# MATHLEX

# A WEB-BASED MATHEMATICAL ENTRY SYSTEM

An Undergraduate Research Scholars Thesis

by

MATTHEW J. BARRY

Submitted to Honors and Undergraduate Research
Texas A&M University
in partial fulfillment of the requirements for the designation as

UNDERGRADUATE RESEARCH SCHOLAR

Approved by
Research Advisor: Dr. Philip B. Yasskin

May 2013

Major: Computer Science
Applied Mathematical Sciences

# TABLE OF CONTENTS

Page

# ABSTRACT

MathLex
A Web-Based Mathematical Entry System. (May 2013)

Matthew J. Barry
Department of Computer Science and Engineering
Texas A&M University

Research Advisor: Dr. Philip B. Yasskin
Department of Mathematics

Mathematical formulas are easy to convey in handwritten media, but how should they be represented in electronic format? Unfortunately, mathematical content has not been as well-implemented on the Web as images and video. There are two sides to this problem: *display* and *input*. The former has been solved in multiple ways by representing formulas as images, MathML, or LaTeX (via MathJax). Representing math input is much more difficult and is the subject of this thesis. The goal is to enable users to enter complex formulas. Unfortunately, existing languages either are too complex for an average user (difficult to learn and/or read), only work in a particular environment (they have system and browser compatibility issues), or lack certain math concepts. Some do not even retain mathematical meaning. This thesis presents MathLex, an intuitive, easy-to-type, unambiguous, mathematically faithful input language and processing system intended for representing math input (and potentially display) on the web. It aims to mimic handwritten math as much as possible while maintaining semantic meaning.

## NOMENCLATURE

AJAX Asynchronous JavaScript and XML; a misnomer acronym used to describe the process of making an HTTP request without refreshing the active page

AST Abstract Syntax Tree

BNF Backus-Naur Form (for grammar encoding)

CAS Computer Algebra System

CCLI (NSF DUE) Course Curriculum and Laboratory Improvement program

CDN Content Distribution Network

CSS3 Cascading StyleSheets, version 3

DOM Document Object Model

DUE (NSF) Division of Undergraduate Education

EBNF Extended Backus-Naur Form (for grammar encoding; an extension of BNF)

Flash A browser plugin and animation framework designed by Macromedia and then acquired by Adobe

Formal Language A strictly defined language for interpretation by a computer program

Grammar A standardized encoding of the syntax rules of a formal language

HTML5 HyperText Markup Language, version 5

HTTP HyperText Transfer Protocol

Java A popular object-oriented programming language built to run cross-platform software in a virtual machine; developed by Sun Microsystems (obtained by Oracle in 2009)

Jison A JavaScript parser Generator Library developed by Zach Carter

JS JavaScript; technically an implementation of ECMAScript

JSON JavaScript Object Notation; an increasingly popular data serialization construct alternative to XML

LaTeX Extension of TeX macros for easier document typesetting; written in 1985 by Leslie Lamport and still very popular today

LHS Left-Hand-Side (of a binary expression)

M4C Maplets for Calculus

MathML Math Markup Language, XML specification for representing mathematics on the Web (version 3 released in Oct 2010)

NSF The National Science Foundation

OpenMath An XML standard similar to MathML, but designed to retain semantic mathematical meaning

Parser Validates an input stream (in MathLex's case, a Token stream) and optionally outputs a parse result (MathLex's Parser produces an AST)

Renderer A recursive function utility that traverses the AST produced by MathLex's Parser

RHS Right-Hand-Side (of a binary expression)

TeX A document typesetting language written by Donald Knuth in 1978

Token A string of one or more characters representing a mathematical symbol or quantity

Tokenizer Produces a list/stream of Tokens from an input string

Translator (see Renderer)

TUES (NSF) Transforming Undergraduate Education in Science

URL Uniform Resource Locator

W3C The World Wide Web Consortium

XML eXtensible Markup Language; a superset of HTML that uses arbitrary tag names

# CHAPTER I

# INTRODUCTION

## Audience

This document is written for a broad audience. Although its content is heavily laced with computer science and mathematical theory, the reader needs only a minimal understanding of such concepts.

## Background Information

Mathematical Formulas are easy to convey in handwritten media, but how should they be represented in electronic forms such as email, a web forum, an online homework system, or simple typesetting? There are two sides to this question:

**Question 1.** *How is math embedded (accounting for visual appearance and semantic value) in web pages and other electronic formats?*

**Question 2.** *How can a user input semantic math in an electronic form for submission?* (This is the topic of this thesis)

## Existing Technology

### *Response to Question 1*

The first computer typesetting system was TEX, written by Donald Knuth in 1978 [1]. Leslie Lamport extended TEX to LATEX in 1985 to make it more user-friendly [2]. It is still the most popular math typesetting language, and many technical journals require that papers be submitted in some style of LaTeX; even this thesis is written using LaTeX. With the advent of the Web, math was initially displayed by inserting images of rendered LaTeX. Wikipedia uses this method to represent intricate mathematical formulas. However, pixel-based images do not scale as clearly as text and other page contents. A recent JavaScript plugin, named MathJax, directly renders vector-based, scalable LaTeX on a web page without using images, and is currently lauded as one of the best method to display math on the Web [3].

With the growth in the use of computer algebra systems such as Maple [4], Mathematica [5], TI's Derive [6], Matlab [7], Sage [8], PocketCAS [9], etc., mathematicians became aware that any system for storing mathematical formulas should also preserve the meaning of those formulas. Unfortunately, LaTeX is primarily meant to be a display language and is not intended for storing semantic mathematical content. In contrast, all computer algebra systems have input languages which preserve content and allow computers to interpret and evaluate the input but can obscure math with unconventional notation.

In 1998, the World Wide Web Consortium (W3C) approved the first recommendation of MathML , an XML representation of math content [10]. Even though MathML has undergone two major revisions, manually typing MathML is very tedious and lengthy. Furthermore, browsers' built-in MathML renderers (if present at all) are often inferior compared to LaTeX. MathJax also addresses this problem since it can render MathML in the same way as LaTeX. Apart from MathML, mathematical content has not been well-implemented compared to images and video as Web technology has improved [3].

*Response to Question 2*

Most average users dislike the programming-language-like rigidity of computer algebra systems (CAS) , so they should not be expected to type input for a CAS, let alone in MathML or LaTeX. Nonetheless, users should easily be able to enter subscripts, superscripts, fractions, expanding brackets, special symbols, etc. Some web designers have solved this problem with different plugins, but each has certain "flaws":

- **Illegibility and Difficulty.** As mentioned previously, LaTeX is hard for students to learn, and MathML is essentially impossible. Similarly, most CAS input languages can be difficult to learn and read since they are reminiscent of programming languages. Indeed most systems adopt the conventions of the language on which they are built. For example, Matlab resembles the C programming language; Maplets for Calculus uses Maple's input language, which in turn is based on Pascal; and Sage is a close relative of Python. Thus the ease of learning to use a CAS often depends on the user's familiarity with the underlying language. This is especially apparent in simple interfaces such as Sage Notebook or Maple Worksheet mode: the user is presented with *prompts* for evaluating CAS expressions. The interfaces are minimal and usually do not provide much assistance. Each supported math concept has a specific syntax which must be learned and usually does not resemble handwritten math. For example, the Sage code to evaluate $\int_{\pi/6}^{\pi/3} \mathrm{e}^{a\,t} \cos 3t \,\mathrm{d}t$ must be entered as follows:

**Listing I.1** Sample Integration in Sage

```
a = var('a')
t = var('t')
integrate(exp(a*t)*cos(3*t), t, pi/6, pi/3)
```

  Although not too bizarre, this syntax might not be intuitive to someone who has never used Sage before. Maple allows you to use palettes to enter this integral graphically, but needs a space between $a$ and $t$. A space is not needed between the 3 and $t$.

- **Platform Compatibility.** (i.e. browsers and operating systems). Online homework systems such as Pearson's MyMathLab [11], WileyPlus [12], WeBWorK [13], WebAssign [14], and MapleTA [15] use Oracle Java or Adobe Flash input plugins for writing mathematics. These plugins use a combination of keyboard entry and palettes to build a graphical formula in the input box. Java and Flash only work on some platforms—even then only when the necessary software is installed. Mobile phones and tablets do not support Java or Flash: support for these plugins has dwindled in favor of HTML5 , CSS3 , and JavaScript.

- **Mathematical Completeness.** Any given CAS might lack certain concepts such as boolean algebra, logic, calculus of finite differences, and certain mathematical operations.

- **Retention of Semantic Mathematical Meaning.** LaTeX comes to mind as a math entry system that does not retain mathematical meaning. It works great as a typesetting system, but symbols such as `\times` (×, for multiplication, cross product, etc.) and `^` (superscript or exponent) have multiple meanings that cannot be discriminated. And rightly so, since LaTeX was designed for *presentation* and not *evaluation*.

- **Ambiguity.** WolframAlpha accepts a wide variety of input formats (math included) in a single-line text field and determines meaning based on context [16]. Since the input domain is so large, many mathematical concepts are difficult for it to recognize without the "interference" of non-mathematical interpretations.

  Furthermore, entry palettes and graphical equation editors have made entry easier, but hidden characters used can cause ambiguity: should $ax$ be the name of a variable with two characters, or the product of variable $a$ with variable $x$? Graphical editors often insert a hidden multiplication operator, but the true remedy to this ambiguity would be to insert a dot ($a \cdot x$) or space ($a\,x$) between all multiplications, making them explicit as they would be in a linear, plain text format

All of the technology mentioned so far have limitations, and these limitations are the motivation to build a new and better solution for web-based mathematical input.

## Introducing MathLex

MathLex is a mathematical input language and processing system intended for web browsers. To ensure compatibility with portable media, the MathLex processing system is written in pure JavaScript. All processing is done client-side to increase responsiveness. Its language is meant to be natural, intuitive, easy to type, unambiguous, extensible, and mathematically faithful. It aims to mimic traditional handwritten math as much as possible while maintaining semantic meaning. At the same time, it also supports and draws inspiration from other languages to enhance familiarity among programmers and other CAS users. To maximize user-friendliness, MathLex should be intelligent and flexible enough to automatically repair simple errors.

The MathLex processing system is currently nothing more than an advanced parser: it accepts a linear input string and produces an interpreted syntax tree of the math's semantic value. The resulting syntax tree may then be translated into any CAS language, into OpenMath [17] or MathML for storage, or into LaTeX for display (rendered using MathJax).

MathLex materialized in response to a desire to port Maplets for Calculus (M4C) to mobile devices. M4C is an electronic math tutor authored by Dr. Philip Yasskin and Dr. Douglas Meade that guides students through randomly generated problems and gives feedback at each intermediate step [18]. Rather than building native apps for each device, the goal was to create web apps that could be accessed anywhere on any platform. Since M4C relies on Maple's math language and CAS for math evaluation, accepting student input is relatively easy from the Maplet windowing system's text fields. However, it still insists on using Maple's input language, which many students find cryptic. Furthermore, few web-based math input solutions exist, and the available plugins will not work on mobile devices.

Maplets for Calculus and MathLex are supported in part by the NSF Division of Undergraduate Education (DUE) Course Curriculum and Laboratory Improvement Program (CCLI) grants 0737209 (Meade) and 0737248 (Yasskin); and Transforming Undergraduate Education in Science (TUES) grants 1123170 (Meade) and 1123255 (Yasskin).

# CHAPTER II

# MATHLEX FOR THE NOVICE (STUDENT & INSTRUCTOR)

## Introduction

MathLex aims to create a natural language for mathematical entry, and the Web is its primary target. Uniform entry across multiple platforms is another cornerstone to this project: MathLex should be easy and natural to use from a computer, from a tablet, and from a smartphone. Therefore, to kickstart this application, the MathLex Language specification has been implemented as a JavaScript plugin for websites. Webkit- and Gecko-based browsers (e.g. Google Chrome, Apple Safari, and Mozilla Firefox) have been the primary target browsers as they are the most widely used browsers and provide the best support for new Web technologies. The MathLex plugin has been successfully tested on Windows, Macintosh, and Linux operating systems and on multiple Apple iOS and Google Android devices.

The entirety of this and the next chapter has been made available as public documentation at the following web address: `http://ugrthesis.mathlex.org`. Much of the information here is also available in chapter IV, but this chapter is intended to be a non-technical reference for end-users (student and perhaps instructor) in contrast to developers seeking to implement the parser for the MathLex Language or write a renderer for the MathLex JavaScript implementation.

**Language Specification for the Masses**

*Summary*

This section describes the syntax used to enter mathematical content into MathLex. All input consists of *Tokens*, or strings of characters representing mathematical symbols or quantities. For example, the token '`>=`' represents the mathematical symbol $\geq$ and the concept "greater than or equal to". Tokens in the input string may be separated by spaces, but MathLex is intelligent enough to automatically separate tokens in *most* cases. MathLex looks for tokens in a greedy fashion in which it tries to match the largest token possible. For example, 5!=120 would be interpreted as $5 \neq 120$ even though the intended meaning might have been $5! = 120$. Therefore, it is necessary to insert spaces to separate tokens that might be part of other tokens. See page 44 for more detailed information about this issue.

The basic types of tokens in MathLex are Numbers (further subdivided into Integers and Floats/Decimals), Identifiers (further subdivided into Keywords and Variables), Constants, and Operators. After these basic types are defined, the collection of all tokens is presented in two sets of tables: the first is organized by how symbols are used in mathematics (e.g. binary operators, relations, etc.); the second is organized by the topic (e.g. calculus, set theory, etc.). The second set of tables is redundant but included for clarity.

Numbers

A *Number* is exactly as it seems: `42`, `3.14`, etc. Scientific notation is also allowed: `5e-2`, `3.0E8`. In either case, decimal points do not need a leading or trailing zero: `.5`, `78.`, `9.E-4`, `.22e7`.

Note that negative numbers are treated as a negation operation on the positive value of the number (consistent with algebraic notation). Likewise, fractions are treated as division operations on whole numbers.

Identifiers

An *Identifier* is an upper- or lowercase letter followed by any number of upper- or lowercase letters, numbers, or underscores (_). All of the following are valid identifiers: `x`, `A`, `b0`, `my_var`, `infinity`, `union`, and `arccos`. Identifiers fall into two categories: reserved and unreserved. Reserved identifiers, also called keywords, may be synonyms for certain constants or operators or may be the names of known functions. Each keyword for a constant or operator has its own token. They are listed in Table II.1 and again later in the tables of constant and operator tokens. The keywords for known functions are all assigned as *TIdentifier* tokens and are treated as general functions by the parser. They only get treated as specific functions by the translators and renderers. They are listed in Table II.2 and again later in the table of functions and occasionally in the tables of constant and operator tokens. Unreserved identifiers may be used as variables or user-defined functions. Of the above identifiers, `infinity` is a constant keyword, `union` is an operator keyword and `arccos` is a known function keyword, while the rest are valid variable or user-defined function names.

**Table II.1** Reserved Constant and Operator Keywords

| | | | |
|---|---|---|---|
| and | as | congruent | divides |
| equiv | exists | false | forall |
| if | iff | impliedby | implies |
| in | infinity | intersect | minus |
| mod | ndivide | ndivides | nequiv |
| not | notdivide | notdivides | onlyif |
| or | para | parallel | perp |
| perpendicular | propersubset | propersuperset | propersupset |
| propsubset | propsuperset | propsupset | psubset |
| psuperset | psupset | sim | similar |
| subset | superset | supset | then |
| true | union | unique | when |
| whenever | xor | | |

**Table II.2** Reserved Function Name Keywords

| | | | | | |
|---|---|---|---|---|---|
| `abs` | `acos` | `acosh` | `acot` | `acoth` | `acsc` |
| `acsch` | `arccos` | `arccosh` | `arccot` | `arccoth` | `arccsc` |
| `arccsch` | `arcsec` | `arcsech` | `arcsin` | `arcsinh` | `arctan` |
| `arctanh` | `asec` | `asech` | `asin` | `asinh` | `atan` |
| `atanh` | `C` | `ceil` | `ceiling` | `cos` | `cosh` |
| `cot` | `coth` | `csc` | `csch` | `curl` | `diff` |
| `div` | `exp` | `floor` | `gamma` | `grad` | `int` |
| `int` | `Integral` | `integral` | `Intersect` | `lim` | `limit` |
| `ln` | `log` | `P` | `pdiff` | `prod` | `product` |
| `root` | `sec` | `sech` | `sin` | `sinh` | `sqrt` |
| `sum` | `tan` | `tanh` | `Union` | | |

Constants

Similar to Identifiers, *Constants* are globally defined values or constructs. In MathLex, constants are usually typed as a number sign (`#`; also called *hash*, *sharp*, or *pound*) followed by the name of the constant. For example in MathLex, one would type `#pi` (or `#p` for short) to represent "pi" ($\pi$) and `#R` to represent the set of real numbers ($\mathbb{R}$). See the table of Constants below for a comprehensive list.

Operators

An *Operator* is just a catch-all term for any symbol that is not a Number, Identifier, or Constant, but is generally a mathematical operation or delimiter. With the exception of a few reserved keywords, operators usually consist of a few non-alphanumeric characters. Some operators start with an ampersand (`&`) to distinguish them from similar symbols. Some mathematical operators can be represented in multiple ways in MathLex. The following tables outline all mathematical operators understood by MathLex and all ways to represent them. Pick your favorite.

The numbers in the *Precedence* column of operator tables reflect which operations are more tightly bound (e.g. the "Parentheses-Exponents-Multiplication-Division-Addition-Subtraction (PEMDAS)" order of operations from grade school mathematics). Operators of higher precedence (or greater numeric value) will be identified and grouped before operations with lower precedence (or lesser numeric value). Operators of equal precedence will be grouped as they are encountered according to their associativity.

For unary and binary operators, the precedence number in the Precedence/Associativity (P/A) column is followed by an indicator of *Associativity*, i.e. how chained operations would be bound together:

- Left-associative operators (L) will be grouped from left to right (like subtraction and division): $a - b - c - d = ((a - b) - c) - d$
- Right-associative operators (R) will be grouped from right to left (like exponents): `a^b^c^d` $= a^{b^{c^d}} = a^{(b^{(c^d)})}$
- Non-associative operators (N) cannot be chained. For example, the triple dot product &v `a` &. &v `b` &. &v `c` $= \vec{a} \cdot \vec{b} \cdot \vec{c}$ does not make any sense since the result of a dot product is a scalar.
- Associative operators (like addition and multiplication) may be considered left- or right-associative without loss of meaninng. However, MathLex handles such operators as left-associative for definiteness.

At present, MathLex cannot chain relations, so they are regarded as non-associative.

**A special note about Functions.** Functions receive special treatment in that a majority of them are tokenized initially as unreserved identifiers and then interpreted after being parsed, but some functions have special tokens and syntax. Traditionally, functions are identifiers appended with a parenthesized list of parameters, e.g. `f(x,y,z)`. Some functions like sum, product, and limit have alternate notations that closely mimic handwritten notation and are thus called "written syntax". For example, the traditional CAS-like function to represent

$\sum_{x=0}^{n} \frac{1}{x}$ in MathLex is `sum(1/x, x, 0, n)`, and MathLex's alternate written syntax is `&sum &_(x=0) &^n 1/x`. Both are accepted by MathLex.

Function operators like composition and builder notation, e.g. $(f \circ g + h)(x)$, are allowed, so function application is parsed as a parenthetical postfix. See the note below Table II.4 for more information.

*Symbols by Type*

**Table II.3** Constants

| Name | Symbol | Code | Description |
|---|---|---|---|
| Pi | $\pi$ | `#pi`, `#p` | $3.14\cdots$ |
| Tau | $\tau$ | `#tau` | $2\pi \approx 6.28\cdots$ |
| E | e | `#e` | $2.718\cdots$, Natural Base, Euler-Napier number |
| Gamma | $\gamma$ | `#gamma` | $0.577\cdots$, Euler-Mascheroni constant |
| Infinity | $\infty$ | `#infinity`, `infinity` | `ERROR: memory overflow` |
| Imaginary Unit | $i$ | `#i` | $\sqrt{-1}$ |
| True | $\mathbf{T}$ | `#T`, `#true`, `true` | Case-insensitive |
| False | $\mathbf{F}$ | `#F`, `#false`, `false` | Case-insensitive |
| Natural Numbers | $\mathbb{N}$ | `#N` | |
| Integer Ring | $\mathbb{Z}$ | `#Z` | |
| Rational Field | $\mathbb{Q}$ | `#Q` | |
| Real Field | $\mathbb{R}$ | `#R` | |
| Complex Field | $\mathbb{C}$ | `#C` | |
| Quaternion Ring | $\mathbb{H}$ | `#H` | Hamilton numbers |
| Octonion Algebra | $\mathbb{O}$ | `#O` | Cayley numbers, Type "Oh". |
| Universal Set | $\mathbb{U}$ | `#U` | |
| Empty Set | $\emptyset$ | `#empty`, `{}` | |
| Zero Vector | $\vec{0}$ | `#v0` | |
| $x$ Unit Vector | $\hat{\imath}$ | `#ui`, `#vi` | |
| $y$ Unit Vector | $\hat{\jmath}$ | `#uj`, `#vj` | |
| $z$ Unit Vector | $\hat{k}$ | `#uk`, `#vk` | |
| Zero Matrix | $\mathbf{0}$ | `#0` | Type "zero". |
| Unit Matrix | $\mathbf{I}$ | `#1` | Identity Matrix, Type "one". |

**Table II.4** Unary Operators

| **Name** | **Symbol** | **Code** | **Description** | **P/A** |
|---|---|---|---|---|
| Positive | $+a$ | `+a` | | 17R |
| Negative | $-a$ | `-a` | | 17R |
| Positive/Negative | $\pm a$ | `+/- a`, `&pm a` | | 17R |
| Negative/Positive | $\mp a$ | `-/+ a`, `&mp a` | | 17R |
| Square Root | $\sqrt{a}$ | `sqrt(a)` | | * |
| Absolute Value | $\|a\|$ | `abs(a)` | | * |
| Factorial | $n!$ | `n!` | | 21L |
| Natural Exponential | $\exp(a)$ | `exp(a)` | | * |
| Natural Logarithm | $\ln(a)$ | `ln(a)` | | * |
| Real Part | $\Re a$ | `&Re a` | | 17R |
| Imaginary Part | $\Im a$ | `&Im a` | | 17R |
| Not | $\neg p$ | `not p, ~p, !p` | Logical Negation | 17R |
| Prime derivative | $f'$ | `f'` | Derivative w.r.t. $x$, 1st, or only var | 21L |
| Dot derivative | $\dot{f}$ | `f.` | Derivative w.r.t. $t$ or second var | 21L |
| Change | $\Delta x$ | `&D x` | Coordinate Difference | 17N |
| Differential | $\mathrm{d}x$ | `&d x` | | 17N |
| Partial Differential | $\partial x$ | `&pd x` | | 17N |
| Vector | $\vec{a}$ | `&v a` | | 17N |
| Unit Vector | $\hat{a}$ | `&u a` | | 17N |
| Gradient | $\vec{\nabla} f$, $\mathrm{grad}(f)$ | `&del f`, `grad(f)` | | 17L |
| Divergence | $\vec{\nabla} \cdot F$, $\mathrm{div}(F)$ | `&del. F`, `div(F)` | | 17N |
| Curl | $\vec{\nabla} \times F$, $\mathrm{curl}(F)$ | `&delx F`, `curl(F)` | | 17L |

In general, prefix operators are **right-associative** and postfix operators are **left-associative**.

* Although not listed, a pair of parentheses, when used as a function application, may be considered a postfix unary operator. As such, it is **left-associative** and has a precedence of **18**, just below that of function composition and exponents.

**Table II.5** Binary Operators

| Name | Symbol | Code | Description | P/A |
|---|---|---|---|---|
| Plus | $a+b$ | `a+b` | Addition | 9L |
| Minus | $a-b$ | `a-b` | Subtraction | 9L |
| Plus/Minus | $a \pm b$ | `a+/-b, a &pm b` | | 9L |
| Minus/Plus | $a \mp b$ | `a-/+b, a &mp b` | | 9L |
| Times | $a \cdot b$ | `a*b` | Multiplication | 14L |
| Divided by | $\frac{a}{b}$, $a/b$ | `a/b`, `a &/ b` | Division | 14L |
| Power | $a^b$ | `a^b`, `a**b` | Exponentiation | 20L |
| $n$-th Root | $\sqrt[n]{a}$ | `root(a, n)` | | * |
| Logarithm with Base | $\log_b a$ | `log(a, b)` | | * |
| Ratio | $p : q$ | `p&:q` | | 8N |
| Modulus | $a \pmod{n}$ | `a%n`, `a mod n` | | 14L |
| Combination | $\binom{n}{r}$ | `&C(n,r)`, `combination(n,r)` `n choose r` | Binomial Coefficient choose; ; `comb` for short | 15N* |
| Permutation | $P(n,r)$, | `&P(n,r)`, `permutation` | `perm` for short | * |
| Function Composition | $f \circ g$ | `f @ g` | | 19L |
| Function Repeated Composition | $f^{\circ n}$ | `f @@ n` | not implemented | 20R |
| Dot Product | $\vec{a} \cdot \vec{b}$ | `&v a &. &v b` | | 15N |
| Cross Product | $\vec{a} \times \vec{b}$ | `&v a &x &v b` | | 16L |
| Wedge Product | $\mathrm{d}x \wedge \mathrm{d}y$ | `&d x &w &d y` | | 16L |
| Tensor Product | $T \otimes S$ | `T &ox S` | | 16L |
| Cartesian Product | $A \times B$ | `A &* B`, `A &x B` | | 16L |
| Direct Sum | $A \oplus B$ | `A &o+ B` | | 11L |
| Subscript | $a_b$ | `a &_b` | Indexing | 22L |
| Multiple Subscript | $a_{i,j,k}$ | `a &_[i,j,k]` | | 22L |
| Superscript | $a^b$ | `a &^b` | Indexing | 22L |
| Multiple Superscript | $a^{i,j,k}$ | `a &^[i,j,k]` | | 22L |
| Mixed Subscripts and Superscripts | $T^{i}{}_{j}{}^{k}$ | `T &^i &_j &^k` | Tensor Indexing | 22L |
| Union | $A \cup B$ | `A union B` | | 12L |
| Intersection | $A \cap B$ | `A intersect B` | | 13L |
| Set Difference | $A \setminus B$ | `A \ B, A minus B` | | 10L |

**Table II.6** Logical Connectives and Quantifiers

| Name | Symbol | Code | Description | P/A |
|---|---|---|---|---|
| And | $p \wedge q$ | `p && q`, `p and q` | Conjunction | 5L |
| Or | $p \vee q$ | `p \|\| q`, `p or q` | Disjunction | 3L |
| Exclusive Or | $p \veebar q$ | `p xor q` | Exclusion | 4L |
| Implies | $p \to q$ | `p -> q`, `p implies q`, `p onlyif q`, `if p then q` | Conditional | 2L |
| Implied By | $p \leftarrow q$ | `p <- q`, `p impliedby q`, `p if q`, `p when q`, `p whenever q`, | Reverse Conditional | 2L |
| If And Only If | $p \leftrightarrow q$ | `p <-> q`, `p iff q` | Biconditional | 1N |
| Such That | $p : q$ | `p:q` | Used with set builder and quantifiers | |
| Universal Quantifier | $\forall x$ we have $P(x)$ $\forall x : Q(x)$ we have $P(x)$ | `forall x->P(x)` `forall x:Q(x)->P(x)` | "For all ..." | 6L |
| Existential Quantifier | $\exists x : Q(x)$ | `exists x : Q(x)` | "There exists ... such that" | 6L |
| Unique Quantifier | $\exists! x : Q(x)$ | `unique x : Q(x)` | "There exists a unique ... such that" | 6L |

**Table II.7** Relations

| **Name** | **Symbol** | **Code** | **Prec.** |
|---|---|---|---|
| Equal | $a = b$ | `a = b`, `a == b` | 7 |
| Not Equal | $a \neq b$ | `a /= b`, `a != b`, `a <> b` | 7 |
| Less than | $a < b$ | `a < b` | 7 |
| Greater than | $a > b$ | `a > b` | 7 |
| Less than or Equal | $a \leq b$ | `a <= b` | 7 |
| Greater than or Equal | $a \geq b$ | `a >= b` | 7 |
| Divides | $p \mid q$ | `p\|q`, `p divides q` | 7 |
| Not Divides | $p \nmid q$ | `p /\| q`, `p ~\| q`, `p ndivides q`, `p ndivide q`<br>`p notdivides q`, `p notdivide q` | 7 |
| Ratio Equality | $a : b :: c : d$ | `a&:b :: c&:d`, `a&:b as c&:d` | 7 |
| Congruent | $A \cong B$ | `A ~= B`, `A congruent B` | 7 |
| Similar | $A \sim B$ | `A ~B`, `A sim B`, `A similar B` | 7 |
| Parallel | $A \parallel B$ | `A para B`, `A parallel B` | 7 |
| Perpendicular | $A \perp B$ | `A perp B`, `A perpendicular B` | 7 |
| Subset | $A \subseteq B$ | `A subset B` | 7 |
| Superset | $A \supseteq B$ | `A superset B`, `A supset B` | 7 |
| Proper Subset | $A \subset B$ | `A propersubset B`, `A propsubset B`, `A psubset B` | 7 |
| Proper Superset | $A \supset B$ | `A propersuperset B`, `A propsuperset B`,<br>`A psuperset B`, `A propersupset B`,<br>`A propsupset B`, `A psupset B` | 7 |
| Inclusion | $a \in A$ | `a in A` | 7 |
| Equivalent | $a \equiv b$ | `a === b`, `a equiv b` | 0 |
| Not Equivalent | $a \not\equiv b$ | `a /== b`, `a !== b`, `a nequiv b` | 0 |

As previously stated, all relations are **non-associative** since `a = b = c = d` is *NOT* the same as `((a = b) = c) = d` or `a = (b = (c = d))`. Later versions of MathLex may support such expressions as `a = b = c = d` to be "syntactic sugar" for `(a = b) and (b = c) and (c = d)`.

**Table II.8** Delimiters and Indexing

| Name | Symbol | Code | Description |
|---|---|---|---|
| Parentheses | $(\ )$ | `( )` | Order of operation |
| Curly Braces | $\{\ \}$ | `{ }` | Sets |
| Square Brackets | $[\ ]$ | `[ ]` | Lists |
| Angle Brackets | $\langle\ \rangle$ | `< >`, `<:  :>` | Vectors |
| Matrix | $[\langle\ \rangle, \langle\ \rangle]$ | `[< >, < >]`, `[<:  :>, <:  :>]` | Row of Columns |
|  | $\langle[\ ], [\ ]\rangle$ | `<[ ], [ ]>`, `<:[ ], [ ]:>` | Column of Rows |
| Vertical Bars | $\vert\ \vert$ | `\| \|`, `\|:  :\|` | Absolute Value, Length, Determinant, Norm |
| Double Bars | $\Vert\ \Vert$ | `\|\| \|\|`, `\|\|:  :\|\|` | Length, Norm |
| Floor | $\lfloor x \rfloor$ | `floor(x)` | |
| Ceiling | $\lceil x \rceil$ | `ceil(x)`, `ceiling(x)` | |
| Such That | $p : q$ | `p:q` | Used with set builder and quantifiers |
| List Separator | , | `,` | |
| Subscript | $a_b$ | `a &_b` | Indexing |
| Multiple Subscript | $a_{i,j,k}$ | `a &_[i,j,k]` | |
| Superscript | $a^b$ | `a &^b` | Indexing |
| Multiple Superscript | $a^{i,j,k}$ | `a &^[i,j,k]` | |
| Mixed Subscripts and Superscripts | ${T^i}_j{}^k$ | `T &^i &_j &^k` | Tensor Indexing |
| Open Interval | $(a, b)$ | `(:a,b:)` | Exclusive Range Delimiters |
| Closed Interval | $[a, b]$ | `[:a,b:]` | Inclusive Range Delimiters |
| Half-Open Interval | $[a, b)$ | `[:a,b:)` | Mixed Range Delimiters |
| Bra-Ket Notation | $\langle A \mid B \rangle$ | `<:A\|B:>`, `<A\|\|B>` | |
| Bra | $\langle A \vert$ | `<A\|` | |
| Ket | $\vert B \rangle$ | `\|B>` | |

Note that some delimiters have more than one format either with or without colons. Namely, absolute value can be written as `| |` or `|:  :|`, norm can be written as `|| ||` or `||: :||`, and vectors can be surrounded by either `< >` or `<: :>`. Those with colons are *matched pairs* and should be used whenever there might be a chance of confusion about pairing. Those without colons are *context-sensitive* in that they have multiple meanings and therefore may not be automatically matched by the Lexer. Additionally, if an expression is opened with one type of delimiter, it must be closed with the same type (i.e. matched vs. context-sensitive).

All delimiters have “infinite” precedence; any and all contents will be grouped together.

**Table II.9** Functions

| Name | Symbol | Code | Description |
|---|---|---|---|
| Trig | $\sin(\theta), \ldots$ | `sin(theta)`, ... | Also cos, tan,<br>cot, sec, csc |
| Inverse Trig | $\arcsin(x), \ldots$ | `arcsin(x)`, `asin(x)`, ... | Also arccos, acos, arctan, atan<br>arccot, acot, arcsec, asec,<br>arccsc, acsc |
| Hyperbolic Trig | $\sinh(\lambda), \ldots$ | `sinh(lambda)`, ... | Also cosh, tanh,<br>coth, sech, csch |
| Inv. Hyp. Trig | $\operatorname{arcsinh}(x), \ldots$ | `arcsinh(x)`, `asinh(x)`, ... | Also arccosh, acosh, arctanh, atanh,<br>arccoth, acoth, arcsech, asech,<br>arccsch, acsch |
| Absolute Value | $\lvert a \rvert$ | `abs(a)` | |
| Floor | $\lfloor x \rfloor$ | `floor(x)` | |
| Ceiling | $\lceil x \rceil$ | `ceil(x)`, `ceiling(x)` | |
| Square Root | $\sqrt{a}$ | `sqrt(a)` | |
| $n$th Root | $\sqrt[n]{a}$ | `root(a, n)` | |
| Natural Exponential | $\exp(a)$ | `exp(a)` | |
| Natural Logarithm | $\ln(a)$ | `ln(a)` | |
| Logarithm with Base | $\log_b a$ | `log(a, b)` | |
| Combination | $\binom{n}{r}$ | `C(n,r)` | Binomial Coefficient<br>choose |
| Permutation | $P(n, r)$ | `P(n,r)` | |
| Limit | $\lim\limits_{x \to a} f(x)$ | `lim(f(x), x, a)`<br>`&lim &_(x -> a) f(x)` | Also `limit`, `Lim`, `Limit` |
| Derivative | $\frac{\mathrm{d}}{\mathrm{d}x}\left(f(x)\right)$ | `diff(f(x), x)`<br>`&df(x)/&dx` | |
| Partial Derivative | $\frac{\partial}{\partial x}\left(f(x,y)\right)$ | `pdiff(f(x,y), x)`<br>`&pdf(x)/&pdx` | |
| Indefinite Integral | $\int f(x)\,\mathrm{d}x$ | `int(f(x),x)`<br>`&int f(x) &dx` | Also `Int`, `integral`, `Integral` |
| Definite Integral | $\int_a^b f(x)\,\mathrm{d}x$ | `int(f(x),x,a,b)`<br>`&int &_a &^b f(x) &dx` | (see note above) |
| Sum Over Range | $\sum\limits_{i=m}^{n} a_i$ | `sum(a&_i,i,m,n)`<br>`&sum &_(i=m) &^n a&_i` | Also `Sum` |
| Sum Over Set | $\sum\limits_{i \in T} a_i$ | `sum(a&_i, i in T)`<br>`&sum &_(i in T) &^n a&_i` | (see note above) |
| Product Over Range | $\prod\limits_{i=m}^{n} a_i$ | `prod(a&_i,m,n)`<br>`&prod &_(i = m) &^n a&_i` | Also `product`, `Prod`, `Product` |
| Product Over Set | $\prod\limits_{i \in T} a_i$ | `prod(a&_i, i in T)`<br>`&prod &_(i in T) &^n a&_i` | (see note above) |
| Union Over Range | $\bigcup\limits_{i=m}^{n} S_i$ | `Union(S&_i, i, m, n)`<br>`&Union &_(i=m) &^n S&_i` | |
| Union Over Set | $\bigcup\limits_{i \in T} S_i$ | `Union(S&_i, i in T)`<br>`&Union &_(i in T) S&_i` | |
| Intersection Over Range | $\bigcap\limits_{i=m}^{n} S_i$ | `Intersect(S&_i, i, m, n)`<br>`&Intersect &_(i=m) &^n S&_i` | |
| Intersection Over Set | $\bigcap\limits_{i \in T} S_i$ | `Intersect(S&_i, i in T)`<br>`&Intersect &_(i in T) S&_i` | |

Repetition can lead to discrepancy, and this section is already quite repetetive. Please refer to the tables above for precedence and associativity information. These tables are provided merely for convenience when attempting to find a particular token. Hence it is redundant to provide extra information.

**Table II.10** Arithmetic

| **Name** | **Symbol** | **Code** | **Description** |
|---|---|---|---|
| Plus, Positive | $+$ | `+` | binary or unary |
| Minus, Negative | $-$ | `-` | binary or unary |
| Plus/Minus | $\pm$ | `+/-`, `&pm` | binary or unary |
| Minus/Plus | $\mp$ | `-/+`, `&mp` | binary or unary |
| Times | $\cdot$ | `*` | Multiplication |
| Divided by | $\frac{a}{b}$, $a/b$ | `a/b`, `a&/b` | Division |
| Power | $a^b$ | `a^b`, `a**b` | Exponentiation |
| Square Root | $\sqrt{a}$ | `sqrt(a)` | |
| $n$-th Root | $\sqrt[n]{a}$ | `root(a,n)` | |
| Log Base n | $\log_n a$ | `log(a,n)` | |
| Natural Exponential | $\exp(a), e^a$ | `exp(a)`, `#e^a` | |
| Natural Logarithm | $\ln(a)$ | `ln(a)` | |
| Absolute Value | $\lvert a\rvert$ | `\|a\|`, `\|:a:\|`, `abs(a)` | |
| Factorial | $n!$ | `n!` | |
| Imaginary Unit | $i$ | `#i` | $\sqrt{-1}$ |
| Real Part | $\Re a$ | `&Re a` | |
| Imaginary Part | $\Im a$ | `&Im a` | |
| Ratio | $a : b$ | `p&:q` | |
| Ratio Equality | $a : b :: c : d$ | `a&:b :: c&:d`, `a&:b as c&:d` | |
| Equal | $=$ | `=`, `==` | |
| Not Equal | $\neq$ | `/=`, `!=`, `<>` | |
| Less Than | $<$ | `<` | |
| Greater Than | $>$ | `>` | |
| Less Than or Equal | $\leq$ | `<=` | |
| Greater Than or Equal | $\geq$ | `>=` | |
| Parentheses | $(\quad)$ | `( )` | |

**Table II.11** Algebra

| Name | Symbol | Code | Description |
|---|---|---|---|
| Natural Numbers | $\mathbb{N}$ | `#N` | |
| Integer Ring | $\mathbb{Z}$ | `#Z` | |
| Rational Field | $\mathbb{Q}$ | `#Q` | |
| Real Field | $\mathbb{R}$ | `#R` | |
| Complex Field | $\mathbb{C}$ | `#C` | |
| Function Composition | $f \circ g$ | `f @ g` | |
| Function Repeated Composition | $f^{\circ n}$ | `f @@ n` | not implemented |
| Sum Over Range | $\displaystyle\sum_{i=m}^{n} a_i$ | `sum(a&_i,i,m,n)`<br>`&sum &_(i=m) &^n a&_i` | Also `Sum` |
| Sum Over Set | $\displaystyle\sum_{i\in T} a_i$ | `sum(a&_i, i in T)`<br>`&sum &_(i in T) a&_i` | (see note above) |
| Product Over Range | $\displaystyle\prod_{i=m}^{n} a_i$ | `prod(a&_i,m,n)`<br>`&prod &_(i=m) &^n a&_i` | Also `product`, `Prod`, `Product` |
| Product Over Set | $\displaystyle\prod_{i\in T} a_i$ | `prod(a&_i, i in T)`<br>`&prod &_(i in T) a&_i` | (see note above) |

**Table II.12** Geometry

| Name | Symbol | Code | Description |
|---|---|---|---|
| Pi | $\pi$ | `#pi`, `#p` | $3.14\cdots$ |
| Tau | $\tau$ | `#tau` | $2\pi \approx 6.28\cdots$ |
| Open Interval | $(a,b)$ | `(:a,b:)` | Exclusive Range Delimiters |
| Closed Interval | $[a,b]$ | `[:a,b:]` | Inclusive Range Delimiters |
| Half-Open Intervals | $[a,b)$ | `[:a,b:)` | Mixed Range Delimiters |
| Congruent | $\cong$ | `~=`, `congruent` | |
| Similar | $\sim$ | `~`, `sim`, `similar` | |
| Parallel | $\parallel$ | `parallel` | |
| Perpendicular | $\perp$ | `perp`, `perpendicular` | |
| Vector Components | $\langle a,b,c\rangle$ | `<a,b,c>`, `<:a,b,c:>` | |
| Vector | $\vec{a}$ | `&v a` | |
| Unit Vector | $\hat{a}$ | `&u a` | |
| Vector Length | $\lvert\vec{a}\rvert$<br>$\lVert\vec{a}\rVert$ | `\|&v a\|`, `\|:&v a:\|`,<br>`\|\|&v a\|\|`, `\|\|:&v a:\|\|` | |
| Zero Vector | $\vec{0}$ | `#v0` | |
| $x$ Unit Vector | $\hat{\imath}$ | `#ui`, `#vi` | |
| $y$ Unit Vector | $\hat{\jmath}$ | `#uj`, `#vj` | |
| $z$ Unit Vector | $\hat{k}$ | `#uk`, `#vk` | |
| Dot Product | $\vec{a}\cdot\vec{b}$ | `&v a &. &v b` | |
| Cross Product | $\vec{a}\times\vec{b}$ | `&v a &x &v b` | |

**Table II.13** Trigonometry

| Name | Symbol | Code | Description |
|---|---|---|---|
| Trig | $\sin(\theta), \ldots$ | `sin(theta)`, ... | Also cos, tan,<br>cot, sec, csc |
| Inverse Trig | $\arcsin(x), \ldots$ | `arcsin(x)`, `asin(x)`, ... | Also arccos, acos, arctan, atan<br>arccot, acot, arcsec, asec,<br>arccsc, acsc |
| Hyperbolic Trig | $\sinh(\lambda), \ldots$ | `sinh(lambda)`, ... | Also cosh, tanh,<br>coth, sech, csch |
| Inv. Hyp. Trig | $\text{arcsinh}(x), \ldots$ | `arcsinh(x)`, `asinh(x)`, ... | Also arccosh, acosh, arctanh, atanh,<br>arccoth, acoth, arcsech, asech,<br>arccsch, acsch |

**Table II.14** Discrete

| Name | Symbol | Code | Description |
|---|---|---|---|
| Natural Numbers | $\mathbb{N}$ | `#N` | |
| Integer Ring | $\mathbb{Z}$ | `#Z` | |
| Factorial | $n!$ | `n!` | |
| Floor | $\lfloor x \rfloor$ | `floor(x)` | |
| Ceiling | $\lceil x \rceil$ | `ceil(x)`, `ceiling(x)` | |
| Modulus | $a \pmod{n}$ | `a%n`, `a mod n` | |
| Divides | $p \mid q$ | `p\|q` | |
| Not Divides | $p \nmid q$ | `p /\| q`, `p ~\| q`,<br>`p ndivides q`, `p ndivide q`<br>`p notdivides q`, `p notdivide q` | |
| Combination | $\binom{n}{r}$ | `C(n,r)` | Binomial Coefficient<br>choose |
| Permutation | $P(n,r)$ | `P(n,r)` | |
| Sum Over Range | $\sum_{i=m}^{n} a_i$ | `sum(a&_i,i,m,n)`<br>`&sum &_(i=m) &^n a&_i` | Also `Sum` |
| Sum Over Set | $\sum_{i \in T} a_i$ | `sum(a&_i, i in T)`<br>`&sum &_(i in T) &^n a&_i` | (see note above) |
| Product Over Range | $\prod_{i=m}^{n} a_i$ | `prod(a&_i,m,n)`<br>`&prod &_(i = m) &^n a&_i` | Also `product`,<br>`Prod`, `Product` |
| Product Over Set | $\prod_{i \in T} a_i$ | `prod(a&_i, i in T)`<br>`&prod &_(i in T) &^n a&_i` | (see note above) |
| Union Over Range | $\bigcup_{i=m}^{n} S_i$ | `Union(S&_i, i, m, n)`<br>`&Union &_(i=m) &^n S&_i` | |
| Union Over Set | $\bigcup_{i \in T} S_i$ | `Union(S&_i, i in T)`<br>`&Union &_(i in T) S&_i` | |
| Intersection Over Range | $\bigcap_{i=m}^{n} S_i$ | `Intersect(S&_i, i, m, n)`<br>`&Intersect &_(i=m) &^n S&_i` | |
| Intersection Over Set | $\bigcap_{i \in T} S_i$ | `Intersect(S&_i, i in T)`<br>`&Intersect &_(i in T) S&_i` | |

**Table II.15** Calculus

| Name | Symbol | Code | Description |
|---|---|---|---|
| Pi | $\pi$ | `#pi, #p` | $3.14\cdots$ |
| Tau | $\tau$ | `#tau` | $2\pi \approx 6.28\cdots$ |
| E | e | `#e` | $2.718\cdots$, Natural Base, Euler-Napier number |
| Gamma | $\gamma$ | `#gamma` | $0.577\cdots$, Euler-Mascheroni constant |
| Infinity | $\infty$ | `#infinity`, `infinity` | `ERROR: Memory overflow` |
| Limit | $\lim_{x\to a} f(x)$ | `lim(f(x), x, a)` `&lim &_(x -> a) f(x)` | Also `limit`, `Lim`, `Limit` |
| Derivative | $\frac{\mathrm{d}}{\mathrm{d}x}\left(f(x)\right)$ | `diff(f(x), x)` `&df(x)/&dx` | |
| Partial Derivative | $\frac{\partial}{\partial x}\left(f(x,y)\right)$ | `pdiff(f(x,y), x)` `&pdf(x)/&pdx` | |
| Prime derivative | $f'$ | `f'` | Derivative w.r.t. $x$ or 1st/only var. |
| Dot derivative | $\dot{f}$ | `f.` | Derivative w.r.t. $t$ or 2nd var. |
| Change | $\Delta x$ | `&D x` | Coordinate Difference |
| Differential | $\mathrm{d}x$ | `&d x` | |
| Partial Differential | $\partial x$ | `&pd x` | |
| Riemann Sum | $\sum_{i=1}^{n} f(x_i)\,\Delta x_i$ | `sum(f(x&_i)*&Dx&_i,i,1,n)` `&sum &_(i=1) &^n f(x&_i)*&Dx&_i` | Also `Sum` |
| Indefinite Integral | $\int f(x)\,\mathrm{d}x$ | `int(f(x),x)` `&int f(x) &dx` | Also `Int`, `integral`, `Integral` |
| Definite Integral | $\int_a^b f(x)\,\mathrm{d}x$ | `int(f(x),x,a,b)` `&int &_a &^b f(x) &dx` | (see note above) |
| Infinite Series | $\sum_{i=1}^{\infty} a_i$ | `sum(a&_i,i,1,infinity)` `&sum &_(i=1) &^infinity a&_i` | (see note above) |
| Gradient | $\vec{\nabla} f$, $\mathrm{grad}(f)$ | `&del f`, `grad(f)` | |
| Divergence | $\vec{\nabla}\cdot F$, $\mathrm{div}(F)$ | `&del. F`, `div(F)` | |
| Curl | $\vec{\nabla}\times F$, $\mathrm{curl}(F)$ | `&delx F`, `curl(F)` | |

**Table II.16** Logic

| Name | Symbol | Code | Description |
|---|---|---|---|
| True | **T** | `#T`, `#true`, `true` | |
| False | **F** | `#F`, `#false`, `false` | |
| And | $p \wedge q$ | `p && q`, `p and q` | Conjunction |
| Or | $p \vee q$ | `p \|\| q`, `p or q` | Disjunction |
| Exclusive Or | $p \veebar q$ | `p xor q` | Exclusion |
| Not | $\neg p$ | ∼`p`, `!p`, `not p` | Logical Negation |
| Implies | $p \to q$ | `p -> q`, `p implies q`, `p onlyif q`, `if p then q` | Conditional |
| Implied By | $p \leftarrow q$ | `p <- q`, `p impliedby q`, `p if q`, `p when q`, `p whenever q`, | Reverse Conditional |
| If And Only If | $p \leftrightarrow q$ | `p <-> q`, `p iff q` | Biconditional |
| Equivalent | $\equiv$ | `===`, `equiv` | |
| Not Equivalent | $\not\equiv$ | `/==`, `!==`, `nequiv` | |
| Universal Quantifier | $\forall x$ we have $P(x)$ $\forall x : Q(x)$ we have $P(x)$ | `forall x->P(x)` `forall x:Q(x)->P(x)` | "For all ..." |
| Existential Quantifier | $\exists x : Q(x)$ | `exists x : Q(x)` | "There exists ...such that" |
| Unique Quantifier | $\exists! x : Q(x)$ | `unique x : Q(x)` | "There exists a unique ...such that" |

**Table II.17** Set Theory

| Name | Symbol | Code | Description |
|---|---|---|---|
| Set Delimiters | $\{\ \}$ | `{ }` | |
| Such That | $p:q$ | `p:q` | Used with set builder and quantifiers |
| Universal Set | $\mathbb{U}$ | `#U` | |
| Empty Set | $\emptyset$ | `#empty`, `{}` | |
| Natural Numbers | $\mathbb{N}$ | `#N` | |
| Integer Ring | $\mathbb{Z}$ | `#Z` | |
| Rational Field | $\mathbb{Q}$ | `#Q` | |
| Real Field | $\mathbb{R}$ | `#R` | |
| Complex Field | $\mathbb{C}$ | `#C` | |
| Quaternion Ring | $\mathbb{H}$ | `#H` | Hamilton numbers |
| Octonion Algebra | $\mathbb{O}$ | `#O` | Cayley numbers, Type "Oh". |
| Subset | $A \subseteq B$ | `A subset B` | |
| Superset | $A \supseteq B$ | `A superset B`, `A supset B` | |
| Proper Subset | $A \subset B$ | `A propersubset B`, `A propsubset B`<br>`A psubset B` | |
| Proper Superset | $A \supset B$ | `A propersuperset B`, `A psupset B`<br>`A propsuperset B`, `A propsupset B`<br>`A psuperset B`, `A propersupset B` | |
| Inclusion | $a \in A$ | `in` | |
| Union | $A \cup B$ | `A union B` | |
| Intersection | $A \cap B$ | `A intersect B` | |
| Set Difference | $A \setminus B$ | `A \ B`, `A minus B` | |
| Cartesian Product | $A \times B$ | `A &* B`, `A &x B` | |
| Direct Sum | $A \oplus B$ | `A &o+ B` | |
| Union Over Range | $\bigcup_{i=m}^{n} S_i$ | `Union(S&_i, i, m, n)`<br>`&Union &_(i=m) &^n S&_i` | |
| Union Over Set | $\bigcup_{i\in T} S_i$ | `Union(S&_i, i in T)`<br>`&Union &_(i in T) S&_i` | |
| Intersection Over Range | $\bigcap_{i=m}^{n} S_i$ | `Intersect(S&_i, i, m, n)`<br>`&Intersect &_(i=m) &^n S&_i` | |
| Intersection Over Set | $\bigcap_{i\in T} S_i$ | `Intersect(S&_i, i in T)`<br>`&Intersect &_(i in T) S&_i` | |

**Table II.18** Linear Algebra

| Name | Symbol | Code | Description |
|---|---|---|---|
| Vector Delimiters | $\langle\ \rangle$ | `< >`, `<:   :>` | |
| Zero Vector | $\vec{0}$ | `#v0` | |
| $x$ Unit Vector | $\hat{\imath}$ | `#vi` | |
| $y$ Unit Vector | $\hat{\jmath}$ | `#vj` | |
| $z$ Unit Vector | $\hat{k}$ | `#vk` | |
| Matrix | $[\langle\ \rangle, \langle\ \rangle]$<br>$\langle[\ ], [\ ]\rangle$ | `[< >, < >]`, `[<:   :>, <:   :>]`<br>`<[ ], [ ]>`, `<:[ ], [ ]:>` | Row of Columns<br>Column of Rows |
| Zero Matrix | $\mathbf{0}$ | `#0` | Type "zero". |
| Unit Matrix | $\mathbf{I}$ | `#1` | Identity Matrix, Type "one". |

### How MathLex Works

MathLex works in two phases. The first phase compiles a MathLex expression into an Abstract Syntax Tree (AST) that can be represented in memory, and the second phase converts the AST into some type of output.

#### *Input to Syntax Tree*

When provided with a valid MathLex string, `MathLex.parse()` produces an *abstract syntax tree (AST)* representing the inferred value of the MathLex code. Under the hood, this first phase has two components: a preprocessor called a **Tokenizer** and then the main **Parser**.

The **Tokenizer** is responsible for translating the characters in the MathLex input string into a list of *Tokens*, a way to group related characters into a single symbol. For example, "`<=`" is shorthand for "less than or equal to" (in display math, '$\leq$') and is comprised of two separate characters. The Tokenizer groups these characters into a `TLessEqual` Token for the parser. A list of all Tokens is given in Grammars T1 through T7 of Chapter IV.

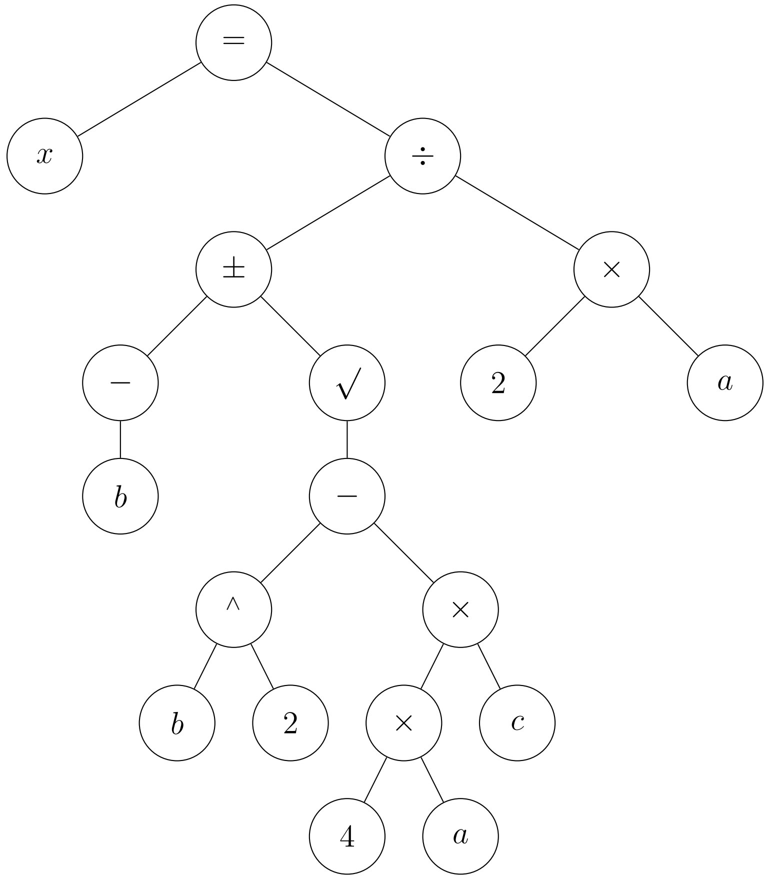
The **Parser** then reads the list of tokens and assembles the corresponding AST. The AST is built from different "node" types represented as a recursive array. Every node has a string name indicating the type of node, and optionally one or more subnodes for its arguments. The grammar rules used by the parser are given in Grammars L1 through L4 of Chapter IV.

For example, the MathLex input for the quadratic formula,

$$x = \frac{-b \pm \sqrt{b^2 - 4a\,c}}{2a}$$

is `x = (-b +/- sqrt(b^2-4*a*c))/(2*a)`. This is an equation, so the root node is an equality ($=$), and its two subnodes are an identifier ($x$) and a quotient ($\div$), which is further broken down into its subnodes as displayed in Figure II.1.

**Fig. II.1.** AST for the Quadratic Formula



*Syntax Tree to Output*

The AST returned by the parser gives a mathematically faithful model of the meaning behind the interpreted input text. It is evaluated correctly by evaluating each nodes' children and then performing the parent node operation on the child values (this is called a *recursive postorder traversal*). Such tools to recursively evaluate the AST are called **Translators** or **Renderers**. These terms are used interchangeably in this thesis. So far, translators have been written for LaTeX, the Sage CAS (partially), and a textual version of the AST. The author plans to write additional translators (Maple, Mathematica, and MathML, for example), and volunteers willing and able to help write such translators are welcome.

**LaTeX Translator.** Using the quadratic formula example in Figure II.1, one could build a LaTeX translator from the following rules:

- An equality is represented as "`LHS = RHS`"
- Variables and numbers are expressed as-is
- A fraction is represented as "`\frac{NUMERATOR}{DENOMINATOR}`"
- Plus-or-Minus is represented as "`LHS \pm RHS`"
- Negation is represented as "`-SUBEXPR`"
- Square Roots are represented as "`\sqrt{SUBEXPR}`"
- Subtraction is represented as "`LHS - RHS`"
- Exponents are represented as "`BASE^{POWER}`". Note the braces around the exponent.
- Multiplication is represented as a space between operands: "`LHS \, RHS`"

This `latex` translator would start at the root node: since it is an equality, the translator will translate the left-hand-side (LHS) and the right-hand-side (RHS) and then put an equals sign (`=`) between them. The LHS is a variable ($x$), so its translated value would be `x`. The RHS is a quotient, and the numerator and denominator will each have to be translated before they can be entered into the LaTeX fraction command. The translator will continue until all sub-nodes are translated, and then the root node's translation will be returned as

```
x = \frac{-b \pm \sqrt{b^{2} - 4 \, a \, c}}{2 \, a}.
```

**Sage Translator.** The `sage` translator works similarly and returns the following line of code:

```
x == ( (?  PlusMinus ?)  )/(2*a)
```

Note that Sage does not support the Plus/Minus operation and therefore cannot be accurately translated. Future support for this operation may split the returned Sage expression into two forms: one plus, and the other minus. If the `+/-` operator is replaced by a `+`, then the Sage renderer returns

```
x == (-b + sqrt(b^(2) - 4 * a * c))/(2 * a)
```

**Text-Tree Renderer.** The `text-tree` renderer yields the output in Listing II.1.

**Listing II.1** Sample Text-Tree Output

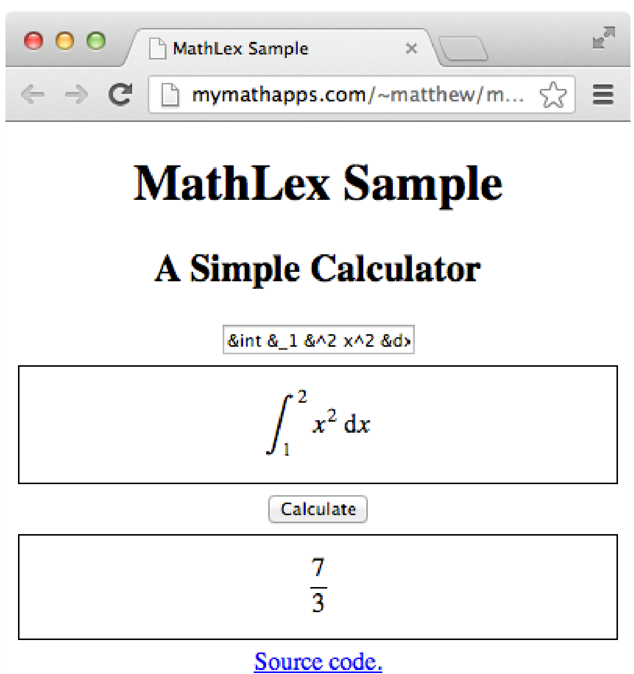
```
Equal
    Variable: x
    Divide
        PlusMinus
            Negative
                Variable: b
            Function
                Builder:
                    Variable: sqrt
                Arguments:
                    Minus
                        Exponent
                            Variable: b
                            Literal: 2
                        Times
                            Times
                                Literal: 4
                                Variable: a
                            Variable: c
        Times
            Literal: 2
            Variable: a
```

# CHAPTER III

# MATHLEX FOR THE INSTRUCTOR

This chapter is meant to be a quick lesson on how to use the MathLex JavaScript library provided on the companion website: `http://ugrthesis.mathlex.org`. If you would like more information on how MathLex works internally, please see Chapter II. For more information on how MathLex works internally, see Chapter IV. In this chapter, the reader will be guided in building a sample page that contains a simple calculator powered by Sage Cell (Aleph) server [19]. It may be viewed at the companion website: `http://ugrthesis.mathlex.org/quick-start/mathlexsample.html` (shown in Figure III.1)

**Fig. III.1.** MathLex Simple Calculator




Before continuing, the reader should have basic knowledge of HTML, JavaScript, and how web pages function. More information on these subjects can be obtained easily from sites such as the Mozilla Developer Network [20], TutsPlus [21], and W3Schools [22]. Before building the page, please download the `mathlex.js` file from the companion website and place it in a directory that will be accessible from the web page to be created.

### Sample Page Source Code

The entire, self-contained source code for the sample Sage calculator is given in Listing III.1. Each section of code is explained in-depth below.

**Listing III.1** Sample Page

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="utf-8">
    <title>MathLex Sample</title>
    <style>
      body { text-align: center; }
      #math-display, #math-output { border: 1px solid #000; margin: 5px 0; }
    </style>
  </head>
  <body>
    <h1>MathLex Sample</h1>
    <h2>A Simple Calculator</h2>
    <input id="math-input" type="text" placeholder="Type math here">
    <div id="math-display">\[ \]</div>
    <input id="send-math" type="button" value="Calculate">
    <div id="math-output">\[ \]</div>

    <script src="javascripts/mathlex.js"></script>
    <script src="http://ajax.googleapis.com/ajax/libs/jquery/1.7.2/jquery.min.js"></script>
    <script src="http://cdn.mathjax.org/mathjax/latest/MathJax.js?config=TeX-AMS-MML_HTMLorMML"></script>
    <script>
      $(document).ready(function () {
        // get MathJax output object
        var mjDisplayBox, mjOutBox;
        MathJax.Hub.Queue(function () {
          mjDisplayBox = MathJax.Hub.getAllJax('math-display')[0];
          mjOutBox = MathJax.Hub.getAllJax('math-output')[0];
        });

        // "live update" MathJax whenever a key is pressed
        $('#math-input').on('keyup', function (evt) {
          var math = $(this).val();
          $(this).css('color', 'black');

          if (math.length > 0) {
            try {
              var tree = MathLex.parse(math),
                  latex = MathLex.render(tree, 'latex');
              MathJax.Hub.Queue(['Text', mjDisplayBox, latex]);
            } catch (err) {
              $(this).css('color', 'red');
            }
          } else {
            // clear display and output boxes if input is empty
            MathJax.Hub.Queue(['Text', mjDisplayBox, '']);
            MathJax.Hub.Queue(['Text', mjOutBox, '']);
          }
        });

        // send output to sage server
        $('#send-math').on('click', function (evt) {
          var math = $('#math-input').val();
          if (math.length > 0) {
            try {
              var tree = MathLex.parse(math),
                  sageCode = MathLex.render(tree, 'sage');
              $.post('http://aleph.sagemath.org/service?callback=?',
                     { code: 'print latex('+sageCode+')' }, function (data) {
                // HACK: Firefox does not convert data to JSON.
                if (typeof(data) === 'string') { data = $.parseJSON(data); }
                // AJAX success callback
                if (data.success) {
                  MathJax.Hub.Queue(['Text', mjOutBox, data.stdout]);
                } else {
                  MathJax.Hub.Queue(['Text', mjOutBox,
                        '\\text{Sage could not understand that input}']);
                }
              });
            } catch (err) {
              MathJax.Hub.Queue(['Text', mjOutBox,
                    '\\text{Check your syntax and try again}']);
            }
          }
        });
      });
    </script>
  </body>
</html>
```

*Page Layout*

The HTML snippet in Listing III.2 creates the layout: The first two lines (12 and 13) create a header, then the input field (named `math-input`) is below on line 14, the following `<div>` tag on line 15 creates a preview window that will be rendered by MathJax, the second `<input>` tag on line 16 makes the submit button, and finally the last line, 17, creates the output window that will also be rendered by MathJax.

**Listing III.2** Sample Page Layout

```
12 <h1>MathLex Sample</h1>
13 <h2>A Simple Calculator</h2>
14 <input id="math-input" type="text" placeholder="Type math here">
15 <div id="math-display">\[ \]</div>
16 <input id="send-math" type="button" value="Calculate">
17 <div id="math-output">\[ \]</div>
```

*JavaScript Inclusions*

To be able to process the math input, the MathLex JavaScript file must be included in the HTML, which is done on line 19. The author recommends putting JavaScript inclusions just before the closing `</body>` tag, but the reader may choose to put it in the `<head>` or elsewhere. Note that the `src` attribute should be replaced by the appropriate path to the reader's copy of the MathLex JavaScript file.

If the reader plans to use MathJax [3], jQuery [23], MooTools [24], Prototype [25], YUI [26], Dojo [27], or another JavaScript toolkit/library, please refer to the corresponding site for installation instructions. This example uses the jQuery library and MathJax, so lines 20 and 21 load jQuery and MathJax from their respective Content Distribution Network (CDN) URLs:

**Listing III.3** Sample Page JS Inclusions

```
<script src="javascripts/mathlex.js"></script>
<script src="http://ajax.googleapis.com/ajax/libs/jquery/1.7.2/
    jquery.min.js"></script>
<script src="http://cdn.mathjax.org/mathjax/latest/
    MathJax.js?config=TeX-AMS-MML_HTMLorMML"></script>
```

*Handling MathJax Output*

The math input from the text field line 14 is processed in two ways:

1. MathLex automatically parses it and translates it into LaTeX, which MathJax displays in the `math-display` window on line 15.
2. When the Calculate button on line 16 is clicked, MathLex parses it, translates it into Sage, and transmits it to a Sage Cell server. The result returned by Sage is rendered by MathJax into the `math-output` window on line 17.

To interface with MathJax, the output objects, `mjDisplayBox` and `mjOutBox`, must be defined. These tell MathJax where to put the output, namely the `math-display` box on line 15 and the `math-output` box on line 17. This is done in lines 24 to 29.

**Listing III.4** Sample Page MathJax Objects

```
24 // get MathJax output object
25 var mjDisplayBox, mjOutBox;
26 MathJax.Hub.Queue(function () {
27   mjDisplayBox = MathJax.Hub.getAllJax('math-display')[0];
28   mjOutBox = MathJax.Hub.getAllJax('math-output')[0];
29 });
```

*Live-Updating Math Display*

To automatically parse the `math-input` from line 14, we use jQuery's DOM event handling systems in lines 31 through 49. Line 32 watches for a `keyup` event in the `math-input` box.

When this occurs, line 33 stores the value of the `math-input` as a variable `math`. Line 36 checks if math has a non-zero length. If it does, then in lines 37 through 39, MathLex tries to parse it into an AST structure and render that structure into LaTeX. Then in line 40, MathJax tries to display the LaTeX code in the `mjDisplayBox` (previously linked to the `math-display` box). If this fails, then the `math-input` box turns red. Here this represents the `math-input` box that is handling the `keyup` event. If `math` is empty (i.e. has zero length), then lines 45 to 47 blank out the `math-display` and `math-output` boxes.

**Listing III.5** Sample Page Live Math Display Update

```
31 // "live update" MathJax whenever a key is pressed
32 $('#math-input').on('keyup', function (evt) {
33   var math = $(this).val();
34   $(this).css('color', 'black');
35
36   if (math.length > 0) {
37     try {
38       var tree = MathLex.parse(math),
39           latex = MathLex.render(tree, 'latex');
40       MathJax.Hub.Queue(['Text', mjDisplayBox, latex]);
41     } catch (err) {
42       $(this).css('color', 'red');
43     }
44   } else {
45     // clear display and output boxes if input is empty
46     MathJax.Hub.Queue(['Text', mjDisplayBox, '']);
47     MathJax.Hub.Queue(['Text', mjOutBox, '']);
48   }
49 });
```

*Sending Math to Sage*

The last thing to do is listen to the **Calculate** button to send the `math-input` from line 14 to a Sage processor and display the result in the `math-output` box. We again use jQuery's DOM event handling system in lines 51 through 70. Line 52 watches for a click event in the `send-math` button. When this occurs, line 53 stores the value of the `math-input` as a variable `math`. Line 54 checks if `math` has a non-zero length, and if it does, then, in lines 56–57, MathLex tries to parse the math into an AST, translate it into Sage code, and then store the result in a variable appropriately named `sageCode`. Then in lines 58–59, the `sageCode` is sent to a Sage server as an Asynchronous (AJAX) request. When the browser receives the AJAX response, it executes the associated function, which is receiving the associated data (more about this later). If the Sage execution was successful, then, on line 64, MathJax displays the result (`data.stdout`) in the `mjOutBox` object (previously linked to the `math-output` box). If the server encountered an error, then lines 66–67 display "Sage could not understand that input" in the `mjOutBox`. If anything else fails (most likely a syntax error), lines 71–72 display "Check your syntax and try again" in the `mjOutBox`.

**Listing III.6** Sample Page Sage Submission

```
51 // send output to sage server
52 $('#send-math').on('click', function (evt) {
53   var math = $('#math-input').val();
54   if (math.length > 0) {
55     try {
56       var tree = MathLex.parse(math),
57           sageCode = MathLex.render(tree, 'sage');
58       $.post('http://aleph.sagemath.org/service?callback=?',
59              {code: 'print latex('+sageCode+')'}, function (data) {
60         // HACK: Firefox does not convert data to JSON.
61         if (typeof(data) === 'string') { data = $.parseJSON(data); }
62         // AJAX success callback
63         if (data.success) {
64           MathJax.Hub.Queue(['Text', mjOutBox, data.stdout]);
65         } else {
66           MathJax.Hub.Queue(['Text', mjOutBox,
67               '\\text{Sage could not understand that input}']);
68         }
69       });
70     } catch (err) {
71       MathJax.Hub.Queue(['Text', mjOutBox,
72           '\\text{Check your syntax and try again}']);
73     }
74   }
75 });
```

*Sage Processing*

Finally, a few words on the Sage processing code on lines 58–64: This sample page communicates with a Sage Cell server at `http://aleph.sagemath.org` operated by the Sage Math organization. The HTTP POST request sends the Sage code `print latex(<sageCode>)`, where `<sageCode>` is the injected code generated by MathLex. Sage will evaluate the sent `sageCode`, simplify the result, convert it to LaTeX, and print it to standard output (hence `stdout`). This output is passed back to the client in the form of a JSON object in Listing III.7. This value is stored as the `data` parameter to the AJAX callback function; so `data.success` on line 63 yields true or false and `data.stdout` yields the output string in LaTeX form (assuming `data.success` is true) to be processed by MathJax on line 64.

**Listing III.7** JSON response from Sage Cell

```
{
  success: true|false,
  stdout: 'output string'
}
```

Finally, the Firefox web browser does not automatically convert the string response into a JSON object. Rather, the `data` parameter is left as a string representation of the JSON data. Therefore, line 61 is a necessary "hack" to make Firefox properly handle the JSON response data.

### Additional Comments

- Both the *live-update* and *Sage-submission* callbacks follow the same abstract structure:

    1. get Mathalex code from `math-input` text field
    2. parse the MathLex code into an AST
    3. translate the AST into another format (e.g. LaTeX or Sage code)
    4. do something with the translated code

- The value of `math-input` was obtained on line 53 using a jQuery command of the form "`var math = $('#' + inputID).val();`", where `inputID` is the input field. In the example, this was "`math-input`". There are many ways to obtain a text field's value; here are the corresponding code snippets for standard JavaScript and each of the libraries mentioned earlier:

  1. Standard JavaScript: `var math = document.getElementById(inputID).value;`
  2. MooTools: `var math = document.id(inputID).value;` or `var math = $(inputID).value;` (The `$` function is aliased to `document.id`).
  3. Prototype: `var math = $(inputID).value;`
  4. YUI: `var math = Y.one('#' + inputID).get('value');`
  5. Dojo: `var math = dom.byId(inputID).value;`

  After any of these, `math` now contains the MathLex input value, but any name is acceptible so long as it is not a JavaScript reserved keyword [28].

- MathLex input is passed to `MathLex.parse()`, which returns an AST. To improve performance, inputs should be parsed only once if possible, although this was not done in the sample page above to keep the code simple. The AST can be used multiple times without unnecessary overhead of reinterpreting the input's meaning.

- An AST can be rendered into several formats with `MathLex.render(ast, format)`, where `format` is the name of a built-in renderer or translator. Three such translators are included by default:

  – `latex`: for use in typesetting LaTeX (perhaps using MathJax)
  – `sage`: input language for the open-source Sage CAS
  – `text-tree`: plain-text, indented tree representation of the AST (for debugging)

  These translators simply walk through the tree recursively, performing a certain action at each node of the AST. For more information about how renderers/translators work, please refer to page 27 in Chapter II. Instructions on how to create a renderer/translator are given on page 54 in Chapter IV.

- The LaTeX and Sage translators are demonstrated in the example above, so here is an example using the `text-tree` renderer: In your HTML, place a "`<pre id="text-tree-output"></pre>`" tag somewhere in the `<body>`, and then include the lines of JavaScript in Listing III.8.

**Listing III.8** Rendering a Text-Tree

```
var treeCode = MathLex.render(syntaxTree, 'text-tree');
document.getElementById('text-tree-output').innerHTML = treeCode;
```

# CHAPTER IV
# MATHLEX FOR THE PROGRAMMER

Unlike human languages such as English, computer languages (known in the computer science field as *formal languages*) are strictly defined and often have no flexibility for ambiguity or exceptions. This rigidity allows the rules governing a formal language to be encoded as a *grammar*, which may then be written as a computer program.

Since MathLex is a language that is meant to be understood by a computer, its syntax must be put forth in a grammar that can be written as a program. At the same time, MathLex is also meant to be understood by human beings, so the language must be expressive and closely resemble classical handwritten math notation.

Before introducing the MathLex grammar, one must have a background in grammar theory and notation.

## Grammar Basics and Theory

If you are already familiar with Grammars and BNF/EBNF notation, you may skip to the next section.

The MathLex language is specified in a subset of the Extended Backus-Naur Form (EBNF) grammar notation [29], an extension of the simpler Backus-Naur Form (BNF) [30]. BNF and EBNF provide a systematic representation of rules that define a valid, syntactically correct statement by using two types of symbols: non-terminal and terminal. Before discussing the "extensions" provided by EBNF, one should know the fundamentals of BNF notation.

Non-terminal symbols are surrounded by angle brackets $\langle\,\rangle$ and may be expanded into any combination of terminal and non-terminal symbols.

Terminal symbols are quoted literals of text that would be typed by the user.

BNF is simple and best understood by examples. Take a look at Grammar 1, which identifies any string with zero or more `a`'s.

**Grammar 1** Zero or more `a`'s

$\langle \mathit{start} \rangle$ ::= $\langle a \rangle$

$\langle a \rangle$ ::= '`a`' $\langle a \rangle$
 | $\epsilon$

The special non-terminal symbol $\langle \mathit{start} \rangle$ defines the entry point into the grammar. In this case, $\langle \mathit{start} \rangle$ is an alias for $\langle a \rangle$, and thus the grammar could have been represented as a single rule. However for clarity, $\langle \mathit{start} \rangle$ will only be an alias for entry points into the grammar.

Each rule in this grammar has a single non-terminal symbol on the left side of the ::= "expansion operator" and any combination of non-terminal and terminal symbols on the right side. Grammars of this format are called *context-free* grammars [31]. The | "alternate operator" denotes an alternate expansion for the rule. Alternates may be defined in-line or on a new line. In the example above, the non-terminal symbol $\langle a \rangle$ has two valid expansions:

1. the terminal symbol `'a'` followed by the same non-terminal symbol $\langle a \rangle$, or

2. the empty string terminal symbol represented by $\epsilon$.

Notice that this rule's expansion includes itself. Rules of this nature are called *directly recursive*. Rules may also be *indirectly recursive* like in Grammar 2, which recognizes any alternating pattern of `a`'s and `b`'s.

**Grammar 2** Alternating `a`'s and `b`'s

$\langle start \rangle ::= \langle a \rangle \mid \langle b \rangle$

$\langle a \rangle ::=$ '`a`' $\langle b \rangle$
$\mid \; \epsilon$

$\langle b \rangle ::=$ '`b`' $\langle a \rangle$
$\mid \; \epsilon$

The non-terminal symbols $\langle a \rangle$ and $\langle b \rangle$ reference each other in their expansions, thus they are indirectly recursive.

*Extended Backus-Naur Form*

All of the examples so far have been in standard BNF notation. EBNF has the following modifications that make grammars easier to type, read, and encode as plain text [29].

- non-terminal symbols are not enclosed in angle brackets
- the expansion operator is simply = instead of ::=
- rules are terminated by a semi-colon (;)
- symbols in an expansion are concatenated by a comma (,)
- repeated sequences may be surrounded by braces { } instead of using recursion
- optional sequences may be surrounded by brackets [ ] instead of using alternates
- "sub-expressions" may be grouped with parentheses ( ) instead of creating and referencing a new non-terminal symbol
- special "named" terminal symbols may be described between question marks
- comments are placed between (* and *) delimiters.

*Modified EBNF*

EBNF offers much flexibility that makes encoding grammars easier. However, for the purposes of this thesis, only the repetition, option, and grouping delimiters are used. In addition, basic regular expressions will be used to define special classes of symbols where appropriate instead of the question marks notation of EBNF. To clarify any confusion about the "repetition" delimiters, { } will represent "one or more", whereas [{ }] will represent "optionally one or more" or simply "zero or more".

Each of the grammars in the previous section can be rewritten in our subset of EBNF as in Grammars 3 and 4.

**Grammar 3** Modified EBNF encoding of Grammar 1

$\langle start \rangle$ ::= $\langle a \rangle$

$\langle a \rangle$ ::= [{ 'a' }]

**Grammar 4** Modified EBNF encoding of Grammar 2

$\langle start \rangle$ ::= $\langle a \rangle$ | $\langle b \rangle$

$\langle a \rangle$ ::= ['a' $\langle b \rangle$]

$\langle b \rangle$ ::= ['b' $\langle a \rangle$]

OR

$\langle start \rangle$ ::= $\langle ab \rangle$

$\langle ab \rangle$ ::= ['a'] [{ 'ba' }] ['b']

### MathLex Grammar

*MathLex Token Grammar*

In MathLex, a mathematical symbol will be called a *token* and is entered as a sequence of characters, which are the terminal symbols. Some tokens may be entered as several different sequences of characters and some can have different meanings when used in different contexts. All of the MathLex tokens are defined in Grammars T1 through T7 and are detected by a lightweight parser called a *tokenizer*. Each token may be considered a non-terminal symbol, but the convention to prevent confusion with other grammar rules will be to prefix a terminal token by a capital 'T' (for "token") and 'CamelCase' its name.

**Grammar T1** MathLex Alphanumeric Tokens

⟨*letter*⟩ ::= /a-zA-Z/

⟨*digit*⟩ ::= /0-9/

⟨*TIntegerLiteral*⟩ ::= { ⟨*digit*⟩ }

⟨*TFloatLiteral*⟩ ::= [{ ⟨*digit*⟩ }] '`.`' { ⟨*digit*⟩ } [ ('`E`' | '`e`') [ ('`+`' | '`-`') ] { ⟨*digit*⟩ } ]
  | { ⟨*digit*⟩ } ('`E`' | '`e`') [ ('`+`' | '`-`') ] { ⟨*digit*⟩ }

⟨*TConstant*⟩ ::= '`false`' | '`true`' | '`infinity`'
  | '`#`' { (⟨*letter*⟩ | ⟨*digit*⟩) }

⟨*TIdentifier*⟩ ::= ⟨*letter*⟩ [{ ('`_`' | ⟨*letter*⟩ | ⟨*digit*⟩) }]

Unlike other tokens, the Literals, Constants, and Identifiers have values; so they are written with their value. For example, the integer token for `549` is written as *TIntegerLiteral*:`549`, the decimal token for `5.23e42` is written as *TFloatLiteral*:`5.23e42`, the constant token for `#pi` ($\pi$) is written as *TConstant*:`pi`, and the function arccos is written as *TIdentifier*:`arccos`

As briefly mentioned in Chapter II, MathLex's tokenizer is greedy in that it will try to find tokens of maximal length in a given input string. Whitespace is treated as a token delimiter and is otherwise ignored. For example, the tokenizer will treat the text '`whenabc123`' as a single *TIdentifier*:`whenabc123` token, but ‘`when abc 123`’ will result in the following token stream:

[ *TIf*, *TIdentifier*:`abc`, *TIntegerLiteral*:`123` ]

Similarly, the Tokenizer will treat the input `5!=120` as

[ *TIntegerLiteral*:`5`, *TNotEqual*, *TIntegerLiteral*:`120` ]

(equivalent to $5 \neq 120$), but treat the input `5! = 120` as

[ *TIntegerLiteral*:`5`, *TBang*, *TEqual*, *TIntegerLiteral*:`120` ]

(equivalent to $5! = 120$).

**Grammar T2** MathLex Logical Tokens

$\langle$*TQForall*$\rangle$ ::= ‘`forall`’

$\langle$*TQExists*$\rangle$ ::= ‘`exists`’

$\langle$*TQUnique*$\rangle$ ::= ‘`unique`’

$\langle$*TIff*$\rangle$ ::= ‘`<->`’ | ‘`iff`’

$\langle$*TImplies*$\rangle$ ::= ‘`->`’ | ‘`implies`’ | ‘`onlyif`’

$\langle$*TIf*$\rangle$ ::= ‘`<-`’ | ‘`if`’ | (‘`when`’ [ ‘`ever`’ ]) | ‘`impliedby`’

$\langle$*TThen*$\rangle$ ::= ‘`then`’

$\langle$*TAnd*$\rangle$ ::= ‘`&&`’ | ‘`and`’

$\langle$*TOr*$\rangle$ ::= ‘`||`’ | ‘`or`’

$\langle$*TXor*$\rangle$ ::= ‘`xor`’

$\langle$*TNot*$\rangle$ ::= ‘`not`’

$\langle$*TSuchThat*$\rangle$ ::= ‘`:`’

**Grammar T3** MathLex Relational Tokens

⟨*TEqual*⟩ ::= ‘=’ | ‘==’

⟨*TNotEqual*⟩ ::= ‘!=’ | ‘/=’ | ‘<>’

⟨*TLess*⟩ ::= ‘<’

⟨*TLessEqual*⟩ ::= ‘<=’

⟨*TGreater*⟩ ::= ‘>’

⟨*TGreaterEqual*⟩ ::= ‘>=’

⟨*TEquivalent*⟩ ::= ‘===’ | ‘equiv’

⟨*TNotEquivalent*⟩ ::= ‘!==’ | ‘/==’ | ‘nequiv’

⟨*TCongruent*⟩ ::= ‘~=’ | ‘congruent’

⟨*TSimilar*⟩ ::= ‘sim’ [ ‘ilar’ ]

⟨*TSubset*⟩ ::= ‘subset’

⟨*TProperSubset*⟩ ::= ‘p’ [ ‘rop’ [‘er’] ] ‘subset’

⟨*TSuperset*⟩ ::= ‘sup’ [‘er’] ‘set’

⟨*TProperSuperset*⟩ ::= ‘p’ [ ‘rop’ [‘er’]] ‘sup’ [‘er’] ‘set’

⟨*TIn*⟩ ::= ‘in’

⟨*TDivides*⟩ ::= ‘divides’

⟨*TNotDivides*⟩ ::= ‘/|’ | ‘~|’ | ‘n’ [‘ot’] ‘divide’ [‘s’]

⟨*TParallel*⟩ ::= ‘para’ [ ‘llel’ ]

⟨*TPerpendicular*⟩ ::= ‘perp’ [ ‘endicular’ ]

⟨*TRatio*⟩ ::= ‘&:’

⟨*TRatioEqual*⟩ ::= ‘::’ | ‘as’

**Grammar T4** MathLex Arithmetic Tokens

⟨*TPlus*⟩ ::= '`+`'

⟨*TMinus*⟩ ::= '`-`'

⟨*TPlusMinus*⟩ ::= '`+/-`' | '`&pm`'

⟨*TMinusPlus*⟩ ::= '`-/+`' | '`&mp`'

⟨*TTimes*⟩ ::= '`*`'

⟨*TDivide*⟩ ::= '`/`'

⟨*TSlashDiv*⟩ ::= '`&/`'

⟨*TExponent*⟩ ::= '`^`' | '`**`'

⟨*TModulus*⟩ ::= '`%`' | '`mod`'

⟨*TImaginary*⟩ ::= '`&Im`'

⟨*TReal*⟩ ::= '`&Re`'

⟨*TCompose*⟩ ::= '`@`'

⟨*TRepeatCompose*⟩ ::= '`@@`'

⟨*TUnion*⟩ ::= '`union`'

⟨*TIntersect*⟩ ::= '`intersect`'

⟨*TSetDifference*⟩ ::= '\' | '`minus`'

⟨*TCartesianProduct*⟩ ::= '`&*`'

⟨*TDirectSum*⟩ ::= '`&o+`'

⟨*TVectorizer*⟩ ::= '`&v`'

⟨*TUnitVectorizer*⟩ ::= '`&u`'

⟨*TSubscript*⟩ ::= '`&_`'

⟨*TSuperscript*⟩ ::= '`&^`'

⟨*TDot*⟩ ::= '`&.`'

⟨*TCross*⟩ ::= '`&x`'

⟨*TWedge*⟩ ::= '`&w`'

⟨*TTensor*⟩ ::= '`&ox`'

**Grammar T5** MathLex Delimiter Tokens

⟨*TLParen*⟩ ::= ‘(’

⟨*TRParen*⟩ ::= ‘)’

⟨*TLCurlyBrace*⟩ ::= ‘{’

⟨*TRCurlyBrace*⟩ ::= ‘}’

⟨*TLSquareBracket*⟩ ::= ‘[’

⟨*TRSquareBracket*⟩ ::= ‘]’

⟨*TLRngIncl*⟩ ::= ‘[:’

⟨*TRRngIncl*⟩ ::= ‘:]’

⟨*TLRngExcl*⟩ ::= ‘(:’

⟨*TRRngExcl*⟩ ::= ‘:)’

⟨*TLPipe*⟩ ::= ‘|:’

⟨*TRPipe*⟩ ::= ‘:|’

⟨*TLDoublePipe*⟩ ::= ‘||:’

⟨*TRDoublePipe*⟩ ::= ‘:||’

⟨*TLVector*⟩ ::= ‘<:’

⟨*TRVector*⟩ ::= ‘:>’

⟨*TListSep*⟩ ::= ‘,’

**Grammar T6** MathLex Differential Calculus Tokens

⟨*TPrimeDiff*⟩ ::= ‘'’

⟨*TDotDiff*⟩ ::= ‘`.`’

⟨*TChangeDelta*⟩ ::= ‘`&D`’

⟨*TDifferential*⟩ ::= ‘`&d`’

⟨*TPartial*⟩ ::= ‘`&pd`’

⟨*TGradient*⟩ ::= ‘`&del`’

⟨*TDivergence*⟩ ::= ‘`&del.`’

⟨*TCurl*⟩ ::= ‘`&delx`’

⟨*TSum*⟩ ::= ‘`&`’ (‘`s`’ | ‘`S`’) ‘`um`’

⟨*TProduct*⟩ ::= ‘`&`’ (‘`p`’ | ‘`P`’) ‘`rod`’ [‘`uct`’]

⟨*TLimit*⟩ ::= ‘`&`’ (‘`l`’ | ‘`L`’) ‘`im`’ [‘`it`’]

⟨*TDivDiff*⟩ ::= ‘`/&d`’

⟨*TDivPartial*⟩ ::= ‘`/&pd`’

⟨*TIntegral*⟩ ::= ‘`&`’ (‘`i`’ | ‘`I`’) ‘`nt`’ [‘`egral`’]

---

**Grammar T7** MathLex Miscellaneous Tokens

⟨*TTilde*⟩ ::= ‘`~`’

⟨*TPipe*⟩ ::= ‘`|`’

⟨*TBang*⟩ ::= ‘`!`’

---

The miscellaneous tokens are used in multiple contexts to mean different things. Their meaning is determined by the Parser based on their context.

*MathLex Language Grammar*

Grammars L1 through L4 outline all parts of the current MathLex language specification. In particular, they define the ways in which the tokens may be combined to form valid mathematical statements. In general, the rules are presented in order of increasing precedence. For more information about the precedence of each recognized operation, please refer to the tables in Chapter II. The names of each rule indicate when that operation *may* be identified. For example, a logical *disjunction* may be matched as such, or it may be expanded by the *exclusion* rule, which could then in turn be an exclusion operation or a conjunction, and so on.

**Grammar L1** MathLex Language Entry Rules

$\langle start \rangle$ ::= $\langle expression \rangle$

$\langle expression \rangle$ ::= $\langle logical \rangle$ [ ($\langle TEquivalent \rangle$ | $\langle TNotEquivalent \rangle$) $\langle logical \rangle$ ]

**Grammar L2** MathLex Language Logical Rules

⟨*logical*⟩ ::= ⟨*biconditional*⟩

⟨*biconditional*⟩ ::= ⟨*implication*⟩ ⟨*TIff*⟩ ⟨*implication*⟩

⟨*implication*⟩ ::= ⟨*reverse implication*⟩
  | { ⟨*disjunction*⟩ ⟨*TImplies*⟩ } ⟨*disjunction*⟩
  | ⟨*TIf*⟩ ⟨*disjunction*⟩ ⟨*TThen*⟩ ⟨*disjunction*⟩

⟨*reverse implication*⟩ ::= ⟨*disjunction*⟩ [{ ⟨*TIf*⟩ ⟨*disjunction*⟩ }]

⟨*disjunction*⟩ ::= [{ ⟨*exclusion*⟩ ⟨*TOr*⟩ }] ⟨*exclusion*⟩

⟨*exclusion*⟩ ::= [{ ⟨*conjunction*⟩ ⟨*TXor*⟩ }] ⟨*conjunction*⟩

⟨*conjunction*⟩ ::= [{ ⟨*negation*⟩ ⟨*TAnd*⟩ }] ⟨*negation*⟩

⟨*negation*⟩ ::= [(⟨*TNot*⟩ | ⟨*TTilde*⟩ | ⟨*TBang*⟩)] ⟨*quantification*⟩

⟨*quantification*⟩ ::= ⟨*relation*⟩
  | ⟨*TQForall*⟩ ⟨*relation*⟩ ⟨*TComma*⟩ ⟨*quantification*⟩
  | (⟨*TQExists*⟩ | ⟨*TQUnique*⟩) ⟨*relation*⟩ ⟨*TSuchThat*⟩ ⟨*quantification*⟩

⟨*relation*⟩ ::= ⟨*ratio*⟩ [ ⟨*TRatioEqual*⟩ ⟨*ratio*⟩ ]
  | ⟨*algebraic*⟩ (⟨*TEqual*⟩ | ⟨*TNotEqual*⟩ | ⟨*TCongruent*⟩ | ⟨*TSimilar*⟩ | ⟨*TTilde*⟩) ⟨*algebraic*⟩
  | ⟨*algebraic*⟩ (⟨*TParallel*⟩ | ⟨*TPerpendicular*⟩) ⟨*algebraic*⟩
  | ⟨*algebraic*⟩ (⟨*TLess*⟩ | ⟨*TLessEqual*⟩ | ⟨*TGreaterEqual*⟩ | ⟨*TGreater*⟩ ) ⟨*algebraic*⟩
  | ⟨*algebraic*⟩ (⟨*TSubset*⟩ | ⟨*TProperSubset*⟩ | ⟨*TSuperset*⟩ | ⟨*TProperSuperset*⟩ | ⟨*TDirectSum*⟩) ⟨*algebraic*⟩
  | ⟨*algebraic*⟩ ⟨*TIn*⟩ ⟨*algebraic*⟩
  | ⟨*algebraic*⟩ (⟨*TDivides*⟩ | ⟨*TPipe*⟩ | ⟨*TNotDivides*⟩) ⟨*algebraic*⟩

⟨*ratio*⟩ ::= ⟨*algebraic*⟩ [ ⟨*TRatio*⟩ ⟨*algebraic*⟩ ]

**Grammar L3** MathLex Language Algebraic Rules

⟨*algebraic*⟩ ::= ⟨*summation*⟩

⟨*summation*⟩ ::= [{ ⟨*composition*⟩ (⟨*TPlusMinus*⟩ | ⟨*TMinusPlus*⟩ | ⟨*TPlus*⟩ | ⟨*TMinus*⟩) }] ⟨*composition*⟩

⟨*composition*⟩ ::= [{ ⟨*set difference*⟩ ⟨*TCompose*⟩ }] ⟨*set difference*⟩

⟨*set difference*⟩ ::= [{ ⟨*set union*⟩ ⟨*TSetDifference*⟩ }] ⟨*set union*⟩

⟨*set union*⟩ ::= [{ ⟨*set intersection*⟩ ⟨*TUnion*⟩ }] ⟨*set intersection*⟩

⟨*set intersection*⟩ ::= [{ ⟨*product*⟩ ⟨*TIntersect*⟩ }] ⟨*product*⟩

⟨*multiplication*⟩ ::= [{ ⟨*dot product*⟩ (⟨*TTimes*⟩ | ⟨*TSlash*⟩ | ⟨*TDivide*⟩ | ⟨*TModulus*⟩) }] ⟨*dot product*⟩

⟨*dot product*⟩ ::= ⟨*vector product*⟩ ⟨*TDot*⟩ ⟨*vector product*⟩

⟨*vector product*⟩ ::= [{ ⟨*prefix*⟩ (⟨*TCross*⟩ | ⟨*TWedge*⟩ | ⟨*TTensor*⟩ | ⟨*TCartesianProduct*⟩) }] ⟨*prefix*⟩

⟨*prefix*⟩ ::= [{ (⟨*TNot*⟩ | ⟨*TPlus*⟩ | ⟨*TMinus*⟩ | ⟨*TPlusMinus*⟩ | ⟨*TMinusPlus*⟩) }] ⟨*function*⟩
| [ (⟨*TPartial*⟩ | ⟨*TDifferential*⟩ | ⟨*TChangeDelta*⟩ | ⟨*TVectorizor*⟩ | ⟨*TUnitVectorizer*⟩) ] ⟨*function*⟩

⟨*function*⟩ ::= ⟨*exponent*⟩ [{ ⟨*TLParen*⟩ ⟨*expression*⟩ [{ ⟨*TComma*⟩ ⟨*expression*⟩ }] ⟨*TRParen*⟩ }]

⟨*exponent*⟩ ::= ⟨*suffix*⟩ [{ ⟨*TExponent*⟩ ⟨*prefix*⟩ }]

⟨*suffix*⟩ ::= ⟨*function*⟩ [{ (⟨*TBang*⟩ | ⟨*TPrime*⟩ | ⟨*TDotDiff*⟩ ) }]

⟨*index*⟩ ::= ⟨*primary*⟩ [{ (⟨*TSubscript*⟩ | ⟨*TSuperscript*⟩) ⟨*primary*⟩ }]

**Grammar L4** MathLex Language Primary Value Rules

⟨*primary*⟩ ::= ⟨*TEmpty*⟩ | ⟨*TIdentifier*⟩ | ⟨*TIntegerLiteral*⟩ | ⟨*TFloatLiteral*⟩ | ⟨*TConstant*⟩
  | ⟨*vector*⟩ | ⟨*absolute value*⟩ | ⟨*norm*⟩ | ⟨*bra ket*⟩
  | ⟨*TLCurlyBrace*⟩ ⟨*set*⟩ ⟨*TRCurlyBrace*⟩
  | ⟨*TLSquareBracket*⟩ [⟨*expression*⟩ [{ ⟨*TComma*⟩ ⟨*expression*⟩ }] ] ⟨*TRSquareBracket*⟩
  | (⟨*TLRngIncl*⟩ | ⟨*TLRngExcl*⟩) ⟨*algebraic*⟩ ⟨*TComma*⟩ ⟨*algebraic*⟩ (⟨*TRRngIncl*⟩ | ⟨*TRRngExcl*⟩)
  | ⟨*TLParen*⟩ [ ⟨*expression*⟩ ] ⟨*TRParen*⟩
  | ⟨*TIntegral*⟩ ⟨*integral bounds*⟩ ⟨*algebraic*⟩ ⟨*TDifferential*⟩ ⟨*algebraic*⟩

⟨*vector*⟩ ::= ⟨*TLess*⟩ ⟨*algebraic*⟩ [{ ⟨*TComma*⟩ ⟨*algebraic*⟩ }] ⟨*TGreater*⟩
  | ⟨*TLVector*⟩ ⟨*algebraic*⟩ [{ ⟨*TComma*⟩ ⟨*algebraic*⟩ }] ⟨*TRVector*⟩

⟨*absolute value*⟩ ::= ⟨*TPipe*⟩ ⟨*algebraic*⟩ ⟨*TPipe*⟩
  | ⟨*TLPipe*⟩ [ ⟨*algebraic*⟩ ] ⟨*TRPipe*⟩

⟨*norm*⟩ ::= ⟨*TOr*⟩ ⟨*algebraic*⟩ ⟨*TOr*⟩
  | ⟨*TLDoublePipe*⟩ [ ⟨*algebraic*⟩ ] ⟨*TRDoublePipe*⟩

⟨*bra ket*⟩ ::= ⟨*TLess*⟩ ⟨*algebraic*⟩ ⟨*TPipe*⟩
  | ⟨*TPipe*⟩ ⟨*algebraic*⟩ ⟨*TGreater*⟩
  | ⟨*TLess*⟩ ⟨*algebraic*⟩ ⟨*TOr*⟩ ⟨*algebraic*⟩ ⟨*TGreater*⟩
  | ⟨*TLVector*⟩ ⟨*algebraic*⟩ ⟨*TPipe*⟩ ⟨*algebraic*⟩ ⟨*TRVector*⟩

⟨*integral bounds*⟩ ::= ⟨*TSubscript*⟩ ⟨*primary*⟩ [ ⟨*TSuperscript*⟩ ⟨*primary*⟩ ]
  | ⟨*TSuperscript*⟩ ⟨*primary*⟩ [ ⟨*TSubscript*⟩ ⟨*primary*⟩ ]

### Building a Renderer

The AST returned by the JavaScript parser are represented as a recursive array: the first element (i.e. index 0) is a string ID of the node type, and the remaining elements are that node's parameters. Renderers should operate recursively on the AST, checking each node's ID and performing a corresponding action. Listing IV.1 shows some snippets from the built-in LaTeX translator. MathLex is programmed in CoffeeScript [32], a highly expressive language that compiles into JavaScript. The names of all nodes and their structure are given in the documentation on the companion website (`http://ugrthesis.mathlex.org`) as the list is not yet stable. The reader may also choose to copy the LaTeX translator and modify it.

**Listing IV.1** LaTeX Translator Snippets in CoffeeScript

```
exports.render = render = (ast) ->
    switch ast[0]
        when 'Plus' then "#{render ast[1]} + #{render ast[2]}"
        when 'Minus' then "#{render ast[1]} - #{render ast[2]}"
        when 'PlusMinus' then "#{render ast[1]} \\pm #{render ast[2]}"
        when 'MinusPlus' then "#{render ast[1]} \\mp #{render ast[2]}"
        when 'Times'
            op = if implMult(ast[1], LEFT) or implMult(ast[2], RIGHT)
                " \\, "
              else
                " \\cdot "
            (render ast[1]) + op + (render ast[2])
        when 'Divide'
            if ast[3]
                "\\frac{#{render unwrap ast[1]}}{#{render unwrap ast[2]}}"
            else
                "#{render ast[1]} / #{render ast[2]}"
        when 'Ratio' then "#{render ast[1]} : #{render ast[2]}"
        when 'Modulus' then "#{render ast[1]} \\pmod{#{render unwrap ast[2]}}"
        when 'Exponent' then "#{render ast[1]}^{#{render unwrap ast[2]}}"
        when 'Superscript'
            rhs = unwrap ast[2]
            if rhs[0] is 'List'
                elements = (render elem for elem in rhs[1])
                sup = elements.join " ,\\, "
            else
                sup = render rhs
            "#{render ast[1]}{}^{#{sup}}"
        when 'Subscript'
            rhs = unwrap ast[2]
            if rhs[0] is 'List'
                elements = (render elem for elem in rhs[1])
                sub = elements.join " ,\\, "
            else
                sub = render rhs
            "#{render ast[1]}{}_{#{sub}}"
        when 'DotProduct' then "#{render ast[1]} \\cdot #{render ast[2]}"
        when 'CrossProduct' then "#{render ast[1]} \\times #{render ast[2]}"
        when 'Union' then "#{render ast[1]} \\cup #{render ast[2]}"
        when 'Intersection' then "#{render ast[1]} \\cap #{render ast[2]}"
        when 'SetDiff' then "#{render ast[1]} \\setminus #{render ast[2]}"
        when 'DirectSum' then "#{render ast[1]} \\oplus #{render ast[2]}"
        when 'CartesianProduct' then "#{render ast[1]} \\times #{render ast[2]}"

        when 'Positive' then "+#{render ast[1]}"
        when 'Negative' then "-#{render ast[1]}"
        when 'PosNeg' then "\\pm #{render ast[1]}"
        when 'NegPos' then "\\mp #{render ast[1]}"
        when 'Partial' then "\\partial #{render ast[1]}"
        when 'Differential' then "\\mathrm{d} #{render ast[1]}"
        when 'Change' then "\\Delta #{render ast[1]}"
        when 'Gradient' then "\\vec\\nabla #{render ast[1]}"
        when 'Divergence' then "\\vec\\nabla \\cdot #{render ast[1]}"
        when 'Curl' then "\\vec\\nabla \\times #{render ast[1]}"
```

# CHAPTER V

# FUTURE DEVELOPMENTS

At present, MathLex does not encompass all of mathematics and probably never will. However, that should not stop us from making additions to the mathematical content. In addition, there is written syntax not yet implemented into the MathLex input language, and the ease of entering math could be further refined. This chapter discusses these future improvements.

## Processing Incomplete Input

The Grammar given in Chapter IV parses only mathematically valid strings. While this is desired in most CAS circumstances, languages such as LaTeX allow for partial expressions. For example, when parsing input in real-time, the expression `&int x*(3*x+)/` would fail to parse under the current grammar rules, but the desired interpretation is an unfinished expression of the form

$$\int \frac{x\ (3x + \square)}{\square}\, \mathrm{d}\square$$

where each box represents an expected sub-expression. Graceful error handling would provide a better user experience with more feedback, especially while entering an expression.

Theoretically, the way to allow such parsing is to add the empty string, $\epsilon$, to the *primary* grammar rule. However, in practice, this would cause much ambiguity and should only be allowed when no alternative interpretation is possible. MathLex's parser is generated using a JavaScript library called Jison [33] , and adding such behavior to the grammar would require more time and research (of Jison's programming and documentation) than what was allowed for this thesis. Nonetheless, the author regards this enhancement with high priority and will likely be implemented soon.

MathLex already handles automatic insertion of matched delimiters where possible. However, the way in which this is handled could be made better: the current “fix” prepends missing opening delimiters at the *very beginning* of the stream and missing closing delimiters immediately before the next expected closing delimiter (or at the end of the stream if none are found). This is actually handled at the Tokenizer level and, to be proper, should be handled by the Parser.

### Implicit Multiplication

At present, all multiplications must be explicitly stated using the ‘`*`’ operator. In contrast, the norm in handwritten mathematics is to place variables of the same term next to each other with no symbol between. While this appears natural, it could introduce ambiguity to a computer; that is, whether $ax$ is a single variable that happens to be two characters in length or the product of two variables depends on the language specification. MathLex allows variables to have an arbitrary length (for flexibility and familiarity among programmers), so $ax$ would be understood as a single variable ‘`ax`’. So the cure is to put a space between the ‘`a`’ and the `'x'`. This is because whitespace is ignored and discarded by the Tokenizer, except to separate tokens. Thus parsing implicit multiplication would require detection of adjacent “factors” in the token stream with no separator or operator between. A grammar rule for this might look like the following:

$\langle \textit{implicit multiplication} \rangle ::= \langle \textit{factor} \rangle \; \langle \textit{factor} \rangle$

Unfortunately, the above grammar introduces a new problem: it treats an adjacent parenthesized expression as a factor, which creates ambiguity with function application. For example, is $(f+g)(x+y)$ a function application on a builder meaning $f(x+y)+g(x+y)$, or is it the factorized multiplication of $f \cdot x + f \cdot y + g \cdot x + g \cdot y$? Determining meaning requires extra type information about $f$ and $g$. (See Type-Checking below.) To a mathematician, the variables

$f$ and $g$ are commonly used to represent functions, and thus the first interpretation seems more natural. However, to a parser, $f$ and $g$, could represent anything.

The present thinking is to treat implicit multiplication and function application as a single "application" operator in the syntax tree: $a\,b$ would be the "application of $a$ and $b$". An application's meaning will be determined later by the type-checker according to Table V.1, which outlines all possible type relationships between the LHS and RHS of an application operator.

**Table V.1** Application Operator Interpretation: $\times$ = multiplication, $f$ = function application

| | | **RHS Type** | | | |
|---|---|---|---|---|---|
| | | PARENTHESIS | VARIABLE | NUMBER | FUNCTION |
| **LHS Type** | PARENTHESIS | $f$ | $\times$ | $\times$ | $\times$ |
| | VARIABLE | $f$ | $\times$ | $\times$ | $\times$ |
| | NUMBER | $\times$ | $\times$ | $\times$ | $\times$ |
| | FUNCTION | $f$ | $f$ | $f$ | $f$ |

Based on the patterns in Table V.1, three tests can determine the interpretation of an application as described in Algorithm 1

**Algorithm 1** Application Operator Interpretation

**if** LHS = FUNCTION **then return** function application
**else if** LHS = NUMBER **then return** implicit multiplication
**else if** RHS = PARENTHESIS **then return** function application
**else return** implicit multiplication
**end if**

This interpretation is not perfect because for example it would misinterpret `(x+y)(y+z)` and `x(x+2)` as function applications unless the type of `x` has been previously determined.

Also note that Table V.1 implies the addition of a new *Function*-type token. At present, the number sign (`#`), the ampersand (`&`), and the colon(`:`) are used to decorate constants, operators, and delimiters (respectively) to distinguish them from alternate meanings. The addition of a "function decorator" would inform the Tokenizer and ultimately the Type-Checker that the current variable identifier is to be treated as a function. The currently unused keyboard characters are: Dollar Sign (`$`), Back-tick (`` ` ``), Double-Quote (`"`), and Question Mark (`?`). Any of these symbols would make a good decorator, and the least intrusive of these in the author's opinion is a dollar-sign prefix (type `$f` to represent the function $f$). This is similar to how PHP treats variables, e.g. `$var`, and the "address-of" operator (`&`) used by C, C++, and Ruby to refer to function blocks). However, by introducing additional unfamiliar syntax, such a decorator might oppose the aim of this thesis to create a *natural* math input language. Since it may only be possible to identify the identifier's type *after* creating the AST, the identification of an application operator as a multiplication or function application will likely require Type-Checking and/or Third-Pass Parsing. (See below.)

Other delimiters can also lead to ambiguity with implicit multiplication. For example, without multiplication signs, the expression $|x+2|y+3|z|$ could be interpreted as `|x+2|*y+3*|z|` or as `|x+2*|y+3|*z|`. We already have a solution to this ambiguity with matched delimiters: the former would be entered into MathLex as `|:x+2:| y+3 |:z:|`, while the latter would be entered as `|:x + 2|:y+3:|z:|`.

### Type-Checking

As briefly mentioned in the section on implicit multiplication, a type-checking system would allow the same operator to have different meanings in different contexts. A great example is the $\times$ symbol: between scalars, it means multiplication; between vectors, this is a cross product; and between sets, it becomes a Cartesian product.

Furthermore, a type-checker would ensure mathematical validity. For example, a dot product operates on two vectors and returns a scalar. At present, MathLex would allow a dot product between a set and a scalar, neither of which are vectors.

Type-checking is easy in statically typed languages since the type of every variable is known. However, MathLex is *dynamically typed* since variables could represent anything. As mentioned in the implicit multiplication section, adding decorators to specify type would aid in type-checking, but would introduce unnatural syntax.

Another approach is to use *type hinting*, or finding a type assignment that satisfies the operator constraints. For example, addition works only for scalars, vectors, vector spaces, etc., and only then when both operands are of the same type. Similarly, the cross product only works on vectors, but the Cartesian product only works on sets. So the variables contained in the expression `a &x b + c` must either all be vectors or all be vector spaces. By themselves, `a`, `b`, and `c` could be anything, but when combined (by precedence) under the $\times$ and $+$ operators, their types may be determined by the operator definitions.

One problem associated with type hinting is uncertainty when multiple type assignments would make sense. The only way to deal with such ambiguity is to make the type domain of each variable consistent by eliminating the types that are incompatible with other variables' domains. For example, the division operation makes sense for scalars and vectors (even then only if the vector is in the numerator). Therefore, the expression `a / b` has the following valid type assignments:

$$\texttt{a} : \{\text{NUMBER},\text{VECTOR}\} \qquad\qquad \texttt{b} : \{\text{NUMBER}\}$$

**Third-Pass Parsing**

MathLex employs two levels of parsing to construct an AST: The tokenizer operates on a linear stream of characters and, adding semantic meaning, groups them into a linear stream of tokens based on predefined patterns. The Parser then operates on this linear stream of tokens and groups them into a tree based on context. However, just as operator tokens can be represented by multiple strings, so too can mathematical concepts be represented by multiple contexts. For example, gradient, divergence, and curl can be represented by functions or prefix operators, and each representation results in a different substructure in the AST: a function node in which the LHS is a "builder" consisting of a single identifier (`grad`, `div`, or `curl`) or a prefix node in which the name of the node is the operation itself. The latter of these representations is preferred since it is easier to build a renderer/translator for such a structure.

For ideal uniformity, the final AST should have only one way to represent each supported mathematical concept. In the previous example, gradient, divergence, and curl should each have only one representation in the syntax tree instead of different structures that depend on syntax. Unfortunately, the parser has no way of jumping states from "looking for a function" to "just found a gradient operation" internally: the parser operates on token types and not on their values.

A more subtle concern is the representation of associative operations (such as addition, multiplication, and union) in the AST. The parser currently treats such operations as left-associative. While there is nothing wrong with this representation, a more mathematically correct representation would be to group chained associative operations under a single node with an arbitrary number of parameters.

The solution to these issues is another (third) layer of parsing that operates on a rudimentary AST and produces a more refined AST by matching and replacing certain substructures with better alternatives.

### Additional Symbols and Alternate Notation

The title of this section speaks for itself: many desirable mathematical operations and concepts are still lacking or have a somewhat unintuitive syntax. For instance,

- Matrix display: $\begin{bmatrix} a & b \\ c & d \end{bmatrix}$
- Function convolution: $(f * g)(t) = \displaystyle\int_0^t f(v)\, g(t-v)\, \mathrm{d}t)$

- Inner Products: $\langle \vec{x}, \vec{y} \rangle$
- Geometric Constructs and Units: $\angle ABC$, $\overline{PQ}$, $\triangle XYZ$, $45°$

In general, more written-style input can and will be added, and the need for ampersands on some operations will be reduced. Alternate intuitive notations for currently supported operations will be added as they are brainstormed or suggested by others.

### Graphical and Handwritten Input

Keyboard entry in MathLex is doable on mobile devices, but still tedious. The demonstration page at the companion website (`http://ugrthesis.mathlex.org`) has graphical palettes to insert symbols and templates more quickly. These are not yet provided in the MathLex JavaScript plugin but should be soon. Even so, the palettes are too big to fit comfortably on mobile displays. The ideal method for mobile entry would be handwriting recognition. This may not be implemented for a very long time, but such an interface should be the desired goal for now. This may change as mobile devices, browsers, and Web technologies continue to mature.

# REFERENCES


[1] D. E. Knuth, *The TEXbook*. Addison Wesley, 13th ed., December 1987.

[2] L. Lamport, *LATEX: A Document Preparation System: User's Guide and Reference Manual*. Addison Wesley, 2nd ed., November 1994.

[3] D. Cervone, "MathJax." `http://www.mathjax.org`, November 2012.

[4] Maplesoft, "Maple." `http://www.maplesoft.com/products/maple/`.

[5] Wolfram Research, "Mathematica." `http://www.wolfram.com/mathematica/`.

[6] Texas Instruments, "Derive." `http://education.ti.com/en-GB/uk/products/computer-software/derive-6/features/features-summary`.

[7] MathWorks, "Matlab." `http://www.mathworks.com/products/matlab/`.

[8] Sagemath, "Sage." `http://sagemath.org/`.

[9] "PocketCAS for iOS." `http://pocketcas.com`.

[10] S. Buswell, S. Devitt, A. Diaz, N. Poppelier, B. Smith, N. Soiffer, R. Sutor, and S. Watt, "Mathematical Markup Language (MathML) 1.0 Specification." `http://www.w3.org/TR/1998/REC-MathML-19980407/`, April 1998.

[11] Pearson, "MyMathLab." `http://www.mymathlab.com/`.

[12] John Wiley & Sons, Inc., "WileyPLUS." `https://www.wileyplus.com/WileyCDA/`.

[13] Mathematical Association of America, "WeBWorK." `http://webwork.maa.org/`.

[14] N. Carolina State Univ., "WebAssign: Online Homework and Grading." `http://www.webassign.net/`.

[15] Maplesoft, "Maple T.A.." `http://www.maplesoft.com/products/mapleta/`.

[16] Wolfram Research, "About Wolfram|Alpha." `http://www.wolframalpha.com/about.html`.

[17] The OpenMath Society, "OpenMath." `http://www.openmath.org`.

[18] MYMathApps, "Maplets for Calculus." `http://www.mymathapps.com/`.

[19] Sagemath, "Sage Cell Server (Aleph)." `http://aleph.sagemath.org/`.

[20] Mozilla Developer Network, "Learn HTML." `https://developer.mozilla.org/en-US/learn/html`.

[21] TutsPlus, "30 Days to Learn HTML & CSS." `http://learncss.tutsplus.com/`.

[22] W3Schools, "Learn HTML." `http://www.w3schools.com/html/`.

[23] The jQuery Foundation, "jQuery." `http://jquery.com`.

[24] "MooTools: A Compact JavaScript Framework." `http://mootools.net`.

[25] A. Dupont, "Prototype JavaScript Framework." `http://prototypejs.org`.

[26] Yahoo, "YUI Library." http://yuilibrary.com.

[27] The Dojo Foundation, "The Dojo Toolkit." http://dojotoolkit.org.

[28] Mozilla Developer Network, "Reserved Words - JavaScript Reference." https://developer.mozilla.org/en-US/docs/JavaScript/Reference/Reserved_Words.

[29] "Information technology - Syntactic metalanguage - Extended BNF," No. ISO 14977, ISO, Geneva, Switzerland, 1996.

[30] J. W. Backus, "The Syntax and Semantics of the Proposed International Algebraic Language of the Zurich ACM-GAMM Conference," in *Proceedings of the Intl. Conf. on Info. Processing, UNESCO*, pp. 125–132, 1959.

[31] N. Chomsky, "On certain formal properties of grammars," *Information and Control*, vol. 2, pp. 137–67, June 1959.

[32] J. Ashkenas, "CoffeeScript." http://coffeescript.org.

[33] Z. Carter, "Jison." http://zaach.github.com/jison/.