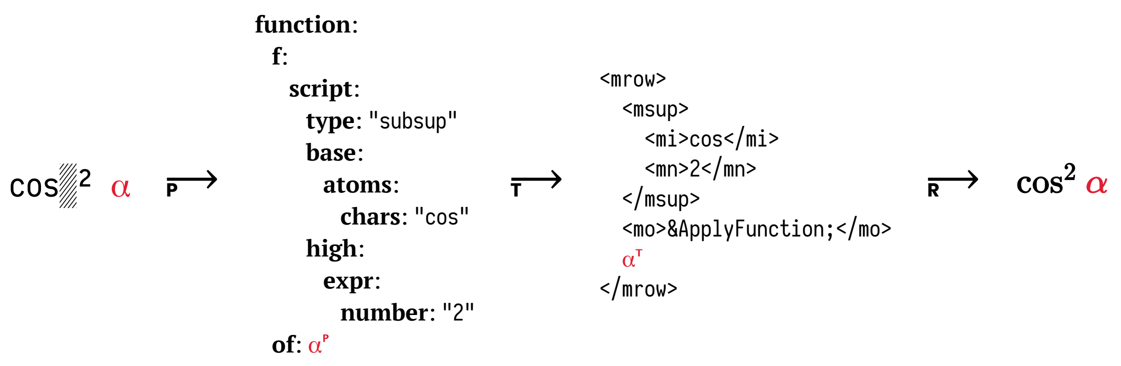
1. a function name (61 common[1] ones are hardcoded in the grammar),
2. an optional function apply operator,
3. an optional postscript that, in the presence of the function apply operator, will be attached to the function name,
4. an optional space (to detach the function from its argument in plain text, but keep them semantically connected), and
5. the function's argument.

Here are two examples:



```
function:
  f:
    atoms:
      chars: "sin"
  of: α
```

```
<mrow>
  <mi>sin</mi>
  <mo>&ApplyFunction;</mo>
  α
</mrow>
```



```
function:
  f:
    script:
      type: "subsup"
      base:
        atoms:
          chars: "cos"
      high:
        expr:
          number: "2"
  of: α
```

```
<mrow>
  <msup>
    <mi>cos</mi>
    <mn>2</mn>
  </msup>
  <mo>&ApplyFunction;</mo>
  α
</mrow>
```

4.14 TEXT

In Section 3.15 of the tech note, Sargent notes that "[s]ometimes one wants ordinary text inside a function argument or in a math zone as in the formula

$$\text{rate} = \frac{\text{distance}}{\text{time}}.$$

[...] For such cases, the alphabetic characters should not be converted to math alphabetic characters and the typography should be that of ordinary text, not math text. To embed such text inside functions or in general in a math zone, the text can be enclosed inside

[1] See https://www.cs.bgu.ac.il/~khitron/Equation%20Editor.pdf.

ASCII double quotes. [...] If you want to include a double quote inside such text, insert \". Another example is $\sin\theta = {}^1\!/_2 e^{i\theta} + \text{c.c.}$ To get the "c.c." as ordinary text, enclose it with ASCII double quotes." [Sargent16]

Thusly motivated, text makes it through the pipeline unscathed as shown below.

$$\begin{tikzcd} \texttt{"}\alpha\texttt{"} \arrow[r, "\mathsf{P}"'] & \textbf{text}\text{: } \alpha \arrow[r, "\mathsf{T}"'] & \begin{array}{l}\texttt{<mtext>}\\ \quad\alpha\\ \texttt{</mtext>}\end{array} \arrow[r, "\mathsf{R}"'] & \alpha \end{tikzcd}$$

4.15 SIZE OVERRIDES

Newly introduced in version 3 of UnicodeMath, "[t]he inverted F character Ⅎ ([U+2132 TURNED CAPITAL F]) followed by various ASCII characters changes the 'font' of the text. For example, `a_ℲA2` builds up as a_2 in contrast to `a_2`, which builds up as a_2." [Sargent16]

Sargent lists four flags: `A` for increasing the font size by one increment, `B` for increasing it by two increments, `C` for decreasing it by one increment, and `D` for decreasing it by two increments.

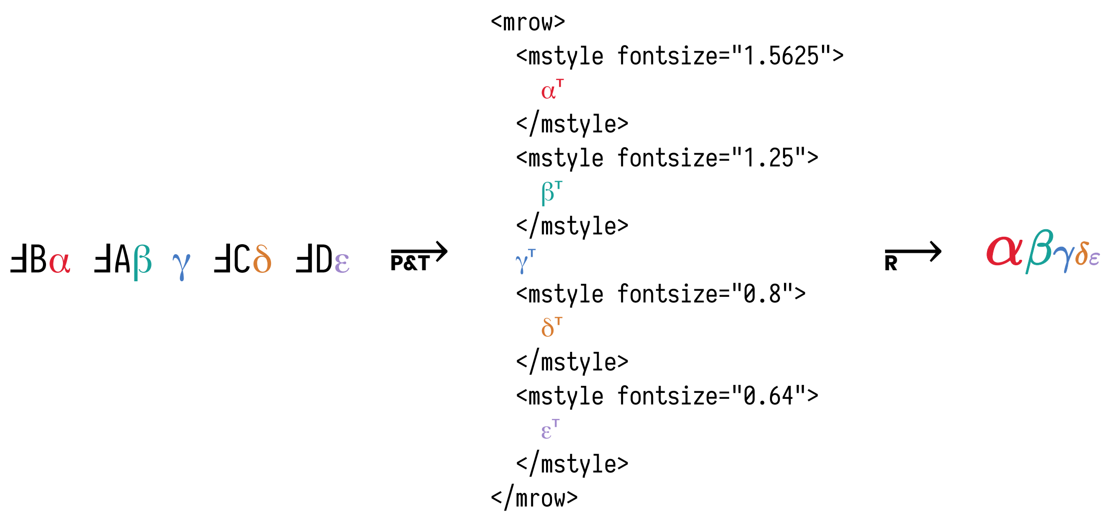
My initial implementation of this feature worked as shown below – `<mstyle>`'s `scriptlevel` attribute which "[c]ontrols mostly the font-size [and] accepts a non-negative integer, as well as a '+' or a '-' sign, which increments or decrements the current value", according[1] to the MDN web docs.

$$\begin{tikzcd} \text{ℲB}\alpha \arrow[r, "\mathsf{P}"'] & \begin{array}{l}\textbf{sizeoverride}\text{:}\\ \quad\textbf{size}\text{: } \texttt{"B"}\\ \quad\textbf{of}\text{: } \alpha^{\mathsf{P}}\end{array} \arrow[r, "\mathsf{T}"'] & \begin{array}{l}\texttt{<mstyle scriptlevel="-">}\\ \quad\texttt{<mstyle scriptlevel="-">}\\ \qquad\alpha^{\mathsf{T}}\\ \quad\texttt{</mstyle>}\\ \texttt{</mstyle>}\end{array} \arrow[r, "\mathsf{R}"'] & \alpha \end{tikzcd}$$

Increasing the script level implies smaller text, so the script level is decreased twice here to increase the font size by two increments. However, while semantically sensible, it turns out that this feature is not implemented in any MathML renderer I'm aware of. So instead, I set `<mstyle>`'s `fontsize` attribute to 1.25^n `em`, with n being the number of increments by which the font size should be increased (negative values result in a decrease by n increments). The constant 1.25 has been determined empirically. Note that this scaling formula is reused for resizing delimiters (see Section 4.20).

This approach works perfectly in Safari and Firefox, but merely well in MathJax, where it does not behave in a cumulative fashion, *i.e.*, `ℲBℲBα` does not yield a larger font than `ℲBα`.

[1] See `https://developer.mozilla.org/en-US/docs/Web/MathML/Element/mstyle`.



```
<mrow>
  <mstyle fontsize="1.5625">
    α^T
  </mstyle>
  <mstyle fontsize="1.25">
    β^T
  </mstyle>
  γ^T
  <mstyle fontsize="0.8">
    δ^T
  </mstyle>
  <mstyle fontsize="0.64">
    ε^T
  </mstyle>
</mrow>
```

4.16 NON-STANDARD EXTENSIONS: COLOR, COMMENTS & TYPEWRITER FONT

In Section 1 of the tech note, Sargent writes that UnicodeMath, as implemented in Microsoft's products, "delegates some rich-text properties like text and background colors, [...] comments, [...] etc., to a higher layer." [Sargent16] Although this works well in the context he envisions, it's impossible when integrating UnicodeMath into Markdeep or HTML – there simply is no suitable higher layer.

To address this issue, four additional, non-standard constructs were introduced: a method for setting a subexpression's foreground and background colors, a notation for comments, and a way of writing text in a typewriter font[1]. None of them are implemented in a particularly interesting way, so I won't go into as much detail here as I do in other sections.

Text colors are denoted as `✎(color&α)` in UnicodeMath code, where `color` is any color name or notation legal in CSS – this laissez-faire approach seems to work just fine in Safari, Firefox and with MathJax. For example, to display the number 6 in rebeccapurple[2], one would write `✎(rebeccapurple&6)`, which renders as 6.

Background colors work analogously, except with ☁ (U+2601 CLOUD) in place of U+270E LOWER RIGHT PENCIL. Text colors and background colors can be combined, of course.

Comments are implemented such that text between ⫷ (U+2AF7 TRIPLE NESTED LESS-THAN) and ⫸ (U+2AF8 TRIPLE NESTED GREATER-THAN) is parsed in much the same way that "text" is, but removed during the translation step.

Typewriter text, denoted by `ｔ(α)` (read ｔ as "tt", even though it's actually U+FF07 HALFWIDTH

[1] Typewrite-style variants of lowercase and uppercase Latin characters exist in Unicode, but that's it. Meanwhile, a *full* typewriter font is available on nearly every system.

[2] See `https://meyerweb.com/eric/thoughts/2014/06/19/rebeccapurple/` – but go get a box of tissues first.

HANGUL LETTER YU), is rendered in the system's default type writer font. This is accomplished by wrapping the text in an `<mstyle` element as follows:

$$\pi(\alpha) \xrightarrow[\mathbf{P}]{} \mathbf{tt}\!:\alpha \xrightarrow[\mathbf{T}]{} \begin{array}{l}\texttt{<mstyle fontfamily="monospace">}\\ \quad\texttt{<mtext>}\\ \qquad\alpha\\ \quad\texttt{</mtext>}\\ \texttt{</mstyle>}\end{array} \xrightarrow[\mathbf{R}]{} \alpha$$

Typewriter text is useful when formulating relational algebra queries[1], for example:

$$\pi_{\texttt{X}\leftarrow\texttt{A}+\texttt{C},\ \texttt{Y}\leftarrow\neg\texttt{B},\ \texttt{Z}\leftarrow\texttt{"LEGO"}}(R)$$

$$\pi_{\texttt{from},\ \texttt{to}\leftarrow\texttt{to2}}\left(\sigma_{\texttt{to}=\texttt{from2}}\left(G\times\pi_{\texttt{from2}\leftarrow\texttt{from},\ \texttt{to2}\leftarrow\texttt{to}}(G)\right)\right)$$

4.17 PRIMES

Primes, similar to scripts, can follow any **entity** or operator. They can be entered by typing

- the ASCII apostrophe `'` (U+0027 APOSTROPHE),
- ′ (U+2032 PRIME),
- ″ (U+2033 DOUBLE PRIME),
- ‴ (U+2034 TRIPLE PRIME),
- ⁗ (U+2057 QUADRUPLE PRIME),

or any combination of the above. During parsing, the primes are counted in a semantic action so that during the translation[2] to MathML,

- if a single prime is present, U+2032 PRIME is emitted in an `<msup>` attached to the prime base (or prepended to a previously-present superscript as described in Section 3.10 of the tech note: “It's [...] important to merge the prime into a superscript that follows, *e.g.*, `a'^c` should display as a'^{c}, where both the prime and the `c` are in the same superscript argument.” [Sargent16]);
- if two primes were given, U+2033 DOUBLE PRIME is emitted in the same manner;
- for three primes, U+2034 TRIPLE PRIME is emitted;
- for four primes, U+2057 QUADRUPLE PRIME is emitted; and
- for more primes, U+2032 PRIME is repeated the appropriate number of times.

Consider the following two examples.

[1] See `https://db.inf.uni-tuebingen.de/staticfiles/teaching/ws1718/DB1/slides/DB1-slides-12.pdf`.

[2] In Section 4.1 of the tech note, Sargent mentions that “for proper typography, the prime should have a large glyph variant that when superscripted looks correct” [Sargent16] – I was unable to find such a variant encoded in Unicode. However, at least Firefox's built-in MathML renderer and MathJax seem to use such a variant in the correct places.

α'' $\xrightarrow{P}$

```
primed:
  base: α^P
  primes: 2
```

$\xrightarrow{T}$

```
<msup>
  α^T
  <mo>″</mo>
</msup>
```

$\xrightarrow{R}$ α''

α'₂^β $\xrightarrow{P}$

```
script:
  type: "subsup"
  base:
    primed:
      base: α^P
      primes: 1
  low:
    expr:
      number: "2"
  high: β^P
```

$\xrightarrow{T}$

```
<msubsup>
  α^T
  <mn>2</mn>
  <mrow>
    <mo>′</mo>
    β^T
  </mrow>
</msubsup>
```

$\xrightarrow{R}$ $\alpha_2^{\prime\beta}$

4.18 FACTORIALS

An **entity** followed by an exclamation mark is recognized as a factorial. If instead followed by two exclamation marks ‼ (U+203C DOUBLE EXCLAMATION MARK), it is recognized as a semifactorial[1], which I've implemented as syntactic sugar for two nested factorials:

α! + β‼ $\xrightarrow{P}$

```
expr: ⟦
  factorial: α^T
  factorial:
    factorial: β^T
⟧
```

$\xrightarrow{T}$

```
<mrow>
  <mrow>
    α^T
    <mo>!</mo>
  </mrow>
  <mo>+</mo>
  <mrow>
    <mrow>
      β^T
      <mo>!</mo>
    </mrow>
    <mo>!</mo>
  </mrow>
</mrow>
```

$\xrightarrow{R}$ $\alpha! + \beta!!$

[1] See https://en.wikipedia.org/wiki/Double_factorial.

4.19 ATOMS

Atoms were once considered to be the fundamental building blocks of the universe. (Dis-)Similarly, in UnicodeMath, an **atom** is

- a Latin or Greek letter[1],
- a character from what I call a "math font"[2] – math fonts consist of 𝐯𝑎𝒓𝗂𝐨𝒖𝒔 𝓋𝓪𝖗𝖎𝗮𝕟𝐭𝒔 of Latin and Greek letters as well as numbers, most of which are located in Unicode's *Mathematical Alphanumeric Symbols* block,
- an emoji,
- one or more space characters (see rule **mathspaces**), or
- a diacriticized letter, digit or delimited expression.

A semantic action attached to the **atoms** rule merges adjacent letters into words – for example, parsing `abc` creates an intermediate list `[{"char": "a"}, {"char": "b"}, {"char": "c"}]` which is transformed into `{"atoms": {"chars": "abc"}}`. Merging stops upon encountering a math space or a diacriticized expression, and resumes afterwards – for example, parsing `abc42` yields `{"atoms": [{"chars": "abc"}, {"number": "42"}]}`.

During the translation step, stand-alone Latin and Greek letters are converted to italicized variants[3] from the *Mathematical Alphanumeric Symbols* block. For example, `a` is turned into a, but `abc` remains "unsullied"[4] on the assumption that it represents a word or function name. If this is not the desired interpretation, the writer may intersperse characters with U+2061 FUNCTION APPLICATION, U+2062 INVISIBLE TIMES, U+2063 INVISIBLE SEPARATOR or U+2064 INVISIBLE PLUS depending on indented semantics (these invisible code points are picked up[5] by screen-reading software, thus aiding accessibility).

Diacriticized expressions, detailed in Section 3.10 of the tech note, are ones that are

---

[1] As noted in a comment within the grammar, this set should include any Unicode character that's in the L∗ categories: "Lowercase Letter" (Ll, 2151 code points), "Modifier Letter" (Lm, 259 code points), "Other Letter" (Lo, 121414 code points), "Titlecase Letter" (Lt, 31 code points), and "Uppercase Letter" (Lu, 1788 code points). However, since JavaScript's regex engine does not support matching on Unicode categories – others do, see `https://stackoverflow.com/questions/1832893/python-regex-matching-unicode-properties` – this would be difficult to implement. As a result, I've made the choice to only support Latin and Greek letters for now. See also: `https://stackoverflow.com/questions/280712/javascript-unicode-regexes`

[2] These characters pass through the UnicodeMathML pipeline unchanged. It would be conceivable to convert them to their ASCII counterparts and use MathML's `mathvariant` attribute (see `https://developer.mozilla.org/en-US/docs/Web/MathML/Element/mi#Attributes` for more information) to switch fonts, however this is not supported by all MathML renderers. Anyhow, "[t]he `mathvariant` attribute was added to MathML before the Unicode math alphanumerics were encoded in Unicode 3.1.0 (March, 2001). But now it's only needed for reading existing documents that contain it." [Sargent19]

[3] Section 4.1 of the tech note: "[i]t is easier to type ASCII letters than italic letters, but when used as mathematical variables, such letters are traditionally italicized in print. Accordingly a user might want to make italic the default alphabet in a math context, reserving the right to overrule this default when necessary. A more elegant approach in math zones is to translate letters deemed to be standalone to the appropriate math alphabetic characters […]." [Sargent16]

[4] Not in the *Game of Thrones* sense.

[5] See `https://www.w3.org/TR/MathML2/chapter2.html`.

succeeded by a character from Unicode's "Combining Diacritical Marks" or "Combining Diacritical Marks for Symbols" blocks. In plain text, these diacritical marks combine with their predecessor character into a new character, *e.g.*, an a followed by U+0302 COMBINING CIRCUMFLEX ACCENT becomes â. In MathML, this can be replicated by wrapping base and diacritic in an `<mover accent="true">` tag[1]. For example, the Fourier transform

$$\widehat{f}(\xi)=\int_{-\infty}^{\infty} f(x)e^{-2\pi ix\xi}\,dx$$

features a diacriticized f. Similarly, Newton's notation for differentiation[2] heavily relies on diacritical dots (which some MathML renderers don't align neatly):

$$\textbf{y's fifth derivative} = \overset{5}{\dot{y}} = \overset{\cdot\cdot\cdot\cdot\cdot}{y} = \overset{\vdots\,\vdots}{y} = \overset{\vdots\,\vdots}{y}$$

Some diacritics can stretch to fit on top of multiple characters at a time. For example, `(a+b)&nbsp;̂` renders as $\widehat{a+b}$.

Numbers[3] – optionally with a decimal dot or comma – are related to atoms in that they can occur in the same contexts, as are some special double-struck letters detailed in Section 3.11 of the tech note:

- ⅆ encodes the letter "d" as used in Leibniz's differential notation[4] – regarding the example below, Sargent notes in Section 3.4 "that the ⅆ character automatically introduces a small space between the x and the dx and by default displays as a math-italic d when it appears in a math zone." [Sargent16]

$$\frac{\int_0^a x\,dx}{x^2+a^2}$$

- Analogously, ⅅ is used in Euler's differential notation[5].
- ⅇ encodes the natural exponent, *i.e.*, Euler's number – it renders as e.
- ⅈ and ⅉ encode the imaginary unit, rendering as i and j.

Note that "[i]n US patent applications these characters should be rendered as ⅆ, ⅅ, ⅇ, ⅈ, ⅉ as defined, but in regular US technical publications, these quantities can be rendered as math italic" [Sargent16], which is how I've implemented it. "In European technical publications, they are sometimes rendered as upright characters. [...] [T]he display routines should provide the appropriate glyphs and spacings." [Sargent16] Perhaps it would be useful

1 Note that not all diacritics should display above their base – some go below it (which I've implemented with `<munder>`), either directly or separated with a small space, other are superimposed over the base. A Unicode Technical Note outlines a general rendering algorithm for combining marks: `https://www.unicode.org/notes/tn2/`

2 See `https://en.wikipedia.org/wiki/Notation_for_differentiation#Newton's_notation`.

3 Analogously to letters, anything from Unicode's "Decimal Number" (Nd, 630 code points) category should be supported here – my implementation only supports `[0-9]` for the reason detailed five footnotes ago.

4 See `https://en.wikipedia.org/wiki/Notation_for_differentiation#Leibniz's_notation`.

5 See `https://en.wikipedia.org/wiki/Notation_for_differentiation#Euler's_notation`.

to let the user configure this in the future (see Section 8.1).

Here's a number of examples demonstrating how atoms, numbers and double-struck symbols are translated into MathML:

`abc+a` $\underset{P}{\longrightarrow}$

```
expr: ⟦
  atoms:
    chars: "abc"
  operator: "+"
  atoms:
    chars: "a"
  ⟧
```

$\underset{T}{\longrightarrow}$

```
<mrow>
  <mi>abc</mi>
  <mo>+</mo>
  <mi>a</mi>
</mrow>
```

$\underset{R}{\longrightarrow}$ $\mathrm{abc} + a$

`30-50🐗` $\underset{P}{\longrightarrow}$

```
expr: ⟦
  number: "30"
  operator: "-"
  ⟦
    number: "50"
    atoms:
      chars: "🐗 "
    ⟧
  ⟧
```

$\underset{T}{\longrightarrow}$

```
<mrow>
  <mn>30</mn>
  <mo>-</mo>
  <mrow>
    <mn>50</mn>
    <mi>🐗 </mi>
  </mrow>
</mrow>
```

$\underset{R}{\longrightarrow}$ $30 - 50$🐗

$\dot{\ddot{\alpha}}$ $\underset{P}{\longrightarrow}$

```
atoms: ⟦
  diacriticized:
    base: αᴾ
    diacritics: ⟦"¨", "˙"⟧
  ⟧
```

$\underset{T}{\longrightarrow}$

```
<mover accent="true">
  <mover accent="true">
    αᵀ
    <mo>&#776;</mo>
  </mover>
  <mo>&#775;</mo>
</mover>
```

$\underset{R}{\longrightarrow}$ $\dot{\ddot{\alpha}}$

αⅆβ $\underset{P}{\longrightarrow}$

```
expr: ⟦
  αᴾ
  doublestruck: "ⅆ"
  βᴾ
  ⟧
```

$\underset{T}{\longrightarrow}$

```
<mrow>
  αᵀ
  <mrow>
    <mspace width="thinmathspace" />
    <mi>d</mi>
  </mrow>
  βᵀ
</mrow>
```

$\underset{R}{\longrightarrow}$ $\alpha\, d\beta$

4.20 DELIMITERS & GROUPING

The missing UnicodeMath construct is, of course, delimiters. I've already mentioned situations in which parentheses are removed during translation to MathML, but not yet how they come into being in the first place. The **expBracket** grammar rule is used to parse

- absolute values: expressions delimited by the standard pipe symbol |,
- vector norms: expressions delimited by either two pipes or ‖ (U+2016 DOUBLE VERTICAL LINE),
- expressions enclosed within an opening bracket and a closing bracket, and
- multiple-case expressions.

The first two are notable since the opening and closing delimiters are identical for them, which makes parsing of nested absolute values or norms tricky or – if nesting makes an expression ambiguous – impossible. In Section 3.1 of the tech note, Sargent notes that "[n]ested absolute values can be handled unambiguously by discarding the outermost parentheses within an absolute value. For example, the [...] expression $||\boldsymbol{x}|-|\boldsymbol{y}||$ can have the UnicodeMath `|(|x|-|y|)|` [and] the example $|\boldsymbol{a}|\boldsymbol{b}-\boldsymbol{c}|\boldsymbol{d}|$ needs the clarifying parentheses since it can be interpreted as either `(|a|b)-(c|d|)` or `|a(|b-c|)d|`." [Sargent16]

Note that under certain circumstances, the pipe symbol can also be used as a normal opening and closing delimiter. Quoting from Section 3.1 of the tech note, "[t]his handles the important case of the bra vector ⟨ | in Dirac notation. For example, the quantum mechanical density operator ρ has the definition

$$\rho = \sum\nolimits_{\psi} P_{\psi} |\psi\rangle\langle\psi|,$$

where the vertical bars can be input using the [pipe symbol]." [Sargent16]

Switching gears to the third point above, opening delimiters include

- ((U+0028 LEFT PARENTHESIS),
- [(U+005B LEFT SQUARE BRACKET),
- { (U+007B LEFT CURLY BRACKET),
- ⟨ (U+27E8 MATHEMATICAL LEFT ANGLE BRACKET),
- ⌈ (U+2308 LEFT CEILING),
- ⌊ (U+230A LEFT FLOOR),

and closing delimiters include

-) (U+0029 RIGHT PARENTHESIS),
-] (U+005D RIGHT SQUARE BRACKET),
- } (U+007D RIGHT CURLY BRACKET),
- ⟩ (U+27E9 MATHEMATICAL RIGHT ANGLE BRACKET),
- ⌉ (U+2309 RIGHT CEILING), and
- ⌋ (U+230B RIGHT FLOOR).

These delimiters can be paired arbitrarily and they adjust to the size of their contents automatically, thanks to MathML's `<mfenced>` element[1]. As an example, the utterly non-sensical UnicodeMath expression `{a⌋^⟨1/[2)/3]` renders as

$$\{a\rfloor^{\left\langle\frac{\frac{1}{[2)}}{3}\right]}.$$

If a closing delimiter is preceded by ├ (U+251C BOX DRAWINGS LIGHT VERTICAL AND RIGHT), it can be used in a place where an opening delimiter would be expected. Analogously, an opening delimiter after ┤ (U+2524 BOX DRAWINGS LIGHT VERTICAL AND LEFT) can close an expression. If placed between one of these markers and the associated delimiter, a positive integer allows for manual sizing of the delimiter – this reuses the font size adjustment function introduced in Section 4.15. For example, `├3(├1((a)┤1)┤3) /= (((a)))` is rendered as

$$\Bigg(\bigg((a)\bigg)\Bigg) \neq (((a))).$$

Moreover, the special brackets 〖 (U+3016 LEFT WHITE LENTICULAR BRACKET) and 〗 (U+3017 RIGHT WHITE LENTICULAR BRACKET) become invisible during translation to MathML. If they occur together, a grouping `<mrow>` is generated in their place, but each of them can also be paired with a visible delimiter. In conjunction with an equation array, this can be used to replicate LaTeX's `cases` environment:

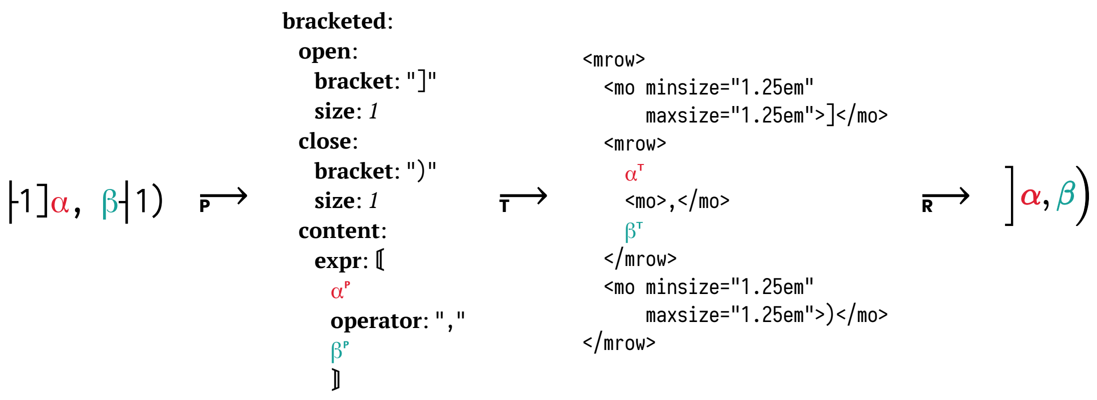
$$|x| = \begin{cases} x & \text{if } x \geq 0 \\ -x & \text{if } x < 0 \end{cases}$$

The same expression can – with a pinch of syntactic sugar added in UnicodeMath version 3 – be written as `|x|=Ⓒ(&x&"if "x≥0@-&x&"if "x<0)` instead of `|x|={█(&x&"if "x≥0@-&x&"if "x<0)〗`.

Here's how a few examples of delimited expressions make it though UnicodeMathML:

$$(\alpha) \xrightarrow[P]{} \begin{array}{l} \textbf{bracketed}: \\ \quad \textbf{open}: \texttt{"("} \\ \quad \textbf{close}: \texttt{")"} \\ \quad \textbf{content}: \alpha^{P} \end{array} \xrightarrow[T]{} \begin{array}{l} \texttt{<mfenced open="("} \\ \qquad\qquad \texttt{close=")">} \\ \quad \alpha^{T} \\ \texttt{</mfenced>} \end{array} \xrightarrow[R]{} (\alpha)$$

[1] See `https://developer.mozilla.org/en-US/docs/Web/MathML/Element/mfenced`.



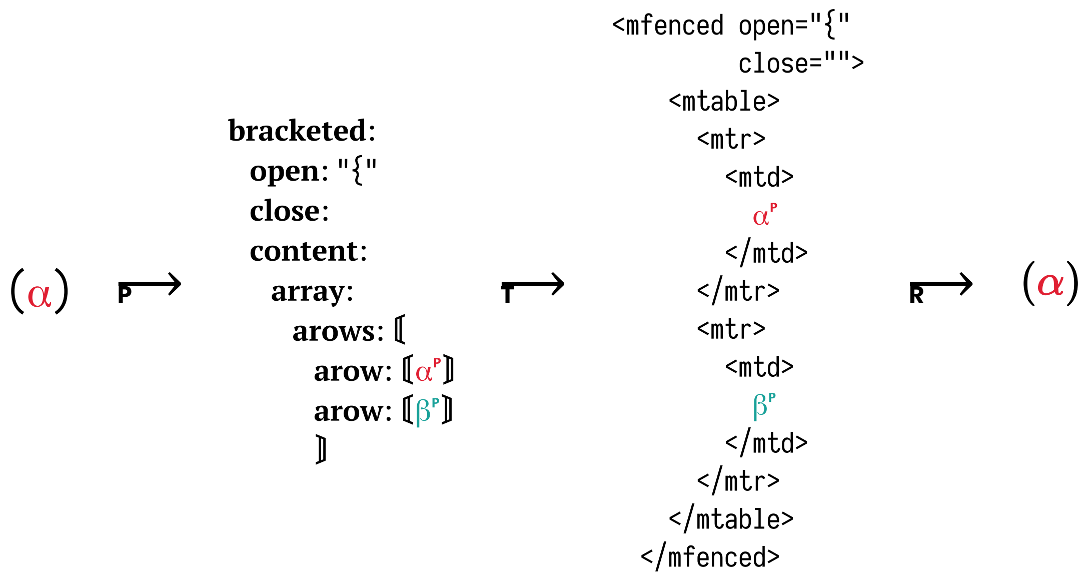
```
├1]α, β┤1)  ─P→

bracketed:
  open:
    bracket: "]"
    size: 1
  close:
    bracket: ")"
    size: 1
  content:
    expr: ⟦
      αᴾ
      operator: ","
      βᴾ
      ⟧

─T→

<mrow>
  <mo minsize="1.25em"
      maxsize="1.25em">]</mo>
  <mrow>
    αᵀ
    <mo>,</mo>
    βᵀ
  </mrow>
  <mo minsize="1.25em"
      maxsize="1.25em">)</mo>
</mrow>

─R→  ]α,β)
```



```
(α)  ─P→

bracketed:
  open: "{"
  close:
  content:
    array:
      arows: ⟦
        arow: ⟦αᴾ⟧
        arow: ⟦βᴾ⟧
      ⟧

─T→

<mfenced open="{"
         close="">
  <mtable>
    <mtr>
      <mtd>
        αᴾ
      </mtd>
    </mtr>
    <mtr>
      <mtd>
        βᴾ
      </mtd>
    </mtr>
  </mtable>
</mfenced>

─R→  (α)
```

Note that “delimiters can also have separators within them” [Sargent16], which are discussed in Section 3.1 of the tech note and match up with the `<mfenced separator="">` attribute: UnicodeMathML supports │ (U+2502 BOX DRAWINGS LIGHT VERTICAL) for expressions like $(a|b|c)$ and ∣ (U+2223 DIVIDES) for expressions like $\{x \mid f(x) = 0\}$. Also, other operators can be treated as separators by preceding them with ║ (U+2551 BOX DRAWINGS DOUBLE VERTICAL), which I haven't implemented because it would 1. complicate parsing of operators, 2. make no visual difference and due to 3. the `<mfenced>` element' recent (see Section 8.1) deprecation.

5

INTEGRATION INTO MARKDEEP AND HTML

Note: At the time of writing, Markdeep 1.06 was the most recent release. Any `diffs` below are with respect to that version.[1] *However, I do not expect the integration approaches described below to require more-than-superficial changes in any future versions of Markdeep.*

The tech note includes a section on recognizing mathematical expressions based on heuristics such as the presence of math symbols. "The basic idea is that math characters identify themselves as such and potentially identify their surrounding characters as math characters as well." [Sargent16] Some of these heuristics seem a bit dubious: "Specifically ASCII letter pairs surrounded by whitespace are often mathematical expressions, and as such should be italicized in print. If a letter pair fails to appear in a list of common English and European two-letter words, it is treated as a mathematical expression and italicized." [Sargent16]

I, however, expect that from the perspective of a user who is not familiar with these specific heuristics, such behavior would likely appear inscrutable at best and infuriating at worst – perhaps it's acceptable in a WYSIWYG environment, but surely not in a context where UnicodeMath expressions are included in a plain-text document and converted into a built-up form only when the document itself is rendered.

[1] Obtainable from `https://casual-effects.com/markdeep/1.06/markdeep.js` and additionally kept at `/code/markdeep-integration/markdeep-1.06-orig.js`.

Luckily, the tech note also defines opening and closing delimiters for UnicodeMath: “Note that if explicit math-zone-on and math-zone-off characters are desired, [...] ⁅ ([U+2045 LEFT SQUARE BRACKET WITH QUILL]) starts a math zone and ⁆ ([U+2046 RIGHT SQUARE BRACKET WITH QUILL]) ends it.” [Sargent16]

I've opted to implement math zone extraction purely based on these delimiters instead of heuristically.

5.1 NAÏVE MARKDEEP INTEGRATION

A quick-and-dirty way to integrate UnicodeMathML into Markdeep is to

1. generate a standalone version of the parser by opening `/code/utils/generate-parser.html` in any browser, upon which a file `unicodemathml-parser.js` will show up wherever that browser stores downloads, and include this file in a Markdeep document using a `<script>` tag ahead of the Markdeep loading code,
2. load `/code/src/unicodemathml.js` in the same manner, and
3. apply the following patch to Markdeep, yielding `/code/markdeep-integration/markdeep-1.06-basic.js`:

```
@@ -2529,6 +2529,13 @@
     // Protect raw HTML attributes from processing
     str = str.rp(/(<\w[^ \n<>]*?[ \t]+)(.*?)(?=\/?>)/g, protectorWithPrefix);

+    // Convert UnicodeMath expressions to MathML
+    if (typeof unicodemathml === "function") {
+        str = str.rp(/⁅([^⁆]*?)⁆/gi, function (unicodemathWithDelimiters, unicodemath) {
+            return protect(unicodemathml(unicodemath).mathml);
+        });
+    }
+
     // End of processing literal blocks
     //////////////////////////////////////////////////////////////////////////
```

The location of the inserted code is no accident. Preceding the UnicodeMathML conversion, HTML tags such as `<style>`, `<code>`, `<svg>`, and their contents are protected from further processing. This means, for example, that UnicodeMath expressions enclosed in `<code>` will not be processed. A similar step is required for `<math>` tags generated by UnicodeMathML: Markdeep's `protect` function excludes its argument string from further processing by Markdeep, preventing issues when math expressions happen to contain valid Markdown code.

This approach has several advantages, most notably its simplicity:

- It is minimally invasive with respect to Markdeep and requires no extra code apart from two `<script>` tags.
- Because Markdeep's MathJax configuration enables MathML display, this yields the desired result even in browsers that don't natively support MathML.
- With regard to *measured* performance, it is about as fast as any integration can

possibly be (since neither DOM manipulation nor any other expensive overhead is involved).

However, there is one deal-breaking disadvantage, along with a few comparatively minor nicks:

- *Perceived* performance is severely lacking. This is because UnicodeMathML bars Markdeep from proceeding while the translation is in progress, *i.e.*, the user will be faced with a blank page until the translation of all UnicodeMath expressions (and the rest of Markdeep's processing) is finished. Although this is barely noticeable for small documents, larger documents may remain in this state for several seconds, which, apart from being inconvenient, may be somewhat disconcerting for the user.
- UnicodeMath expressions are translated before Markdown is converted to HTML, so it is not straightforwardly possible to determine whether to present an expression in `displaystyle` or `textstyle` (recall Section 3.20 of the tech note, which specifies that "if a math zone fills a (hard or soft) paragraph, the math zone is a display math zone [and] [i]f any part of the paragraph isn't in a math zone [...], then the math zone is an inline math zone, which has more compact rendering." [Sargent16]).
- The translation is *pure*, so UnicodeMath expressions that occur n times in a document only need to be translated once: the use of a cache would enable this.
- Some minor points and edge cases are not addressed, notably escaping of UnicodeMath delimiters (Section 3.20 of the tech note, for example, `\⁅a+b⁆` should be displayed as \⁅a+b⁆, not \\$a + b$).

5.2 BETTER MARKDEEP INTEGRATION

For these reasons, the final integration of UnicodeMathML into Markdeep is a two-step process composed of *marking* UnicodeMath zones while Markdeep processes the document and then, once the page has loaded, *translating and rendering* whatever has been marked. I will now discuss this approach, which is implemented in ☁`/code/markdeep-integration/unicodemathml-integration.js` and requires a patched variant of Markdeep, see ☁`/code/markdeep-integration/markdeep-1.06.js` (or below). Note that MathJax[1] and other JavaScript-based math renderers function similarly.

The **marking** step – implemented by the `markUnicodemathInHtmlCode` function – is mainly required to protect math zones from Markdeep processing, with the secondary benefit of making them easy to track down in the second step.

Compared to the naïve integration, a more complex regular expression is used to extract UnicodeMath zones from the document. Since Markdeep appears to convert every occurrence of U+00A0 NO-BREAK SPACE into a character entity `&nbsp;`, this mapping is reversed in order

[1] Quoting from MathJax's documentation: "The support for TeX and LaTeX in MathJax involves two functions: the first looks for mathematics within your web page (indicated by math delimiters like `$$...$$`) and marks the mathematics for later processing by MathJax, and the second is what converts the TeX notation into MathJax's internal format, where one of MathJax's output processors then displays it in the web page." See `http://docs.mathjax.org/en/latest/input/tex/index.html`.

to avoid issues when parsing UnicodeMath expressions containing non-breaking spaces. Then, a <span> element is created as follows:

```
                     <span class="unicodemathml-placeholder"
                           data-attr="encodeURIComponent(α)">
      ⁅α⁆   ⟶           ⁅α⁆
                     </span>
```

The data-unicodemath attribute is required 1. to avoid having to remove the UnicodeMath delimiters in the following step, and 2. to circumvent an implementation-specific encoding issue.

The second **translating and rendering** step – implemented by the renderMarkedUnicodemath function – is executed immediately prior to execution of Markdeep's onLoad hook. After

- initializing an on-screen progress meter,
- setting up a cache (used to avoid translating identical UnicodeMath expressions more than once) and
- defining CSS rules for parse errors (to color any such expression red and hide the likely-verbose error message unless the user hovers over the expression – as an example, the fraction ⁅a/⁆ is missing its denominator),

it iterates through all DOM elements matching the span.unicodemathml-placeholder selector, performing the following actions for each:

1. The UnicodeMath expression stored in the data-unicodemath attribute is decoded using decodeURIComponent.
2. A look at sibling nodes and the parent nodes determines whether to present the expression in displaystyle or textstyle.
3. If present in the cache, a previous translation of the expression is inserted into the DOM in place of the <span> element. Otherwise, the expression is translated from scratch via a call of unicodemathml, cached, and substituted as above.[1]
4. The progress meter is updated.[2]

Once every expression has been handled in this way, the progress meter is hidden and MathJax renders the new MathML elements.

[1] Note that this substitution is responsible for most of the overhead compared to the naïve integration – repeated DOM manipulations are simply slower than string operations. I considered batching the DOM manipulations with the goal of reducing the frequency at which browsers need to recalculate the page layout and paint any changes (which, in Chrome, accounts for up to 30% of the load time), but ultimately decided against this for simplicity's sake – for now, anyway (see Section 8.3).

[2] Although it sounds trivial, regularly updating the progress meter was a bit tricky to figure out – browsers avoid redrawing content *while* a JavaScript function is running, forcing me to use JavaScript's async/await mechanism to effectively CPS transform the renderMarkedUnicodemath function such that a tail call to requestAnimationFrame is performed after translation of each expression.

This two-step approach does not substantially increase the degree to which Markdeep needs to be patched. In surgical terms, three small incisions suffice:

```
@@ -2529,6 +2529,11 @@
     // Protect raw HTML attributes from processing
     str = str.rp(/(<\w[^ \n<>]*?[ \t]+)(.*?)(?=\/?>)/g, protectorWithPrefix);

+    // Mark UnicodeMath expressions for later conversion to MathML
+    if (typeof umml !== "undefined" && umml) {
+        str = markUnicodemathInHtmlCode(str, protect);
+    }
+
     // End of processing literal blocks
     ///////////////////////////////////////////////////////////////////////////////

@@ -4520,9 +4525,10 @@
     }

     function needsMathJax(html) {
-        // Need MathJax if $$ ... $$, \( ... \), or \begin{
+        // Need MathJax if UnicodeMath to MathML translator present, $$ ... $$, \( ... \), or \begin{
         return option('detectMath') &&
-            ((html.search(/(?:\$\$[\s\S]+\$\$)|(?:\\begin{)/m) !== -1) ||
+            ((typeof umml !== "undefined" && umml) ||
+             (html.search(/(?:\$\$[\s\S]+\$\$)|(?:\\begin{)/m) !== -1) ||
             (html.search(/\\\(.*\\\)/) !== -1));
     }

@@ -4657,6 +4663,11 @@

         document.body.style.visibility = 'visible';

+        // Kick off UnicodeMath to MathML translation asynchronously
+        if (typeof umml !== "undefined" && umml) {
+            setTimeout(renderMarkedUnicodemath, 0);
+        }
+
         var hashIndex = window.location.href.indexOf('#');
         if (hashIndex > -1) {
             // Scroll to the target; needed when loading is too fast (ironically)
```

Note that the boolean variable `umml` is set *iff* the integration itself is present, and true *iff* both the parser and the translation function are present as well.

The user's `.md.html` document, then, needs to be structured as shown below.[1] The `unicodemathmlOptions` variable is used to configure some aspects of the UnicodeMathML integration – it can, however, be omitted if the default values, shown here, are desired.

```
<meta charset="utf-8">

Document content and ⁅m+a/t_h⁆ goes here.
```

[1] If such a document were made available on the web, concatenating the four JavaScript files would be advisable in order to reduce the number of HTTP requests. Locally, especially during development of UnicodeMathML, it's more convenient to keep them separate, though.

```
<script>
    var unicodemathmlOptions = {
        resolveControlWords: false,  // resolve control words, e.g., convert \int to ∫?
        showProgress: true,          // show progress meter in the bottom right corner?
        before: Function.prototype,  // hook executed prior to translation (no-op function)
        after: Function.prototype    // hook executed after translation
    };
</script>
<script src="unicodemathml.js"></script>
<script src="unicodemathml-parser.js"></script>
<script src="unicodemathml-integration.js"></script>
<script src="markdeep-1.06.js"></script>  <!-- patched Markdeep -->
```

5.3 HTML INTEGRATION

The main difference between the Markdeep and HTML integrations lies in the input: Markdeep provides the document as a string, so a regular expression can do the heavy lifting during the *marking* step. HTML documents, once loaded by a browser far enough for JavaScript code to be executed, instead exist in the form of a DOM tree.

The **marking** step of the HTML integration, which is implemented by the `markUnicodemathInHtmlDom` function of `/code/markdeep-integration/unicodemathml-integration.js`, thus walks the DOM starting at the `document.body` node[1]. If it encounters a non-text node[2], the function recursively processes the child nodes until it encounters a text node. Text nodes are split into UnicodeMath zones, which are replaced with a placeholder <span> in the manner explained in the previous section, and text nodes.

This approach enables full reuse of the **translating and rendering** step originally written with the Markdeep integration in mind.

Since Markdeep is not present and thus cannot do it for them, the user needs to jumpstart the marking and translation process via a call to the `renderUnicodemath` function, which simply calls `markUnicodemathInHtmlDom` and `renderMarkedUnicodemath` in succession. In a nutshell, the user's `.html` document needs to include the following lines:[3]

```
<script>
    var unicodemathmlOptions = …;
</script>
<script src="../src/unicodemathml.js"></script>
<script src="unicodemathml-parser.js"></script>
<script src="unicodemathml-integration.js"></script>
<script>
    document.body.onload = renderUnicodemath();
</script>
```

[1] A different node can be passed to this function. This is handy if, for example, the document has been modified via AJAX and there's new UnicodeMath in need of being rendered.

[2] Save for nodes corresponding the the HTML tags `<pre>`, `<code>`, `<textarea>`, `<script>`, `<style>`, `<head>`, and `<title>` – the contents of these nodes examined further for obvious reasons.

[3] If browsers other than Firefox and Safari need to be supported, a MathML renderer such as MathJax must be loaded as well – if present, MathJax is automatically invoked once UnicodeMathML is done.

6

ANCILLARY WORK

In this section, I will first describe the functionality and implementation of the UnicodeMathML playground, a web app that was very helpful in developing and testing UnicodeMathML.

I will then briefly share notes on some Markdeep-adjacent tools that were built during the thesis period. While not central to the UnicodeMathML conversion, they enable the creation of this thesis document and the slide deck I will use during the thesis defense.

Finally, I will give quick explanations of various Unicode-related data transformation scripts written while implementing UnicodeMathML.

6.1 UNICODEMATHML PLAYGROUND

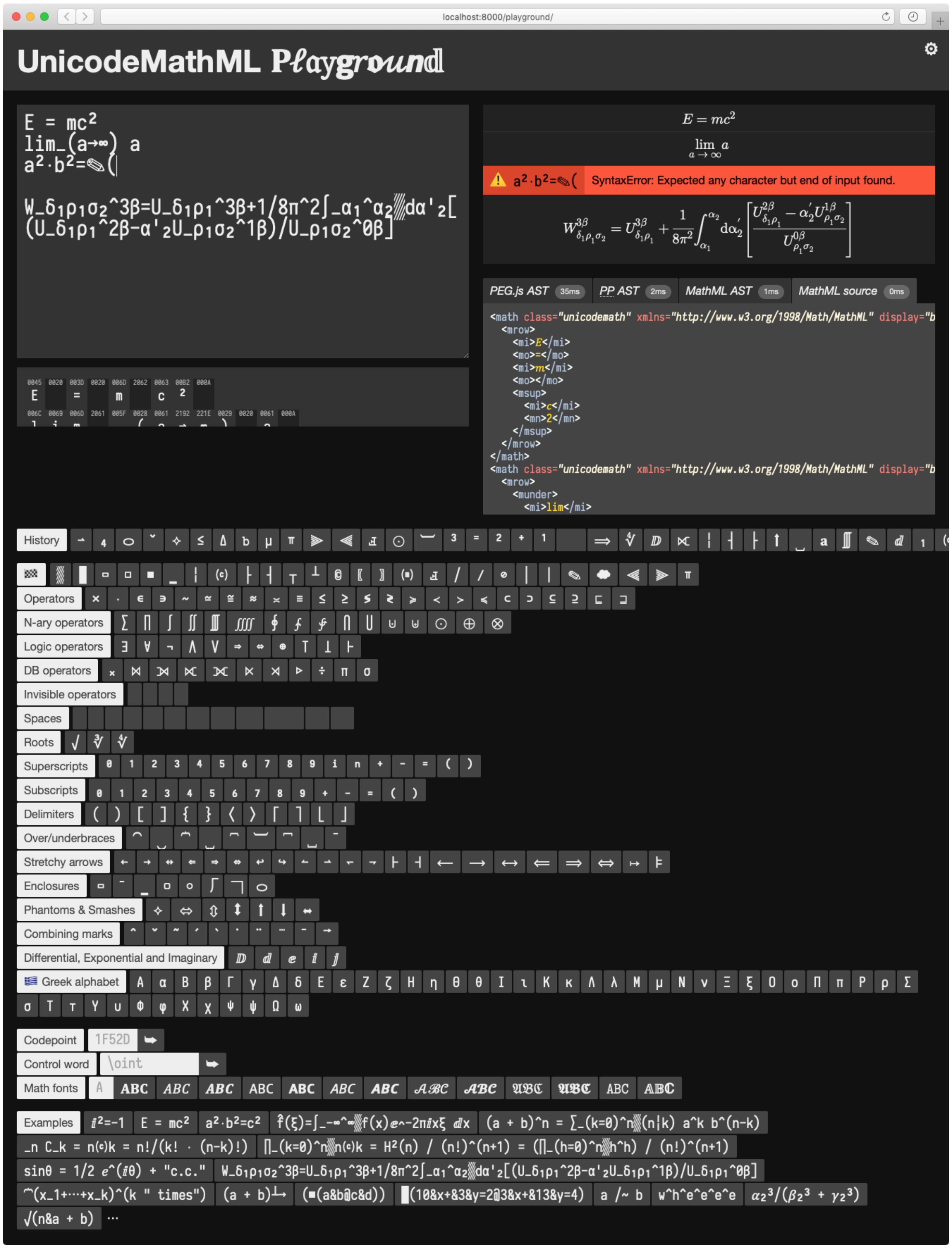
Originally indented as a parser development aid, the playground allows writing of UnicodeMath expressions with instant preview, character info, control word substitution, math font selection and other features. Its interface is shown in Figure 5 on the next page.

The playground can be found at ☁`/code/playground/`. You *might* be able to try it out by opening the contained `index.html` file in their browser – however, most browsers' implementation of a same-origin policy[1] prevents[2] this. If a Python interpreter is available,

[1] See `https://developer.mozilla.org/en-US/docs/Web/Security/Same-origin_policy`.

[2] For good reason – were this not the case, any HTML document served from the local file system could employ a snippet of JavaScript code to sneakily send arbitrary files to a remote server.

navigating to 🌰/code/ in a terminal session, running `python -m SimpleHTTPServer 8000` and pointing the browser to `localhost:8000` should do the trick.



```
E = mc²
lim_(a→∞) a
a²·b²=✎(

W_δ₁ρ₁σ₂^3β=U_δ₁ρ₁^3β+1/8π^2∫_α₁^α₂▒dα'₂[(U_δ₁ρ₁^2β-α'₂U_ρ₁σ₂^1β)/U_ρ₁σ₂^0β]
```

$E = mc^2$

$\lim_{a \to \infty} a$

⚠ a²·b²=✎(SyntaxError: Expected any character but end of input found.

$$W_{\delta_1\rho_1\sigma_2}^{3\beta} = U_{\delta_1\rho_1}^{3\beta} + \frac{1}{8\pi^2}\int_{\alpha_1}^{\alpha_2} d\alpha_2' \left[\frac{U_{\delta_1\rho_1}^{2\beta} - \alpha_2' U_{\rho_1\sigma_2}^{1\beta}}{U_{\rho_1\sigma_2}^{0\beta}}\right]$$

```
<math class="unicodemath" xmlns="http://www.w3.org/1998/Math/MathML" display="b
  <mrow>
    <mi>E</mi>
    <mo>=</mo>
    <mi>m</mi>
    <mo></mo>
    <msup>
      <mi>c</mi>
      <mn>2</mn>
    </msup>
  </mrow>
</math>
<math class="unicodemath" xmlns="http://www.w3.org/1998/Math/MathML" display="b
  <mrow>
    <munder>
      <mi>lim</mi>
```

```
ⅈ²=−1   E = mc²   a²·b²=c²   f̂(ξ)=∫_-∞^∞▒f(x)ⅇ^-2πⅈxξ ⅆx   (a + b)^n = ∑_(k=0)^n▒(n¦k) a^k b^(n-k)
_n C_k = n⒞k = n!/(k! · (n-k)!)   ∏_(k=0)^n▒n⒞k = H²(n) / (n!)^(n+1) = (∏_(h=0)^n▒h^h) / (n!)^(n+1)
sinθ = 1/2 ⅇ^(ⅈθ) + "c.c."   W_δ₁ρ₁σ₂^3β=U_δ₁ρ₁^3β+1/8π^2∫_α₁^α₂▒dα'₂[(U_δ₁ρ₁^2β-α'₂U_δ₁ρ₁^1β)/U_δ₁ρ₁^0β]
⏞(x_1+⋯+x_k)^(k " times")   (a + b)┴→   (■(a&b@c&d))   █(10&x+&3&y=2@3&x+&13&y=4)   a /~ b   w^h^e^e^e^e   α₂³/(β₂³ + γ₂³)
√(n&a + b)   ⋯
```

Figure 5: *A typical session in the UnicodeMath playground.*

6.1.1 INTERFACE

In the top left, a `<textarea>` accepts UnicodeMath expressions. After each key stroke, this input is split on newlines into multiple separately parsed and translated expressions[1]. Below the input area, the Unicode code points corresponding to the entered characters are displayed, with whitespace characters highlighted to alert the user if, for example, unintended zero-width spaces are present. This code point view in truncated by default but expands on hover. Hovering over a specific code point produces a tooltip with additional information.

The translated MathML code is inserted into an element on the right side of the page and – in browsers that don't natively support MathML – rendered by MathJax. Figure 5 showcases Safari 13's native MathML support, with math text set in a variant[2] of Donald Knuth's *Computer Modern* font.

Under this results view, a tabbed interface allows access to syntax-highlighted versions of the intermediate data structures – the *PEG.js AST* generated during the parsing[3] step, the *PP AST* that emerges after preprocessing, and the *MathML AST* resulting from the main translation step. Finally, the *MathML source* is shown in a separate tab.

The time measurement displayed in each tab represents the total time taken by the step that generated the data structure shown in that tab. Hovering over these measurements reveals expression-level timing data.

Below the two-column view, a sort of on-screen keyboard is visible. Clicking on one of its “keys” inserts the respective character into the input field at the current cursor position. A horizontally scrollable history of characters inserted in this manner is shown in the top row for convenience. The second row contains characters with special meaning in UnicodeMath – you will surely recognize most of them by now. The following rows feature various other frequently used characters, including the Greek alphabet. Hovering over many of the “keys” pops up a tooltip with a short explanation.

The “Codepoint” text field allows the user to enter a Unicode character's hexadecimal representation, a click on the adjacent "➥" button inserts this character into the input field.

The “Control word” text field functions similarly, accepting the control words defined in Section 4.6 and Appendix B of the tech note, among others – with or without a leading backslash.

[1] This splitting can be disabled in the options pane, which opens when hovering over the gear icon in the top right corner of the page.

[2] See `http://www.gust.org.pl/projects/e-foundry/lm-math/`.

[3] Another item of the options pane enables display of a parse trace. This feature is not enabled by default as it slows things down considerably.

The "Math fonts" text field and the buttons next to it are a feature I'm rather proud of: Upon entering a character, the buttons signifying a "font"[1] that the specific character is not available in are grayed out. If the text field is empty but text is selected in the main input field, a click on one of the button converts as many characters as possible accordingly, leaving the rest unchanged.

Finally, a number of examples is shown at the bottom of the page. They are clickable in the same manner as the "keys" further up.

Hovering over the gear icon in the top right corner reveals an options pane. Here, the user can enable and disable a number of features, such as

- whether control words such as `\alpha` should be replaced with the respective character before parsing,
- whether to present the results in `displaystyle` mode (as opposed to `textstyle` mode),
- whether the parser should cache intermediate results (discussed in Section 2.7.1.2 and Section 7.1.5),
- whether a debug mode with additional `console.log` output should be enabled, and
- whether to output LaTeX instead of MathML (which is still experimental at this time – see Section 8.1).

Note that the playground keeps a copy of its state (encompassing input, active tab, history and selected options) in local storage[2]. Apart from guarding against data loss in the event of a crash, this allows the user to simply reload the page after making changes to any steps of the UnicodeMathML pipeline to see these changes reflected in the output.

6.1.2 IMPLEMENTATION

Some itemized notes on the playground's implementation:

- The UnicodeMath parser is generated by PEG.js from the grammar on-demand at load time – any errors are shown to the user. A detailed error object is `console.logged`.
- Syntax highlighting of JSON objects and MathML code is based on code snippets found on the Stack Overflow and W3Schools – embedding a library like highlight.js[3] seemed like overkill.
- The JavaScript library jQuery[4] is used in some places, mainly to simplify event handling.
- Whether a browser supports MathML is based on the browser's user agent. It's conceivable to determine this experimentally, for example by introducing a hidden

[1] More precisely, these "fonts" are subsets of the *Mathematical Alphanumeric Symbols* block. See `https://en.wikipedia.org/wiki/Mathematical_Alphanumeric_Symbols#Tables_of_styled_letters_and_digits` for a character table, along with a history of documents relating to this block – it turns out that a certain Murray Sargent has authored a number of them.

[2] See `https://developer.mozilla.org/en-US/docs/Web/API/Window/localStorage`.

[3] See `https://highlightjs.org/`.

[4] See `https://jquery.com/`.

`<math>` element containing a simple fraction and checking the element's aspect ratio: in a MathML-capable browser, the element would be taller than it is wide, in other browsers, *vice versa*. This level of future-proofing for browser feature evolution seemed unnecessary, though.

- The width of the “keys” for inputting the various math spaces needed to be set manually – the actual widths of the spaces depend on the font and are by no means consistent with their definitions (neither are they internally consistent within a given font).
- The `draw` function, called upon to re-translate and draw the input after each key stroke, splits the input into separate expressions on newlines. In an previous version of the playground, it would iterate through these expressions and integrate the translation results into the DOM – *i.e.*, the append the new MathML code to the output area and update each tab's content in the same manner - after translating each expression. This turned out to be a performance bottleneck when operating on tens of expressions at a time, which I addressed by collecting the new data in temporary variables and inserting these into the DOM after all UnicodeMath expressions have been translated.
- The playground does not implement the sort of caching used within the Markdeep and HTML integration – this is owed to its use as a development tool, where caching might lead to confusion.

6.2 MARKDEEP-BASED TOOLS AND APPLICATIONS

6.2.1 MARKDEEP-THESIS

As the work on parsing and transforming UnicodeMath to MathML came to a close, the task of writing about it began to emerge from the understory.

Faced with the need of showing both UnicodeMath source code and the parsed, transformed, and – most importantly – rendered result, there were three options:

1. Write the thesis in LaTeX and manually rewrite any translation results in LaTeX for rendering.
2. Write the thesis in LaTeX, but use its `--shell-escape` machinery to call a script of some sort which would spin up a Node.js server that would 1. apply the UnicodeMath to MathML transformation to a given UnicodeMath expression and 2. render the MathML code using MathJax, yielding an SVG file. The script would then rely on the command-line version of Inkscape[1] or a similar tool to convert the SVG file into a format (ideally PDF) that could then be inserted into a LaTeX document.
3. Write my own browser-based document typesetting tool.

I felt that choosing option 1 would be somewhat dishonest, and option 2 seemed both more prone to breakage and less interesting (and certainly less reusable) than option 3.

[1] See `https://tex.stackexchange.com/questions/2099/how-to-include-svg-diagrams-in-latex`.

As the name suggests, the tool I came up with is based on Markdeep, which takes care of the Markdown to HTML conversion and kicks off the UnicodeMath to MathML transformation. Then, a custom style sheet is applied to make the document look like a thesis. Finally, the JavaScript library Bindery[1] takes over and splits the web page generated by Markdeep into several printable pages. The resulting document can be saved as a PDF file via the browser's[2] print dialog.

It wasn't all smooth sailing, however:

1. Markdeep makes extensive use of CSS counters[3] for section numbering and code listing line numbering. These counters occasionally reset as Bindery subdivides the document into pages, which is addressed in a Markdeep-postprocessing step where each instance where the value of a CSS counter is accessed is replaced with a snapshot of the counter's value at the time.
2. Markdeep supports endnotes, not footnotes. Although this is not a bug *per se*, footnotes are more convenient for the reader. Thus, in another Markdeep-postprocessing step, the text associated with every endnote is temporarily stored in the superscripted reference to it. Then, as Bindery processed the document, a `Bindery.PageReference` hook converts these stored endnote contents to footnotes on the respective pages.
3. Bindery can parse a table of contents and insert the page number of each entry. However, the table of contents generated by Markdeep is not formatted in a way that is conducive to this task, so a third postprocessing step is required to transform it accordingly.[4]
4. The MathJax version integrated into Markdeep is not configured for optimal print quality, so a differently-configured MathJax variant is loaded after Markdeep postprocessing and before Bindery is invoked.
5. Finally, the order of operations – Markdeep, postprocessing, MathJax, Bindery – was a bit tricky to get right on page load since there is no standard way for each of these libraries to signal that they're done processing the document.

Note that although I finished `markdeep-thesis` (save for minor improvements) before implementing the final variant of UnicodeMathML's Markdeep integration and commencing work on this thesis, replacing stock Markdeep with my patched version and including the integration itself went without a hitch.

Another note I'd like to leave is that browsers don't excel at rendering justified text, as you may have noticed reading this document[5] – their rendering algorithms appear to be

[1] See `https://evanbrooks.info/bindery/`.

[2] In fact, only Google Chrome respects the page size set using the CSS `@page` selector. Surprisingly, the resulting PDF misses some fonts or has incorrectly sized text when summoning the print dialog using a keyboard shortcut – things seem to only reliably look right when opening the dialog via the *File* menu.

[3] See `https://developer.mozilla.org/en-US/docs/Web/CSS/CSS_Lists_and_Counters/Using_CSS_counters`.

[4] Note that the dots guiding the reader's gaze from section title to page number in the table of contents of this thesis are created using W3C-approved CSS magic: `https://www.w3.org/Style/Examples/007/leaders.en.html`

[5] Although in this instance, it's somewhat alleviated by browsers' equally non-excellent hyphenation support, see `https://developer.mozilla.org/en-US/docs/Web/CSS/hyphens`.

optimized for speed above all else. While there exists an open-source implementation[1] of the Knuth and Plass line breaking algorithm [Knuth81], and a related issue[2] has been open in Firefox's bug tracker for nine years, there is no "production-ready" way of using a better line breaking algorithm.

Despite the issues mentioned above, `markdeep-thesis` has performed well in typesetting this thesis document (about 20 seconds elapse between opening it in a browser and it finishing rendering, which is roughly on par with LaTeX for a document of this size and complexity). My aim is to release[3] it as free software shortly after submission of this thesis.

6.2.2 MARKDEEP-SLIDES

Having built several slide decks for seminar and thesis presentations at university – initially with LaTeX and the Beamer package, then with the Markdown-based, but proprietary Mac app Deckset, and finally with a modified version of a Markdeep-powered slide template[4] – I realized that it wouldn't take a great deal of effort to build a Markdeep-powered presenation tool.

Built before `markdeep-thesis`, `markdeep-slides` utilizes some of the same postprocessing concepts to split a Markdeep document into slides. Depending on the the user's preference, slide breaks are either inserted before headings or they replace horizontal rules, expressed in Markdeep as `---`. Display of presenter notes is supported as well – in Markdeep source code, they need to be wrapped in triply nested block quotes, *i.e.*, preceded by >>>.

A keyboard shortcut allows the user to switch between two modes:

- *Draft* mode, where all slides are shown in a scrolling view, with presenter notes below each slide. This mode is intended to be used during the making of slides.
- *Presentation* mode, where a single slide fills the screen and standard shortcuts switch between them. The presenter notes, along with a timer, are shown in a separate window visible to the presenter.

A PDF version of the slides can be generated in the same manner as it can for `markdeep-thesis`. Three pre-built themes are available. The author of the slides can set aspect ratio, theme, font size and other settings in their Markdeep source code.

The central problem that needed solving here was the scaling of slides to arbitrary screen sizes. Instead of relying on CSS transforms (which may not work reliably when including videos or making slides interactive with JavaScript) or the non-standard `zoom` CSS property[5], the size of all elements visible during a presentation is specified in `rem` or `em` units.

[1] See `https://github.com/bramstein/typeset`.

[2] See `https://bugzilla.mozilla.org/show_bug.cgi?id=630181`.

[3] It will be publicly available at `https://github.com/doersino/markdeep-thesis` – or, if you're living in the future, it might already be up.

[4] See `https://casual-effects.com/markdeep/slides.md.html`.

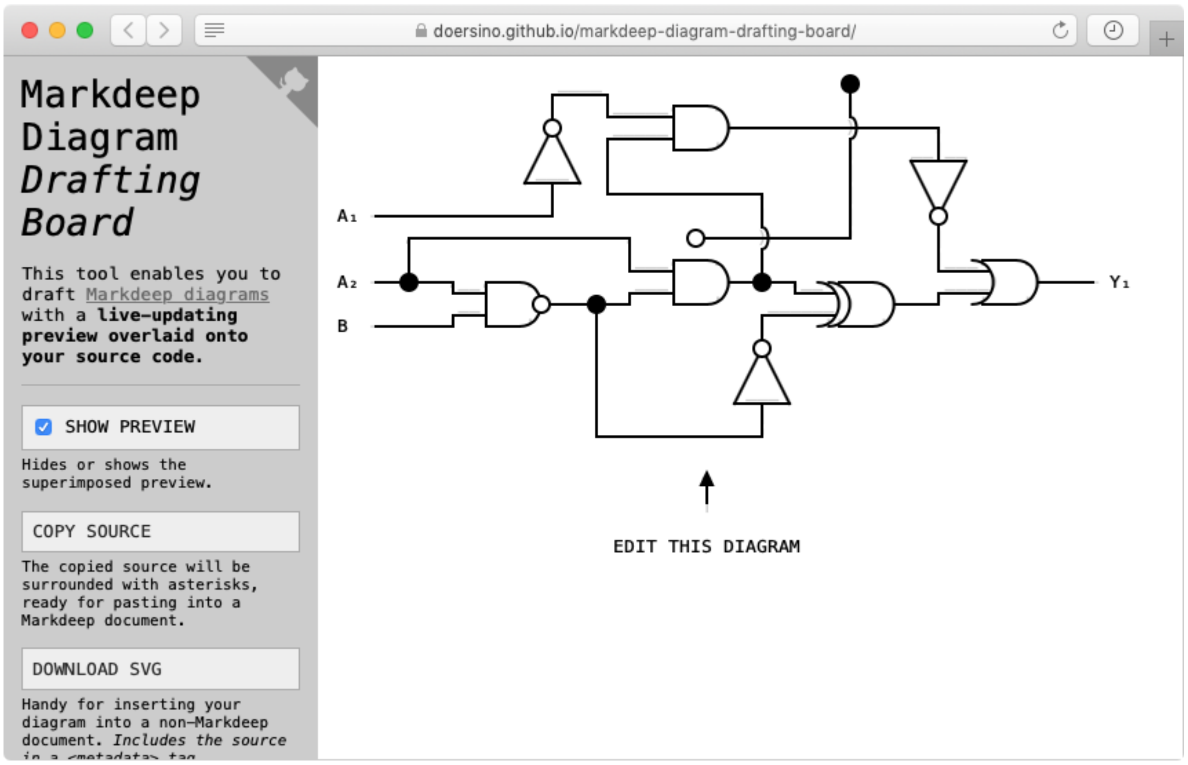
[5] See `https://developer.mozilla.org/en-US/docs/Web/CSS/zoom`.

These units are relative to the root elements font size, which I specify in `vw` or `wh` units (depending on how the configured slide aspect ratio compares to the viewport's size) – meaning that all elements of the page scale properly[1] when the window size changes.

My thesis defense will utilize a slide deck built with `markdeep-slides`. (I plan on augmenting it by integrating a miniature version of the UnicodeMathML playground for interactive demonstrations.)

6.2.3 MARKDEEP-DIAGRAM-DRAFTING-BOARD

Markdeep diagrams are designed to be readable in plain text form, but sometimes it's still nice to have a preview. Due to this plaintext readability constraint, rendered diagrams can be superimposed over their source code and instantly updated as the user modifies the source. Guided by this insight (and in a fit of procrastination), I implemented this behavior in a small tool, available at `https://github.com/doersino/markdeep-diagram-drafting-board` and shown in Figure 6



Markdeep Diagram Drafting Board

This tool enables you to draft Markdeep diagrams with a **live-updating preview overlaid onto your source code.**

Hides or shows the superimposed preview.

The copied source will be surrounded with asterisks, ready for pasting into a Markdeep document.

Handy for inserting your diagram into a non-Markdeep document. *Includes the source* in a <metadata> tag

Figure 6: *markdeep-diagram-drafting-board in action.*

[1] Barring browser bugs, of course: `https://github.com/doersino/markdeep-slides#known-issues`

6.3 PYTHON SCRIPTS

The following Python 3 scripts are located in `/code/utils/`.

- Given a newline-separated list of characters (either from a file or via standard input), `characters-to-codepoints.py` outputs the corresponding Unicode code points. The inverse function is implemented by `codepoints-to-characters.py`. The two scripts are actually isomorphic, *i.e.*, `python3 codepoints-to-characters.py foo.txt | python3 characters-to-codepoints.py` (and *vice versa*) is a rather inefficient way of running `cat foo.txt`.
- Given two files from `https://www.unicode.org/Public/12.1.0/ucd/` – a publicly available dump of the *Unicode Character Database* [Whistler19] – `charinfo.py` generates a piece of JavaScript code that is used within the UnicodeMathML playground to provide information on Unicode characters. Stored as `/code/playground/charinfo.js`, it provides a mapping from code point to name, block, and category. (In addition, it's responsible for enabling a special code point notation in this thesis, *e.g.*, turning `___U+1F52D___` into U+1F52D TELESCOPE during typesetting.)
- Finally, `emoji.py` parses `https://unicode.org/Public/emoji/12.0/emoji-data.txt` and outputs a list of code point ranges that aim to identify all emoji, as well as a mapping of astral plane emoji into the BMP's *Private Use Area* – the `emoji` parse rule in `/code/src/unicodemathml.pegjs` is directly based on this script's output. This is further discussed in Section 7.1.2.

7

EVALUATION

The focus of this chapter is divided among two points: First, I will discuss aspects of the work that were challenging to me (or just plain challenging – you will be the judge of that, I suppose) for a variety of reasons. Also, I will swiftly take a look at how UnicodeMathML – viewed in isolation, as well as its integration into Markdeep – performs and why it performs this way.

7.1 CHALLENGES

7.1.1 PARSER GENERATORS

In Section Section 2.7.2, I've already mentioned that I switched parser generators twice: From PEG.js to ANTLR and back again (the second switch was made easier as my ANTLR branch was just about at the same level as the PEG.js at that time). This happened around a third of the way through the project.

The switch **from PEG.js to ANTLR** was prompted by my realizing that PEG.js does not handle astral plane characters properly. This is not an "unforced error" – PEG.js simply inherits the limitations inherent to JavaScript's UTF-16 string encoding (refer to Section 2.4 for a primer) where astral characters are specified in the form of surrogate pairs.

This would not be a deal-breaking issue if only *specific operators*, *e.g.*, one of the fraction slashes, were astral-plane characters (they are specified as strings in the grammar). However, when it comes to *ranges* of characters – such as the *Mathematical Alphanumeric*

Symbols block – that need to be matched using PEG.js's regular-expression-like syntax, things become tricky.

For example, a range specified as `[𝒜-𝒵]`, *i.e.*, the range from U+1D49C MATHEMATICAL SCRIPT CAPITAL A to U+1D4B5 MATHEMATICAL SCRIPT CAPITAL Z – which, by the way, cannot be written as `[\u1D49C-\u1D4B5]` since PEG.js's own grammar doesn't allow this – is interpreted by PEG.js as `[\uD835\uDC9C-\uD835\uDCB5]`, *i.e.*, either U+D835, the nonsensical range from U+DC9C to U+D835, or U+DCB5. While it is technically possible to instead specify this range as a list of alternatives (like `"𝒜" / "ℬ" / ⋯ / "𝒵"`) and have PEG.js parse it correctly, this would result in dreadful parsing performance (the reasons for which I will discuss in Section 7.1.5).

After trialling a number of other JavaScript-emitting parser generators (which all replicated PEG.js's shortcomings in this area), I came across ANTLR. A basic overview that explains its upsides (including the option of targeting other languages, not just JavaScript, in the future) is given in Section 2.7.2.

Once I had verified that its JavaScript target[1] deals with astral plane ranges correctly[2], I transformed my PEG.js-based UnicodeMath grammar into ANTLR's format. This was non-trivial because ANTLR requires a separate lexing stange to be run prior to parsing. Furthermore, I was unable to use semantic actions to build up a custom AST during parsing – instead I had the options of

1. dealing with ANTLR's automatically generated parse tree during the translation stage or
2. transforming it into a form resembling the AST generated by PEG.js's semantic action before the actual translation to MathML.

I went with option 2 in order to keep the central translation function interchangeable between the two different parsing infrastructures. This, combined with the lexer requirement and the generally higher toolchain complexity, made working with the ANTLR variant more cumbersome than with PEG.js.

This is not what broke the camel's back (or, perhaps more fittingly, the deer's antler), though: Parsing performance turned out to be extremely poor when using ANTLR's JavaScript target. After implementing only a fairly constrained subset of the UnicodeMath grammar (located at ☁`/antlr-experiment/UnicodeMath.g4`), the UnicodeMath expression `a^b^c^d^e` took almost 13 seconds to parse. You may verify this by opening ☁`/antlr-experiment/index.html` in any browser and clicking the button highlighted in Figure 7.

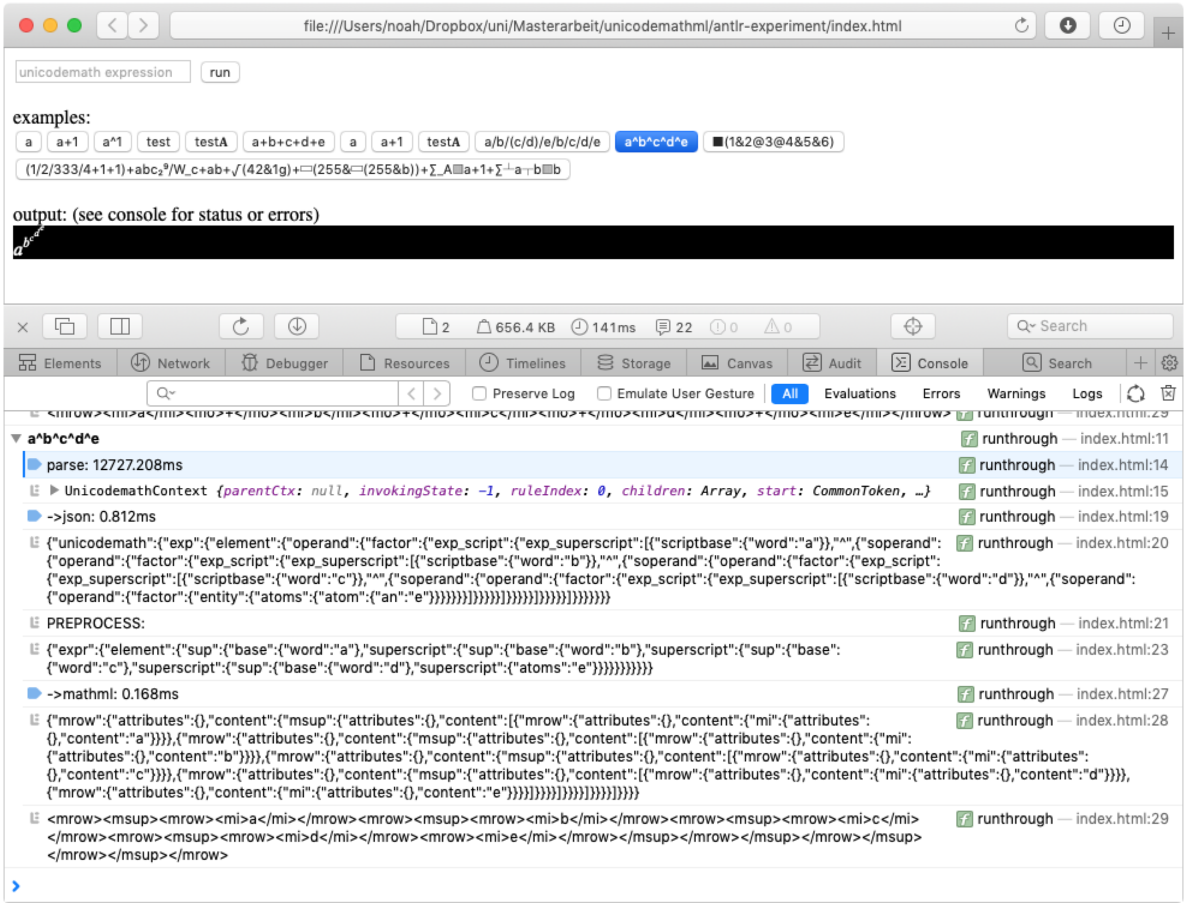
Based on some cursory tests I carried out with Chrome's profiler at the time, the issue seemed to be rooted in the ALL(∗) algorithm consistently mispredicting which grammar rules to try next, but I didn't explore this in detail.

For comparison, when using ANTLR's native Java target by means of invoking `bash`

[1] See `https://github.com/antlr/antlr4/blob/master/doc/javascript-target.md`.

[2] See `https://github.com/antlr/antlr4/blob/master/doc/unicode.md`.

☁`/antlr-experiment/tree.sh "a^b^c^d^e"` (note that this bash script is a thin wrapper around ☁`/antlr-experiment/Tree.java`; refer to ☁`/antlr-experiment/README.md` for help with setting things up), the same expression is parsed in around 460 ms, at least 300 ms of which is parser generation overhead.



```
▼ a^b^c^d^e                                                                                         runthrough — index.html:11
  parse: 12727.208ms                                                                                runthrough — index.html:14
  ▶ UnicodemathContext {parentCtx: null, invokingState: -1, ruleIndex: 0, children: Array, start: CommonToken, …}   runthrough — index.html:15
  ->json: 0.812ms                                                                                   runthrough — index.html:19
  {"unicodemath":{"exp":{"element":{"operand":{"factor":{"exp_script":{"exp_superscript":[{"scriptbase":{"word":"a"}},"^",{"soperand":   runthrough — index.html:20
  {"operand":{"factor":{"exp_script":{"exp_superscript":[{"scriptbase":{"word":"b"}},"^",{"soperand":{"operand":{"factor":{"exp_script":
  {"exp_superscript":[{"scriptbase":{"word":"c"}},"^",{"soperand":{"operand":{"factor":{"exp_script":{"exp_superscript":[{"scriptbase":{"word":"d"}},"^",{"soperand":
  {"operand":{"factor":{"entity":{"atoms":{"atom":{"an":"e"}}}}}}}]}}}}}]}}}}}]}}}}}]}}}}}}}
  PREPROCESS:                                                                                       runthrough — index.html:21
  {"expr":{"element":{"sup":{"base":{"word":"a"},"superscript":{"sup":{"base":{"word":"b"},"superscript":{"sup":{"base":   runthrough — index.html:23
  {"word":"c"},"superscript":{"sup":{"base":{"word":"d"},"superscript":{"atoms":"e"}}}}}}}}}}
  ->mathml: 0.168ms                                                                                 runthrough — index.html:27
  {"mrow":{"attributes":{},"content":{"msup":{"attributes":{},"content":[{"mrow":{"attributes":{},"content":{"mi":{"attributes":   runthrough — index.html:28
  {},"content":"a"}}}},{"mrow":{"attributes":{},"content":{"msup":{"attributes":{},"content":[{"mrow":{"attributes":{},"content":{"mi":
  {"attributes":{},"content":"b"}}}},{"mrow":{"attributes":{},"content":{"msup":{"attributes":{},"content":[{"mrow":{"attributes":{},"content":{"mi":{"attributes":
  {},"content":"c"}}}},{"mrow":{"attributes":{},"content":{"msup":{"attributes":{},"content":[{"mrow":{"attributes":{},"content":{"mi":{"attributes":{},"content":"d"}}}},
  {"mrow":{"attributes":{},"content":{"mi":{"attributes":{},"content":"e"}}}}]}}}}]}}}}]}}}}]}}}}
  <mrow><msup><mrow><mi>a</mi></mrow><mrow><msup><mrow><mi>b</mi></mrow><mrow><msup><mrow><mi>c</mi>   runthrough — index.html:29
  </mrow><mrow><msup><mrow><mi>d</mi></mrow><mrow><mi>e</mi></mrow></msup></mrow></msup></mrow></msup>
  </mrow></msup></mrow>
>
```

Figure 7: *A simple UnicodeMath expression takes an eternity to parse when using ANTLR's JavaScript backend.*

Another point of comparison: At the time, my PEG.js-based parser supported the same UnicodeMath subset and parsed the same expression instantly.

With ANTLR evidently leading me nowhere, I **switched back to PEG.js** thinking that if UnicodeMathML couldn't support fancy math alphabets, I should at least be fast. This eventually led me to the idea I refer to as *astral mapping*, described in Section 3.2.2, which fixed the issue in practice.

This is also not entirely unrelated to emoji: some of them are encoded in the SMP.

7.1.2 EMOJI

Supporting emoji as first-class operands in UnicodeMath (*i.e.*, not only within plain text zones) was one of my goals (even though there is no mention of emoji in the tech note). In teaching and similar contexts, symbolic equations denoted using emoji can be rather elegant.

However, emoji as they are encoded in Unicode are rather complicated:

- There is no "Emoji" block – emoji are scattered across both the BMP and SMP.
- Emoji can precede various modifiers[1] which, while being separate code points, modify[2] the base character. For example, the emoji 🧘🏼‍♀️ is the result of concatenating the five code points U+1F9D8 PERSON IN LOTUS POSITION, U+1F3FC EMOJI MODIFIER FITZPATRICK TYPE-3, U+200D ZERO WIDTH JOINER, U+2640 FEMALE SIGN, and U+FE0F VARIATION SELECTOR-16, not all of which are categorized as emoji on their own.
- Symbols that are not categorized as emoji can display as emoji when paired with a variation selector[3] (and *vice versa*). For example, ☀ (U+2600 BLACK SUN WITH RAYS) turns into the sun emoji ☀️ when followed by U+FE0F VARIATION SELECTOR-16.

Before realizing how complex[4] this would be if it was to be done correctly, I had started writing a Python script (see Section 6.3) that parses one[5] of the various[6] emoji-related and outputs 1. a mapping of astral emoji into the BMP's *Private Use Area* and 2. a corresponding PEG.js rule. This list only includes simple one-code-point emoji that *do* display as emoji by default, which limits its usefulness – implementing anything more complicated, though, would negatively impact UnicodeMath parsing performance, which I didn't want to risk for a relatively unimportant feature like this.

The previously mentioned constraints (see Section 4.1 and Section 4.19) relating to the grammar rules **αn** and **nn** are also caused by the symbols comprising the relevant Unicode categories being scattered across multiple blocks, making it difficult to assemble regular expressions that are both performant and accurate.

[1] See `https://www.unicode.org/emoji/charts/full-emoji-modifiers.html`.

[2] See `https://blog.emojipedia.org/emoji-zwj-sequences-three-letters-many-possibilities/`.

[3] See `https://unicode.org/Public/emoji/12.0/emoji-variation-sequences.txt`.

[4] In fact, while working on this, I realized that different browsers, text editors and operating systems differ wildly in terms of completeness and correctness of their emoji rendering. Nobody seems to do it right, and whenever anyone comes close, Unicode seems to add more emoji, along with more ways of combining existing ones. See also `https://www.unicode.org/reports/tr44/proposed.html#emoji-data.txt`.

[5] See `https://unicode.org/Public/emoji/12.0/emoji-data.txt`.

[6] See `https://www.unicode.org/emoji/charts/`.

7.1.3 ABSTRACT BOXES

In Section 3.7 of the tech note, Sargent introduces abstract boxes. They can supposedly be used to change "alignment, spacing category, size style *[sic]*, and other properties" [Sargent16] based on a user-specified bit mask, however Sargent does not explain the precise nature of these changes, instead just giving a table mapping bit mask values to rather inscrutable keys like "fXPositioning", "fBreakable", or "nSpaceDifferential". (Other bit masks are explained in the same manner, but the associated meanings are clear from context in these cases.)

Due to the subtle nature of some of these changes, I was also unable to experimentally determine what most of the possible bit mask values effect by testing them in Microsoft Word.

For this reason, abstract boxes remain largely unimplemented in UnicodeMathML.

7.1.4 EQUATION BREAKING AND ALIGNMENT

As previously noted in Section Section 4.3, MathML's `<malignmark>` is not implemented by any renderers I'm aware of. Although I've implemented them, equation arrays are made somewhat useless by this.

For this reason – and because it would introduce more complexity to the grammar as well as the translation step – I have opted not to implement inter-equation alignment as detailed in Section 3.23 of the tech note, which is really just syntactic sugar for equation arrays.

7.1.5 PARSING PERFORMANCE

As glad as I was after switching away from ANTLR's glacially slow JavaScript backend and back to PEG.js – during continued expansion of the grammar to support more of UnicodeMath's features and more accurately reproduce some of its edge cases, parsing performance took a steep turn downhill for some[1] inputs. Eventually, when the grammar was basically "done", expressions containing more than one level of bracketed subexpressions were basically unusable. Even innocent-looking expressions like $\{v_i : i \in \{1,2,3,4,5\}\}$ took seconds until they were parsed, and expressions containing many absolute values performed even worse (because opening and closing delimiter are equal for them), leading to some initial pessimism on my side as to the applicability of UnicodeMathML in any real-world context.

While I initially had a hard time finding purchase on this issue, I eventually addressed it – aided by Chrome's built-in JavaScript profiler – by...

1. ...collapsing grammar rules consisting of long lists of literal (*i.e.*, not referencing

[1] Related: `https://github.com/pegjs/pegjs/issues/623`

other grammar rules) alternatives like `"ⅅ" / "ⅆ" / "ⅇ" / "ⅈ" / "ⅉ"` into regex-style rules like `[ⅅⅆⅇⅈⅉ]` wherever possible. This had a modest, but noticeable effect on the parse times of all UnicodeMath expressions I tested.

2. ...taking advantage of the fact that PEG.js deterministically tries alternatives in order. For example, the **sfactor** rule was defined as follows at one point:

```
sfactor
    = enclosed
    / abstractbox
    / hbrack
    / root
    / function
    / text
    / sizeOverride
    / colored
    / comment
    / tt
    / primed
    / factorial
    / entity
```

   Notice that the rule **entity** is listed at the very bottom. This is because PEG.js parsers are greedy: Both **primed** and **factorial** can begin with **entity***s (the rules listed above them are all prefixed with a specific operator, so they are discarded quickly during parsing if they do not match), so the more common* ***entity** alternative can only be tried after both of them have been tried and discarded. To circumvent this, I carefully modified the grammar as follows:

```
sfactor
    = enclosed
    / abstractbox
    / hbrack
    / root
    / function
    / text
    / sizeOverride
    / colored
    / comment
    / tt
    / e:entity !("'" / "′" / "″" / "‴" / "⁗" / "‼" / "!") {return e}  // ϟ performance
optimization
    / primed
    / factorial
    /// entity  // ϟ performance optimization
```

   Applying the same thinking to a number of other rules (marked by `// ϟ performance optimization` in `/code/src/unicodemathml.pegjs`) made the parser significantly faster in pathological cases while at the same time not noticeably negatively affecting other cases.

Performance teetered on the edge of acceptability, but was still not ideal: The 250 test expressions listed at the bottom of `/code/markdeep-integration/markdeep.md.html` still took about 10 total seconds to parse successfully when pasted (sans delimiters) into the

UnicodeMathML playground running in Chrome and Firefox, and closer to 20 seconds in Safari (this disparity is largely due to effective caching mechanisms built into Chrome's and Firefox's JavaScript engines). But I could live[1] with that.

However: While writing Section 2.7.1 of this thesis, I was going through PEG.js's documentation for one reason or another, noticing that there is a *caching* option I had not noticed before. When I configured the playground to enable this option, the parsing time for the 250 test expressions dropped to around half a second in Chrome and Firefox and one second in Safari, about 20 times faster[2] than sans caching.

Needless to say, "RTFM" is one of the central lessons I'm taking away from my work on this thesis.

7.2 PERFORMANCE

The discussion in the previous section isn't all I want to note about UnicodeMathML's performance:

Firstly, the parsing step is still the bottleneck, even with caching enabled: The time taken by preprocessing, translation and pretty-printing is negligible. It's below 1 ms for all reasonable UnicodeMath expressions I've tried (the only times when these steps end up taking multiple milliseconds is when they coincide with a garbage collection cycle).

As I've brought up in Section 5, the naïve integration of UnicodeMathML into Markdeep is the fastest way of translating UnicodeMath zones contained in a document into MathML. Because this method is wholly devoid of overhead (apart from the evaluation of a regular expression and a loop through its matches), it is up to twice as fast as translating the same UnicodeMath expressions in the UnicodeMathML playground, where a lot of extra work is done in order to display various ASTs.

The slowest – but still acceptably fast – method is the "better" integration of UnicodeMathML into Markdeep. It's also the only one that gives the user an insight into the translation progress. See Section 8.3 for ideas on how to speed it up.

Note that in order to compare performance of these three translation "pathways" on real-world data (instead of the 250 test expressions located in ♣ `/code/markdeep-integration/markdeep.md.html` or the 158 examples in this thesis), I manually translated the first 253 mathematical expressions contained in "A quick and dirty guide to Geometric Algebra" by Isaac Serrano Guasch[3] from LaTeX into Markdeep – this guide is already a

[1] I considered radically rewriting the grammar in order to left factor it (see `https://stackoverflow.com/questions/15194142/difference-between-left-factoring-and-left-recursion` for more information), but 1. this would make for a significantly less human-readable grammar and 2. would require wide-reaching changes to the entire pipeline.

[2] I tried removing my optimizations once I figured this out, but that had a negligible impact on parsing performance, so I decided to leave them in for now – for documentation purposes, if nothing else.

[3] See `http://atridas87.cat/GA/quickga.md.html`.

Markdeep document, so I was able to drop in the UnicodeMathML integration in a matter of seconds. My version of it is located at `/code/markdeep-integration/quickga.md.html`.

7.3 EXTENSIBILITY AND MAINTAINABILITY

I've kept the structure of the parser and translator deliberately simple in order to aid future adjustments – and indeed, that's what I'm planning to do: Since writing about *extensibility and maintainability* implies that there's *future work* on the horizon, let's switch gears to that.

8

FUTURE WORK

I'm indent on releasing[1] UnicodeMathML as free software in the near future. Before going ahead with this, I would like to address a number of issues – some of which have already been mentioned – that I either didn't end up having enough time for or that weren't within the scope of this thesis. They will be discussed in this chapter, along with some further-fetched ideas that I likely won't implement.

8.1 UNICODEMATHML

The core of the work – parsing UnicodMath and translating it into MathML – works well enough, but a few matters remain to be addressed:

- Recall that the grammar rules **αn** and **nn** currently don't accept as many different characters as prescribed by the tech note due to JavaScript regular expressions lacking support for matching Unicode categories: I'd like to fix that.
- I believe that I can improve parsing performance even more, either through careful small-scale grammar adjustments or outright left factoring of the grammar.
- Three days before submission of this thesis, I found out[2] that the `<mfenced>` element has recently been deprecated[3] – apparently, any occurrence of it can be replaced with an `<mrow>`s containing the delimited expression between two `<mo>`s containing the delimiters. More generally, the recently established

[1] It will be located at `https://github.com/doersino/unicodemathml`.

[2] See `https://wiki.developer.mozilla.org/en-US/docs/Web/MathML/Element/mfenced$history`.

[3] See `https://github.com/mathml-refresh/mathml/issues/2`.

MathML Refresh CG[1] has been and will be making changes to MathML “so that it better aligns with the current web environment, eases the burden on browser implementations, and increases support for assistive technology”. It's not yet clear when and whether this upcoming version of MathML will be adopted by browsers, but any changes will need to be reflected in UnicodeMathML's generated code.
- As previously mentioned, I've implemented an experimental, rudimentary LaTeX code generator that can be used in place of the MathML code transformation and pretty-printing component. This work
  1. provided the impetus for separating out the preprocessing step as described in Section 3.2.4 – this separation isn't quite optimal yet, which I want to address in the future; and
  2. has not yet been finished – some UnicodeMath constructs are not yet supported by my LaTeX-emitting translation step, largely due to time constraints and because it was merely intended as a proof of concept within the scope of this thesis. I'd like to complete it, thus enabling the use of KaTeX for rendering instead of MathJax.
- As mentioned in Section 4.19, I aim to make the appearance of differential, exponential, and imaginary symbols configurable.
- Before releasing UnicodeMathML as open-source software, I will reexamine the thoughts on modularization I laid out in Section 3.3.

8.2 UNICODEMATHML PLAYGROUND

- Currently, the playground's instant preview feature is implemented *synchronously*. For this reason, when previewing many different equations at once, the playground becomes unresponsive for a noticeable length of time as the translation is in progress – and since translation is kicked off whenever the value of the input field changes, this limits the user's typing speed. I'd like to “asynchronize” this similarly how the integration of UnicodeMath into Markdeep has been implemented.
- Another playground-related bottleneck is the loading of the 1.5 MB large file ☁ `/code/playground/charinfo.js`. I think it's possible to compress the code point metadata contained within this file to dramatically reduce its size.
- It'd be neat to implement Microsoft-Office-style equation build-up in the playground, but this would be a major undertaking.
- A start would be to replace control words with their meanings as the user finishes typing them them. In the same vein, I'd like to implement the hexadecimal input method detailed in Section 4.3 of the tech note.
- I've been thinking about what should be done if the user were to paste text containing ⁅delimited⁆ UnicodeMath expressions into the input filed – should *only* these expressions be translated? Or should the pasted text be processed as a Markdeep or HTML document? Currently, the playground can handle “raw” UnicodeMath input only.
- User-configurable control words (essentially macros) would be useful – MathJax implements this feature for LaTeX.

[1] See `https://www.w3.org/community/mathml4/`.

8.3 MARKDEEP INTEGRATION

- Here, too, user-configurable control words would be useful.
- As previously mentioned, repeated DOM manipulations during transformation of the expressions contained within documents negatively impact performance. This could be addressed by batching these DOM manipulations and, in addition, updating the progress meter less frequently, perhaps once or twice a second instead of after translation of each expression.
- Ultimately, one goal of this thesis is the merging of the integration-related changes to Markdeep into “mainline” Markdeep. I intend to propose this to Morgan McGuire once UnicodeMathML has been made available as open-source software.
- Some folks have strong opinions on monospaced fonts, so I will make the font used in typewriter text mode configurable.

8.4 GRAB BAG

- While it was not a priority within the scope of this thesis, I will explore how to embed UnicodeMathML into server-side programs – there are, of course, use cases where it's more efficient to translate UnicodeMath into MathML (and perhaps render it at the same time) just once instead of on every page load.
- Server-side caching of translated UnicodeMath expressions might be a helpful feature in certain contexts, too.
- “TeXnique”[1] is a LaTeX speed-typesetting game – forking it and turning it into a UnicodeMath learning tool seems doable (once better input methods have been implemented). Finding an equally punny name seems more difficult, however.
- I will continue maintaining `markdeep-slides` and `markdeep-diagram-drawing-board` along with their new siblings `markdeep-thesis` and `unicodemathml`.

Master's theses are supposed to have the author draw conclusions at the end. I suspect that any *actionable* conclusions have been discussed in sufficient detail in the previous two chapters, and a summary of my work can be found in both the abstract and the introduction, so all that's left to say is:

$$\text{So long} \land \text{thanks} \; \forall \; \text{🐟 🐠 🐡} \; .$$

[1] See `https://github.com/akshayravikumar/TeXnique`.

BIBLIOGRAPHY

All web sources were accessed on November 30, 2019.

[**Beeton16**] Barbara Beeton, Richard Palais. 2016. "Communication of Mathematics with TEX". In *Visual Language*, August 2016. `http://tug.org/pubs/vislang-16/article.pdf`

[**Beeton17**] Barbara Beeton, Asmus Freytag, Murray Sargent III. 2017. Unicode Technical Report #25: Unicode Support for Mathematics. `https://www.unicode.org/reports/tr25/tr25-15.pdf`

[**Carlisle03**] David Carlisle et al. 2003. Mathematical Markup Language (MathML) Version 2.0 (Second Edition). W3C Recommendation. `https://www.w3.org/TR/MathML2/`

[**Davis06**] Mark Davis. 2006. Foreword. In: The Unicode Standard, Version 5.0. Addison-Wesley. Boston, Massachusetts. `https://www.unicode.org/versions/Unicode5.0.0/Foreword.pdf`

[**Ford04**] Bryan Ford. 2004. Parsing expression grammars: a recognition-based syntactic foundation. In *SIGPLAN Notices*, Volume 39, Issue 1, 111-122. `https://doi.org/10.1145/982962.964011`

[**Gruber04**] John Gruber. 2004. Markdown: Syntax. `https://daringfireball.net/projects/markdown/syntax`

[**Igalia19**] Igalia *(no author listed)*. 2019. MathML and Browsers. `https://mathml.igalia.com/news/2019/08/28/mathml-and-browsers/`

[**Kajiya86**] James T. Kajiya. 1986. The Rendering Equation. In *Proceedings of Computer Graphics and Interactive Techniques (SIGGRAPH '86)*, ACM, 143-150. `http://dx.doi.org/10.1145/15922.15902`

[**Kerninghan75**] Brian W. Kernighan, Lorinda L. Cherry. 1975. A system for typesetting mathematics. In *Communications of the ACM*, Volume 18, Issue 3, 151-157. `https://research.swtch.com/eqn.pdf`

[**Knuth81**] Donald E. Knuth, Michael F. Plass. 1981. Breaking paragraphs into lines. In *Software: Practice and Experience*, Volume 11, Issue 11, 1119-1184. `https://doi.org/10.1002/spe.4380111102`

[**Knuth99**] Donald E. Knuth. 1999. Digital Typography. CSLI Publications. Stanford, California.

[**McGuire19**] Morgan McGuire. 2019. Markdeep. `https://casual-effects.com/markdeep/`

[**Parr14**] Terence Parr, Sam Harwell, and Kathleen Fisher. 2014. Adaptive LL(*) parsing: the power of dynamic analysis. In *ACM SIGPLAN Notices*, Volume 49, Issue 10, 579-598. https://www.antlr.org/papers/allstar-techreport.pdf

[**Prusty15**] Narayan Prusty. 2015. Learning ECMAScript 6. Packt Publishing. Birmingham, United Kingdom.

[**Sargent06**] Murray Sargent III. 2006. How I got into technical WP. https://blogs.msdn.microsoft.com/murrays/2006/09/20/how-i-got-into-technical-wp/

[**Sargent07**] Murray Sargent III. 2007. Two Linear Formats Interoperable with MathML. In *Mathematical User-Interfaces Conferences 2007*. http://www.cermat.org/events/MathUI/07/proceedings/Sargent-TwoSyntaxes-MathUI07.pdf

[**Sargent10**] Murray Sargent III. 2010. Linear Format Notations for Mathematics. https://blogs.msdn.microsoft.com/murrays/2010/08/30/linear-format-notations-for-mathematics/

[**Sargent11**] Murray Sargent III. 2011. Two Math Typography Niceties. https://blogs.msdn.microsoft.com/murrays/2011/04/30/two-math-typography-niceties/

[**Sargent16**] Murray Sargent III. 2016. Unicode Technical Note 28: UnicodeMath: A Nearly Plain-Text Encoding of Mathematics. Version 3.1. https://www.unicode.org/notes/tn28/tn28-5.html

[**Sargent16a**] Murray Sargent III. 2016. UnicodeMath. https://blogs.msdn.microsoft.com/murrays/2016/09/07/unicodemath/

[**Sargent16b**] Murray Sargent III. 2016. High-Quality Editing and Display of Mathematical Text in Office 2007. https://blogs.msdn.microsoft.com/murrays/2006/09/13/high-quality-editing-and-display-of-mathematical-text-in-office-2007/

[**Sargent16c**] Murray Sargent III. 2016. UnicodeMath Version 3.1. https://blogs.msdn.microsoft.com/murrays/2016/11/30/unicodemath-version-3-1/

[**Sargent19**] Murray Sargent III. 2019. Using Math Alphanumerics in Code and Web Pages. https://blogs.msdn.microsoft.com/murrays/2019/02/27/using-math-alphanumerics-in-code-and-web-pages/

[**Siracusa11**] John Siracusa and Dan Benjamin. 2011. The Mouse is Not a Finger. In *Hypercritical*. https://5by5.tv/hypercritical/3

[**Whistler19**] Ken Whistler, Laurențiu Iancu. 2019. Unicode Standard Annex #44: Unicode Character Database. Revision 24. https://www.unicode.org/reports/tr44/tr44-24.html

[**Wolfram00**] Stephen Wolfram. 2000. Mathematical Notation: Past and Future. https://www.stephenwolfram.com/publications/mathematical-notation-past-future/

SELBSTSTÄNDIGKEITSERKLÄRUNG

Hiermit versichere ich, dass ich diese schriftliche Abschlussarbeit selbstständig verfasst habe, keine anderen als die angegebenen Hilfsmittel und Quellen benutzt habe und alle wörtlich oder sinngemäß aus anderen Werken übernommenen Aussagen als solche gekennzeichnet habe.

Ort, Datum, Unterschrift