# Package mathfont v. 3.0a Implementation

Conrad Kosowsky

February 2026

kosowsky.latex@gmail.com

For easy, off-the-shelf use, type the following in your preamble and compile with XeLaTeX or LuaLaTeX:

\usepackage[⟨*font name*⟩]{mathfont}

As of version 2.0, using LuaLaTeX is recommended. Minor backwards incompatible changes in version 3.0.

**Overview**

The mathfont package adapts Unicode text fonts for math mode. The package allows the user to specify a default font for different classes of math symbols, and it enables Unicode input in math mode. The package provides tools to change the font locally for math alphabet characters. When typesetting with LuaTeX, mathfont adds resizable delimiters, big operators, and a MathConstants table to text fonts.

---

This file documents the code for the mathfont package. It is not a user guide! If you are looking for instructions on how to use mathfont in your document, see `mathfont-user-guide.pdf`, which is included with the mathfont installation and is available on CTAN. See also the other `pdf` documentation files for mathfont. Section 1 of this document begins with the implementation basics, including package declaration and package options. Section 2 provides package default settings, and section 3 deals with errors and messaging. Section 4 contains the fontloader, and section 5 contains the optional-argument parser for `\mathfont`. Section 6 documents the code for the `\mathfont` command itself. Section 7 contains the code for local font changes. Section 8 contains miscellaneous material. Sections 9–11 contain the Lua code to modify font objects at loading, and section 12 lists the Unicode hex values used in symbol declaration. Version history and code index appear at the end of the document.

Acknowledgements: Thanks to Lyric Bingham for her work checking my Unicode hex values. Thanks to Matthew Braham, Sergio Callegari, Daniel Flipo, Nikos Platis, Shyam Sundar, Adrian Vollmer, Herbert Voss, and Andreas Zidak for pointing out bugs in previous versions of mathfont. Thanks to Jean-François Burnol for pointing out an error in the documentation in reference to his mathastext package.

At high level, the package works as follows: the font-loader `\M@newfont` is a wrapper around NFSS macros to declare fonts—namely `\DeclareFontFamily` and `\DeclareFontShape`—or, if the user requested to use fontspec as a backend, the macro `\fontspec_set_family:Nnn`. All font-setting macros in the package will call `\M@newfont`. The primary font-setting command `\mathfont` is a wrapper around `\DeclareSymbolFont` and calls various `\M@`⟨*keyword*⟩`@set` commands. Each `\M@`⟨*keyword*⟩`@set` macro is a wrapper around a number of `\Umathcode` declarations that do the actual work of setting default font(s). The local font-change commands are wrappers around `\DeclareMathAlphabet`, and the Lua font adjustments alter the font table through the `luaotfload.patch_font` callback when TEX loads the font. Specifically, we change the top-level flag `nomath` to false, alter character-level entries in the table to make the font more suitable for math typesetting, and add a MathConstants table based on font dimensions.

# 1 Setup

First, the package should declare itself. The first 61 lines of `mathfont.sty` are comments.

```
\NeedsTeXFormat{LaTeX2e}
\ProvidesPackage{mathfont}[2026/02/07 v. 3.0a]
```

Informational message.

```
\def\@mathfontinfo#1{\wlog{Package mathfont Info: #1}}
```

We specify conditionals and one count variable that we use later in handling options and setup.

```
\newif\ifM@XeTeXLuaTeX     % is engine one of xetex or luatex?
\newif\ifM@Noluaotfload    % cannot find luaotfload.sty?
\newif\ifM@adjust@font     % should adjust fonts with lua script?
\newif\ifM@font@loaded     % load mathfont with font specified?
\newif\ifE@sterEggDecl@red % already did easter egg?
\newcount\M@loader         % specifies which font-loader to use
```

We disable the twenty user-level commands. If mathfont runs normally, it will overwrite these “bad” definitions later, but if it throws one of its two fatal errors, it will `\endinput` while the user-level commands are error messages. That way the commands don’t do anything in the user’s document, and the user gets information on why not. The bad definitions gobble their original arguments to avoid a “missing `\begin{document}`” error. To streamline the process, we metacode most of the error messages, namely the macros that `\@gobble` their argument and the macros that `\@gobbletwo` their argument.

```
\long\def\@gobble@brackets[#1]{}
\def\M@NoMathfontError#1{\PackageError{mathfont}
  {\MessageBreak Invalid command\MessageBreak
  \string#1 on line \the\inputlineno}
  {Your command was ignored. I couldn't\MessageBreak
  load mathfont, so I never defined this\MessageBreak
  control sequence.}}
```

\M@NoMathfontError

The macro `\M@robust@def` is an engine-dependent command to define robust control sequences. We want this part of the package to work regardless of the engine, so we need an approach that doesn't depend on $\epsilon$-TEX support. However, I don't want to use `\DeclareRobustCommand` with XETEX or LuaTEX because that will leave useless macros like `\mathfont␣` defined when we fill in the proper definitions of the user-level commands later.

```
\ifx\protected\@undefined
  \let\M@robust@def\DeclareRobustCommand
\else
  \def\M@robust@def{\protected\def}
\fi
```

First the commands that normally accept a single argument—the "bad" versions `\@gobble` the argument. To keep the syntax straightforward, we expand the definition using `\edef`. (We can't use `\expanded` because that's XETEX/LuaTEX only.) We need to do this because the macro name is stored in `\@i`, and otherwise, we would end up with a mess of `\expandafter`s to expand all instances of `\@i`.

```
\@tfor\@i:=\setfont
  \RuleThicknessFactor
  \IntegralItalicFactor
  \SurdVerticalFactor
  \SurdHorizontalFactor
  \charmline
  \charmfile
  \CharmLine
  \CharmFile
  \CharmInfo
  \CharmType\do{%
    \edef\@tempa{\noexpand\M@robust@def\expandafter\noexpand\@i{%
      \noexpand\M@NoMathfontError\expandafter\noexpand\@i
      \noexpand\@gobble}}
    \@tempa}
```

Now for the macros that `\@gobbletwo` their argument. The code is essentially the same.

```
\@tfor\@i:=\newmathrm
  \newmathit
  \newmathbf
  \newmathbfit
  \newmathsc
  \newmathscit
  \newmathbfsc
  \newmathbfscit\do{%
    \edef\@tempa{\noexpand\M@robust@def\expandafter\noexpand\@i{%
      \noexpand\M@NoMathfontError\expandafter\noexpand\@i
      \noexpand\@gobbletwo}}
    \@tempa}
```

The two commands with weird "arguments": `\charminfo` and `\charmtype` scan and remove the next integer, so we assign a count value instead of gobbling stuff.

```
\M@robust@def\charminfo{\M@NoMathfontError\charminfo
  \begingroup
    \afterassignment\endgroup
    \count@}
\M@robust@def\charmtype{\M@NoMathfontError\charmtype
  \begingroup
    \afterassignment\endgroup
    \count@}
```

For the optional argument, we check if the following character is a `[`. If yes, we gobble first the brackets and then the mandatory argument. If not, we gobble the single mandatory argument.

```
\@tfor\@i:=\documentfont
  \mathfont
  \mainfont
  \mathfontshapes
  \mathconstantsfont\do{%
    \edef\@tempa{\noexpand\M@robust@def\expandafter\noexpand\@i{%
      \noexpand\M@NoMathfontError\expandafter\noexpand\@i
      \noexpand\@ifnextchar[%
        {\noexpand\expandafter\noexpand\@gobble
          \noexpand\@gobble@brackets}
        {\noexpand\@gobble}}}
    \@tempa}
```

We code `\newmathfontcommand` by hand because it is the only command with four arguments.

```
\M@robust@def\newmathfontcommand{%
```

```
  \M@NoMathfontError\newmathfontcommand\@gobblefour}
```

Check that the engine is XeTeX or LuaTeX. If yes, set `\ifM@XeTeXLuaTeX` to true. (Otherwise the conditional will be false by default.)

```
\ifx\directlua\@undefined
\else
  \M@XeTeXLuaTeXtrue
\fi
\ifx\XeTeXrevision\@undefined
\else
  \M@XeTeXLuaTeXtrue
\fi
```

The package can raise two fatal errors: one if the engine is not XeTeX or LuaTeX (and cannot load OpenType fonts) and one if TeX cannot find the luaotfload package. In both cases, the package will stop loading, so we want a particularly conspicuous error message. For each message, we check the appropriate conditional to determine if we need to raise the error. If yes, we change space to catcode 12 inside a group. We define a `\GenericError` inside a macro and then call the macro for a cleaner error context line. The `\@gobbletwo` eats the extra period and return that LaTeX adds to the error message. Notice that we expand the error before the `\endgroup`—this is because we need to switch `\M@XeTeXLuaTeXError` with its replacement text while it is still defined before we leave the group. At the same time, we want `\AtBeginDocument` and `\endinput` outside the group. The second `\expandafter` means that we expand the final `\fi` before `\endinput`, which balances the original conditional.

```
\ifM@XeTeXLuaTeX\else
\begingroup
\catcode`\ =12\relax
\def\M@XeTeXLuaTeXError{\GenericError{}%
{\MessageBreak\MessageBreak
Package mathfont error:%
\MessageBreak\MessageBreak
**********************\MessageBreak
*                     *\MessageBreak
*       UNABLE TO     *\MessageBreak
*     LOAD MATHFONT   *\MessageBreak
*                     *\MessageBreak
*     Missing XeTeX   *\MessageBreak
*       or LuaTeX     *\MessageBreak
*                     *\MessageBreak
**********************\MessageBreak\@gobbletwo}%
{See the mathfont package documentation for explanation.}%
```

\M@XeTeXLuaTeXErro

```
{I need XeTeX or LuaTeX to use mathfont. It\MessageBreak
looks like the current engine is something\MessageBreak
else, so I'm going to stop reading in the\MessageBreak
package file now. (You won't be able to use\MessageBreak
commands from mathfont in your document.) To\MessageBreak
load mathfont correctly, please retypeset your\MessageBreak
document with one of those two engines.^^J}}%
\expandafter\endgroup
\M@XeTeXLuaTeXError
  \AtEndOfPackage{%
    \typeout{:: mathfont :: Failed to load\on@line.}}
  \expandafter\endinput % we \endinput with a balanced conditional
\fi
```

Now do the same thing in checking for luaotfload. If the engine is LuaTEX, we tell mathfont to implement Lua-based font adjustments by default. The conditional `\ifM@Noluaotfload` will keep track of whether TEX could find `luaotfload.sty`. If the engine is XETEX, issue a warning.

```
\ifdefined\directlua
  \M@adjust@fonttrue % if engine is LuaTeX, adjust font by default
  \IfFileExists{luaotfload.sty}
    {\M@Noluaotfloadfalse\RequirePackage{luaotfload}}
    {\M@Noluaotfloadtrue}
\else
  \AtEndOfPackage{\PackageWarningNoLine{mathfont}{%
    The current engine is XeTeX, but as\MessageBreak
    of mathfont version 2.0, LuaTeX is\MessageBreak
    recommended. Consider compiling with\MessageBreak
    LuaLaTeX. Certain features will not\MessageBreak
    work with XeTeX}}
\fi
```

If the engine is LuaTEX, we must have luaotfload because LuaTEX needs this package to load OpenType fonts. Before anything else, TEX should check whether it can find `luaotfload.sty` and stop reading in mathfont if it cannot. Same command structure as before. Newer LATEX versions load luaotfload as part of the format, but it never hurts to double check.

```
\ifM@Noluaotfload % true if LuaTeX AND no luaotfload.sty
\begingroup
\catcode`\ =12\relax
\def\M@NoluaotfloadError{\GenericError{}%
{\MessageBreak\MessageBreak
Package mathfont error:%
```

\M@NoluaotfloadErr

```
\MessageBreak\MessageBreak
**************************\MessageBreak
*                        *\MessageBreak
*       UNABLE TO        *\MessageBreak
*     LOAD MATHFONT      *\MessageBreak
*                        *\MessageBreak
*    Cannot find the     *\MessageBreak
*  file luaotfload.sty   *\MessageBreak
*                        *\MessageBreak
**************************\MessageBreak\@gobbletwo}%
{You are likely seeing this message because you haven't^^J%
installed luaotfload. Check your TeX distribution for a^^J%
list of the packages on your system.^^J^^J%
See the mathfont documentation for further explanation.}%
{You're in trouble here. It looks like the current\MessageBreak
engine is LuaTeX, so I need the luaotfload package\MessageBreak
to make mathfont work correctly. However, I can't\MessageBreak
find luaotfload, which likely means something is\MessageBreak
wrong with your TeX installation. I'm going to stop\MessageBreak
reading in the mathfont package file. (You won't be\MessageBreak
able to use commands from mathfont in your document.)\MessageBreak
To load mathfont properly, make sure you installed\MessageBreak
luaotfload.sty in a directory searchable by TeX or\MessageBreak
compile with XeLaTeX.^^J}}%
\expandafter\endgroup
\M@NoluaotfloadError
  \AtEndOfPackage{%
    \typeout{:: mathfont :: Failed to load\on@line.}}
  \expandafter\endinput % we \endinput with a balanced conditional
\fi
```

Easter egg!!

```
\DeclareOption{easter-egg}{%
  \ifE@sterEggDecl@red\else
    \E@sterEggDecl@redtrue
    \newcount\@easter@egg@
    \protected\def\EasterEggUpdate{%
      \ProcessE@sterEgg\showtokens\expandafter{\E@sterEggUpd@te}}
    \let\ProcessE@sterEgg\relax
```

Two status updates during package loading.

```
    \edef\E@sterEggUpd@te{Easter Egg Status:^^J^^J%
        Okay, opening your Easter egg.^^J%
```

```
228       Type \string\EasterEggUpdate\space in your^^J%
229       document to see the status.^^J^^J}
230     \EasterEggUpdate
231   \def\E@sterEggUpd@te{Easter Egg Status:^^J^^J%
232       Uh oh. It looks like your Easter^^J%
233       egg flew out the window. I don't^^J%
234       I don't suppose you know the best^^J%
235       kind of bait to lure an egg?^^J^^J}
236     \EasterEggUpdate
```

Possible updates if the user types `\EasterEggUpdate`. We define the status update with `\ProcessE@sterEgg`, which stores the current message in `\E@sterEggUpd@te` and changes the message as the user calls `\EasterEggUpdate`. The count `\@easter@egg@` keeps track of how many times the user has requested a status update.

```
237   \def\ProcessE@sterEgg{%
238     \edef\E@sterEggUpd@te{Easter Egg Status:^^J^^J%
239       \ifodd\@easter@egg@
240         \ifcase\numexpr(\@easter@egg@ - 1) / 2\relax
241           An Easter bunny must be related to a^^J%
242           platypus, no? Some sort of monotreme...%
243         \or
244           Don't count your chickens before they hatch^^J%
245           out of Easter eggs! But we don't have any^^J%
246           chickens right now because there are no eggs,^^J%
247           and the supply chain is sad.%
248         \or
249           Sorry, I'm late to a meeting. Can't talk right now.%
250         \or
251           Sunday, Monday, Tuesday, Wednesday, also^^J%
252           known as hump day, as in camel humps, which^^J%
253           I must say look distinctly egg-like if you^^J%
254           squint.%
255         \or
256           I'm calling Eggs Anonymous!%
257         \or
258           Sorry, I'm on the phone. Can't talk right now.%
259         \or
260           Still haven't found your Easter egg. I know^^J%
261           it's floating around here somewhere. Like an^^J%
262           asteroid in space, hopefully without the^^J%
263           massive extinction event.%
264         \or
```

```
265           Did you know eggs are used to make certain^^J%
266           types of vaccines? PSA: get your flu shot^^J%
267           and your covid shot!%
268         \or
269           Three large eggs.^^J%
270           Three large eggs.^^J%
271           See how they crack.^^J%
272           See how they crack.^^J%
273           Their broken shells are so pearly white.^^J%
274           In simmering water they catch the light.^^J%
275           Did you ever see such a sight in your life^^J%
276           As three poached eggs?%
277         \or
278           Do gnus eat eggs? Surely they must.%
279         \or
280           Okay, I have a fishing rod, some twine, and^^J%
281           a hook, but I still haven't caught your Easter^^J%
282           egg. Apparently it's harder to catch an egg^^J%
283           than a fish.%
284         \or
285           Sorry, I'm out fishing. Can't talk right now.%
286         \or
287           Is ghoti really an acceptable phonetic^^J%
288           spelling of fish? I am skeptical.%
289         \or
290           Perhaps an Easter bunny is actually a species^^J%
291           of fish. A rabbit fish.%
292         \else
293           Sorry, I'm all out of witty things to say.^^J%
294           Check back later.%
295         \fi
296       \else
297         Still wrangling. Check back later.%
298       \fi^^J^^J}%
299     \global\advance\@easter@egg@\@ne}
```

One status update `\AtBeginDocument`.

```
300   \AtBeginDocument{\bgroup
301     \let\ProcessE@sterEgg\relax
302     \def\E@sterEggUpd@te{Easter Egg Status:^^J^^J%
303       If we have zero eggs^^J%
304       and zero bunnies, how^^J%
305       many gnats does it take^^J%
```

```
306         to change a lightbulb??^^J^^J}
307       \EasterEggUpdate
308       \egroup}
```

One update at the first instance of math mode, assuming another package doesn't overwrite the contents of `\everymath` first.

```
309     \def\math@E@sterEggUpd@te{\begingroup
310       \let\ProcessE@sterEgg\relax
311       \def\E@sterEggUpd@te{Easter Egg Status:^^J^^J%
312         Scrambled, poached, or sunny side up?^^J^^J}%
313       \EasterEggUpdate
314       \endgroup
315       \global\let\math@E@sterEggUpd@te\relax}
316     \everymath\expandafter{\the\everymath\math@E@sterEggUpd@te}
```

Two status updates `\AtEndDocument`, including the egg itself. First, we disable `\ProcessE@sterEgg` since we don't need it anymore. Then inside a group, we make the control symbols `\*`, `\/`, and `\=` expand to their own names and do some extreme catcode sports. We convert `+` to active and make it expand to a space. Because everything has already been tokenized inside `\DeclareOption`, we have to retokenize the definition of `+` inside `\scantokens`, and we set `\everyeof` to `\noexpand` to avoid an end-of-file error.

```
317     \AtEndDocument{\let\ProcessE@sterEgg\relax
318       \begingroup
319         \edef\*{\@backslashchar*}
320         \edef\/{\@backslashchar/}
321         \edef\={\@backslashchar=}
322         \catcode`\+=\active
323         \everyeof{\noexpand}
324         \scantokens{\def+{ }}
```

At this point we are ready to make the egg message Again, we have to retokenize everything with `\scantokens` because it was previously tokenized. However, if we write `^^J` directly inside `\scantokens`, that primitive will convert the newline to a blank space, so instead we store `^^J` in `\@tempb`. After the `\edef` expands `\scantokens`, it also expands each `\@tempb`, so `\@tempa` has the line breaks we want.

```
325         \def\@tempb{^^J}
326         \edef\@tempa{\scantokens{Easter Egg Status:\@tempb\@tempb
327         The egg has been retrieved. What\@tempb
328         pinnacle of pulchritude!\@tempb\@tempb
329         ++++++++++++++++++\@tempb
330         ++++++++++++++++++++\@tempb
```

```
        ++++++++++++++++++++++++\@tempb
        +++++----------------------\@tempb
        ++++++++++++++++++++++++++\@tempb
        ++++++/\****/\****/\****/\****\@tempb
        ++++++\/****\/****\/****\/*****\@tempb
        ++++++++++++++++++++++++++++\@tempb
        ********************************\@tempb
        ***/\****/\****/\****/\****/\***\@tempb
        ***\/****\/****\/****\/****\/***\@tempb
        ********************************\@tempb
        ********************************\@tempb
        ******/\****/\****/\****/\******\@tempb
        ++++++\/****\/****\/****\/*****\@tempb
        ++++++++++++++++++++++++++++\@tempb
        +++--------------------------\@tempb
        ++++++++++++++++++++++++\@tempb
        ++++++++++++++++++++++++\@tempb
        +++++++++\============/\@tempb
        ++++++++++\========/\@tempb
        ++++++++++\======/\@tempb
        ++++++++++++|====|\@tempb
        ++++++++++++/======\@backslashchar\@tempb
        ++++++++++(________)\@tempb}}
```

Then end the group and store the message in `\E@sterEggUpd@te`.

```
      \expandafter\endgroup\expandafter
      \def\expandafter\E@sterEggUpd@te\expandafter{\@tempa}
      \EasterEggUpdate
      \def\E@sterEggUpd@te{Easter Egg Status:^^J^^J%
        Happy, happy day! Happy,^^J%
        happy day! Clap your hands,^^J%
        and be glad your hovercraft^^J%
        isn't full of eels!^^J^^J}
      \EasterEggUpdate
      \let\E@sterEggUpd@te\relax
      \let\EasterEggUpdate\relax}
  \fi}% my easter egg :)
```

The five real package options. The `default-loader` and `fontspec-loader` tell mathfont what to use as a backend for loading fonts.

```
\DeclareOption{default-loader}{\M@loader\z@}
\DeclareOption{fontspec-loader}{\M@loader\@ne}
```

The options `adjust` and `no-adjust` determine whether mathfont applies Lua-based font adjustments to fonts loaded in the future.

```
\DeclareOption{adjust}{\M@adjust@fonttrue}
\DeclareOption{no-adjust}{\M@adjust@fontfalse}
```

Interpret an unknown option as a font name and save it for loading. In this case, the package sets `\ifM@font@loaded` to true and stores the font name in `\M@font@load`.

```
\DeclareOption*{\M@font@loadedtrue
  \edef\M@font@load{\CurrentOption}}
\ProcessOptions*
```

For the font-loader, we have a bit of processing to do. First print an informational message in the `log` file. The default loader is easy, but if the user requests fontspec, we have to make sure to load everything properly.

```
\ifcase\M@loader
  \@mathfontinfo{Default font-loader was
    requested for font loading.}
\or
  \@mathfontinfo{Package fontspec was
    requested for font loading.}
```

If fontspec was already loaded, check whether `\g__fontspec_math_bool` is true or not. If it is, change it to false.

```
  \@ifpackageloaded{fontspec}
    {\@mathfontinfo{Package fontspec detected.}
      \csname bool_if:NTF\expandafter\endcsname
        \csname g__fontspec_math_bool\endcsname
        {\@mathfontinfo{Setting
          \string\g__fontspec_math_bool to false.}
          \csname bool_set_false:N\expandafter\endcsname
          \csname g__fontspec_math_bool\endcsname}{\relax}}
```

If fontspec was not loaded, check that the package file exists.

```
    {\@mathfontinfo{Package fontspec not detected.}
      \IfFileExists{fontspec.sty}
        {\@mathfontinfo{File fontspec.sty was found.}
          \@mathfontinfo{Loading fontspec.}
          \RequirePackage[no-math]{fontspec}}
        {\PackageError{mathfont}
          {Missing package fontspec;^^J%
          using default font-loader instead}
          {You requested fontspec as the font-loader\MessageBreak
          for mathfont. However, I can't find the\MessageBreak
```

```
397           package file for fontspec, so I'm going to\MessageBreak
398           use mathfont's built-in font-loader. (This\MessageBreak
399           likely means that something is wrong with\MessageBreak
400           your TeX installation.) Check your TeX\MessageBreak
401           distribution for a list of the packages\MessageBreak
402           installed on your system. To resolve this\MessageBreak
403           error, make sure fontspec is installed in\MessageBreak
404           a directory searchable by TeX or load\MessageBreak
405           mathfont with the default-loader option.^^J}
406           \M@loader\z@}}
407 \fi
```

We print an informational message specifying the font-loader in use. We store default OpenType features in `\M@otf@features`. The contents depend on the font-loader because we use XeTeX/luaotfload syntax versus fontspec syntax. By default, mathfont loads fonts with Latin script, default language, TeX and common ligatures, and lining numbers.

```
408 \ifcase\M@loader
409   \@mathfontinfo{Using default font-loader.}
410   \AtEndOfPackage{%
411     \typeout{:: mathfont :: Using default font-loader.}}
\M@otf@features 412   \def\M@otf@features{script=latn;language=dflt;+tlig;+liga;+lnum}
413 \or
414   \@mathfontinfo{Using fontspec as font-loader.}
415   \AtEndOfPackage{%
416     \typeout{:: mathfont :: Using fontspec as font-loader.}}
\M@otf@features 417   \def\M@otf@features{Script=Latin,%
418     Language=Default,%
419     Ligatures={TeX,Common},%
420     Numbers=Lining}
421 \fi
```

We print an informational message depending on whether the user enabled Lua-based font adjustments. If `\directlua` is defined, that means we are using LuaTeX, so we print a message depending on `\ifM@adjust@font`.

```
422 \ifdefined\directlua
423   \ifM@adjust@font
424     \@mathfontinfo{Enabling Lua-based font adjustments.}
425     \AtEndOfPackage{%
426       \typeout{:: mathfont :: Lua-based font adjustments
427         enabled.}}
428   \else
429     \@mathfontinfo{Disabling Lua-based font adjustments.}
```

```
    \AtEndOfPackage{%
      \typeout{:: mathfont :: Lua-based font adjustments
        disabled.}}
  \fi
\else
```

If `\directlua` is undefined, we make sure Lua-based font adjustments are disabled, and we issue an error if the user tried to manually enable them.

```
  \ifM@adjust@font
    \PackageError{mathfont}{Option^^J"adjust" ignored with XeTeX}
    {Your package option "adjust" was ignored.\MessageBreak
    This option works only with LuaTeX, and it\MessageBreak
    looks like the current engine is XeTeX. To\MessageBreak
    enable Lua-based font adjustments, typeset\MessageBreak
    with LuaLaTeX.^^J}
    \M@adjust@fontfalse
  \fi
  \@mathfontinfo{Disabling Lua-based font adjustments.}
  \AtEndOfPackage{%
    \typeout{:: mathfont :: Lua-based font adjustments disabled.}}
\fi
```

# 2 Default Settings

We save four macros from the LaTeX kernel for safe-keeping, and then we change their definitions. As of version 3.0 of mathfont, the new definitions for `\set@mathchar`, etc. are not necessary for implementing mathfont, but we keep them in the package to make `\DeclareMathSymbol` and friends compatible with Unicode. For these three control sequences, we convert the hexadecimal digits in `\count0` and `\count2` back to decimal and change the `\math` primitive to `\Umath`.

```
\let\@@set@mathchar\set@mathchar
\let\@@set@mathsymbol\set@mathsymbol
\let\@@set@mathaccent\set@mathaccent
\let\@@DeclareSymbolFont\DeclareSymbolFont
\let\@@DeclareSymbolFont@m@dropped\DeclareSymbolFont@m@dropped
\@onlypreamble\@@set@mathchar
\@onlypreamble\@@set@mathsymbol
\@onlypreamble\@@set@mathaccent
\@onlypreamble\@@DeclareSymbolFont
\@onlypreamble\@@DeclareSymbolFont@m@dropped
```

```
\@mathfontinfo{Adapting \noexpand\set@mathchar for Unicode.}
\@mathfontinfo{Adapting \noexpand\set@mathsymbol for Unicode.}
\@mathfontinfo{Adapting \noexpand\set@mathaccent for Unicode.}
\@mathfontinfo{Increasing upper bound on
  \noexpand\DeclareSymbolFont to 256.}
```

Kernel command to set math characters from keystrokes.

```
\def\set@mathchar#1#2#3#4{%
  \multiply\count\z@ by 16\relax
  \advance\count\z@\count\tw@
  \global\Umathcode`#2=\mathchar@type#3+#1+\count\z@\relax}
```

Kernel command to set math characters from control sequences.

```
\def\set@mathsymbol#1#2#3#4{%
  \multiply\count\z@ by 16\relax
  \advance\count\z@\count\tw@
  \global\Umathchardef#2=\mathchar@type#3+#1+\count\z@\relax}
```

Kernel command to set accents.

```
\def\set@mathaccent#1#2#3#4{%
  \multiply\count\z@ by 16\relax
  \advance\count\z@\count\tw@
  \protected\xdef#2{%
    \Umathaccent\mathchar@type#3+\number#1+\the\count\z@\relax}}
```

We increase the upper bound on the number of symbol fonts to be 256. LuaTEX and XETEX allow up to 256 math families, but the LATEX kernel keeps the old upper bound of 16 symbol fonts under these two engines. We patch `\DeclareSymbolFont` to change the `\count18<15` to `\count18 <\e@mathgroup@top`, where `\e@mathgroup@top` is the number of math families, which is 256 in XETEX and LuaTEX. We get a sanitized definition with `\meaning` and `\strip@prefix`, implement the patch by expanding `\M@p@tch@decl@re`, and retokenize the whole thing. A simpler approach, such as calling `\M@p@tch@decl@re` directly on the expansion of `\DeclareSymbolFont`, won't work because of how TEX stores and expands parameter symbols inside macros.

As of November 2022, the LATEX team renamed `\DeclareSymbolFont` to `\DeclareSymbolFont@m@dropped`, and now `\DeclareSymbolFont` is a wrapper around the old version of itself. This was done for error checking purposes to remove extra `m`'s from certain NFSS family names. This means that if `\DeclareSymbolFont@m@dropped` is defined, we should patch that macro, and otherwise, we should patch `\DeclareSymbolFont`.

```
\ifx\DeclareSymbolFont@m@dropped\@undefined
  \edef\@tempa{\expandafter
```

```
    \strip@prefix\meaning\DeclareSymbolFont}
  \def\@tempb{\def\DeclareSymbolFont##1##2##3##4##5}
\else
  \edef\@tempa{\expandafter
    \strip@prefix\meaning\DeclareSymbolFont@m@dropped}
  \def\@tempb{\def\DeclareSymbolFont@m@dropped##1##2##3##4##5}
\fi
\def\M@p@tch@decl@re#1<15#2\@nil{#1<\e@mathgroup@top#2}
\edef\M@DecSymDef{\expandafter\M@p@tch@decl@re\@tempa\@nil}
```

Now `\M@DecSymDef` contains the patched text of our new `\DeclareSymbolFont`, all with catcode 12. In order to make it useable, we have to retokenize it. If this package was LuaTEX only, we could use `\scantextokens`, which is nicely behaved and does what we expect. However, to make it compatible with XHTEX, we use `\scantokens`. Unfortunately, while `\scantextokens` is straightforward, `\scantokens` is a menace. The problem is that when it expands, the primitive inserts an end-of-file token (because `\scantokens` mimics writing to a file and `\input`ing what it just wrote) after the retokenized code, and as a result, `\scantokens` can produce an end-of-file error. The trick (realized after much trial and error) is that if we scan the entire definition statement including `\def` and the macro definition, the end-of-file token doesn't end up in the macro definition, and we avoid the "file ended" error message.

```
\scantokens\expandafter{%
  \expandafter\@tempb\expandafter{\M@DecSymDef}}
```

We need to keep track of the number of times we have loaded fonts, and `\M@count` fulfills this role. The `\toks` will record a message that displays in the `log` file when the user calls `\mathfont`. The `\newread` is for Lua-based font adjustments.

```
\newbox\surdbox
\newcount\M@count
\newcount\M@num@localfonts
\newcount\rulethicknessfactor
\newcount\hsurdfactor
\newcount\vsurdfactor
\newmuskip\radicandoffset
\newread\M@Charm
\M@count\z@
\rulethicknessfactor\@m
\hsurdfactor\@m
\vsurdfactor\@m
\radicandoffset=1mu\relax
```

Necessary booleans and default math font shapes.

```
\newif\ifM@upper
\newif\ifM@lower
\newif\ifM@diacritics
\newif\ifM@greekupper
\newif\ifM@greeklower
\newif\ifM@agreekupper
\newif\ifM@agreeklower
\newif\ifM@cyrillicupper
\newif\ifM@cyrilliclower
\newif\ifM@hebrew
\newif\ifM@digits
\newif\ifM@operator
\newif\ifM@symbols
\newif\ifM@extsymbols
\newif\ifM@delimiters
\newif\ifM@radical
\newif\ifM@arrows
\newif\ifM@bigops
\newif\ifM@extbigops
\newif\ifM@bb
\newif\ifM@cal
\newif\ifM@frak
\newif\ifM@bcal
\newif\ifM@bfrak
\newif\if@optionpresent
\newif\if@suboptionpresent
\newif\ifM@arg@good
\newif\ifM@mode@
```

Default shapes.

```
\def\upperdefault{italic}          % latin upper
\def\lowerdefault{italic}          % latin lower
\def\diacriticsdefault{upright}    % diacritics
\def\greekupperdefault{upright}    % greek upper
\def\greeklowerdefault{italic}     % greek lower
\def\agreekupperdefault{upright}   % ancient greek upper
\def\agreeklowerdefault{italic}    % ancient greek lower
\def\cyrillicupperdefault{upright} % cyrillic upper
\def\cyrilliclowerdefault{italic}  % cyrillic lower
\def\hebrewdefault{upright}        % hebrew
\def\digitsdefault{upright}        % numerals
\def\operatordefault{upright*}     % operator font
```

```
\def\delimitersdefault{upright}    % delimiters
\def\radicaldefault{upright}       % surd
\def\bigopsdefault{upright}        % big operators
\def\extbigopsdefault{upright}     % extended big operators
\def\symbolsdefault{upright}       % basic symbols
\def\extsymbolsdefault{upright}    % extended symbols
\def\arrowsdefault{upright}        % arrows
\def\bbdefault{upright}            % blackboard bold
\def\caldefault{upright}           % caligraphic
\def\frakdefault{upright}          % fraktur
\def\bcaldefault{upright}          % bold caligraphic
\def\bfrakdefault{upright}         % bold fraktur
```

The `\M@keys` list stores all the possible keyword options, and `\M@defaultkeys` stores the character classes that `\mathfont` acts on by default.

`\M@keys`
```
\def\M@keys{upper,lower,diacritics,greekupper,%
  greeklower,agreekupper,agreeklower,cyrillicupper,%
  cyrilliclower,hebrew,digits,operator,delimiters,%
  radical,bigops,extbigops,symbols,extsymbols,arrows,%
  bb,cal,frak,bcal,bfrak}
```

`\M@defaultkeys`
```
\def\M@defaultkeys{upper,lower,diacritics,greekupper,%
  greeklower,digits,operator,symbols}
```

If the user enabled Lua-based font adjustments, the `\M@defaultkeys` list also includes delimiters, surd, and big operator symbols.

```
\ifM@adjust@font
```

`\M@defaultkeys`
```
  \edef\M@defaultkeys{\M@defaultkeys,delimiters,radical,bigops}
\fi
```

A few macros that we use for assembling lists of font information and printing messages `\AtBeginDocument`.

```
\let\M@localfonts\@empty
\let\M@symbolfonts\@empty
```

`\M@families`
```
\let\M@families\@empty
```

And now the macros to add to those three control sequences. First is a helper macro that accepts two arguments and stores information about declaration of local font changes. The macro successively adds comma-separated pairs of control sequence and font name information to `\M@localfonts`. It also keeps track of the number of distinct font names in `\M@localfonts` with the count variable `\M@num@localfonts`. The `#1` argument is a control sequence (with all characters having catcode 12 from `\string`), and the `#2` argument is a font name. We have two different approaches depending on whether `\M@localfonts` is empty, i.e. if it's the first time calling `\M@addto@localfonts`. If `\M@localfonts` is

`\@empty`, that means we haven't added any fonts to the list yet, so we increase `\M@num@localfonts`. Otherwise we loop through `\M@localfonts`, and we increase `\M@num@localfonts` only if none of the entries in `\M@localfonts` use the `#2` font. After incrementing (or not) the count, we append `#1` and `#2` to `\M@localfonts`.

```
\def\M@addto@localfonts#1#2#3#4{%
  \begingroup
    \@tempswatrue      % increase by default
    \def\@tempa##1##2##3##4{##2}%
    \@for\@j:=\M@localfonts\do{%
      \edef\@tempb{\expandafter\@tempa\@j}%
      \ifx\@tempbase\@tempb
        \@tempswafalse % if \@tempbase is in list, don't add
      \fi}%
    \expandafter
  \endgroup
  \if@tempswa
    \advance\M@num@localfonts\@ne
  \fi
  \ifx\M@localfonts\@empty
  \else
    \edef\M@localfonts{\M@localfonts,}%
  \fi
  \edef\M@localfonts{\M@localfonts{#1}{#2}{#3}{#4}}}
```

Same thing for symbol fonts.

```
\def\M@addto@symbolfonts#1#2#3#4{%
  \ifx\M@symbolfonts\@empty
  \else
    \edef\M@symbolfonts{\M@symbolfonts,}%
  \fi
  \edef\M@symbolfonts{\M@symbolfonts{#1}{#2}{#3}{#4}}}
```

And font families.

`\M@addto@families`

```
\def\M@addto@families#1{%
  \ifx\M@families\@empty
  \else
```

`\M@families`

```
    \edef\M@families{\M@families,}%
  \fi
```

`\M@families`

```
  \xdef\M@families{\M@families#1}}
```

# 3 Messages and Errors

Some error and informational messages. Table 1 lists all macros defined in this section along with a brief description of their use. We begin with general informational messages.

```
\M@FontChangeInfo 598 \def\M@FontChangeInfo#1#2{\@mathfontinfo{Setting #1 chars to #2!}}
\M@FontFamilyInfo 599 \def\M@FontFamilyInfo#1{\@mathfontinfo{Adding #1 to the nfss.}}
\M@SymbolFontInfo 600 \def\M@SymbolFontInfo#1#2#3{%
                  601   \@mathfontinfo{New symbol font uses TU/#1/#2/#3.}}
\M@NewFontCommandI 602 \def\M@NewFontCommandInfo#1#2#3#4{%
                  603   \@mathfontinfo{New \string#1 uses TU/#2/#3/#4.}}
```

Warnings and errors related to font declaration.

```
\M@NFSSShapesWarni 604 \def\M@NFSSShapesWarning#1#2{%
                  605   \PackageWarningNoLine{mathfont}
                  606   {The nfss family "#1"\MessageBreak
                  607   from line \the\inputlineno\space is missing shapes.\MessageBreak
                  608   You may see some substitutions\MessageBreak
                  609   or errors. See the log file for\MessageBreak
                  610   details}
                  611   \@mathfontinfo{Shapes missing: #2.}}
\M@NoBaseModeError 612 \def\M@NoBaseModeError#1{%
                  613   \PackageError{mathfont}
                  614   {Missing base-mode^^J%
                  615   version of font family "#1"}
                  616   {With LuaTeX, when you tell mathfont to\MessageBreak
                  617   use a font family from the nfss, I try to\MessageBreak
                  618   find aversion of that font in the nfss\MessageBreak
                  619   that uses base mode. I couldn't do that\MessageBreak
                  620   here, so you may see some problems with\MessageBreak
                  621   your math. To resolve this error, either\MessageBreak
                  622   use XeTeX, or make sure the nfss contains\MessageBreak
                  623   a version of your font in base mode, and\MessageBreak
                  624   define \string\<font family>-base to be the nfss\MessageBreak
                  625   name for your base-mode family.^^J}}
\M@FamilyTypeError 626 \def\M@FamilyTypeError#1{%
                  627   \PackageError{mathfont}
                  628   {Invalid family type/^^J%
                  629   optional argument "#1" for \string\mainfont}
                  630   {The optional argument of
                  631   \string\mainfont\space should\MessageBreak
                  632   be one of rm, sf, or tt. You used something\MessageBreak
                  633   else, so I'm changing it to rm.^^J}}
```

**Table 1: Package Messages and Errors and Their Uses**

| Command | Use |
|---|---|
| `\@mathfontinfo` | General informational macro |
| `\M@FontChangeInfo` | When using a new symbol font |
| `\M@FontFamilyInfo` | Declaring new font shape in the NFSS |
| `\M@NewFontCommandInfo` | New local font-change command |
| `\M@SymbolFontInfo` | Declare new symbol font |
| `\M@FamilyTypeError` | Error if bad argument for `\mainfont` |
| `\M@NFSSShapesWarning` | Warning if font is missing shapes |
| `\M@NoBaseModeError` | Error if no base-mode version of a font |
| `\M@InvalidOptionError` | Bad keyword for font-change command |
| `\M@InvalidSupoptionError` | Bad suboption for font-change command |
| `\M@MissingOptionError` | Missing keyword for font-change command |
| `\M@MissingSuboptionError` | Missing suboption for font-change command |
| `\M@FontShapesError` | Tried to add font shapes after preamble |
| `\M@LuaTeXOnlyWarning` | User called LuaTeX-only macro in XeTeX |
| `\M@HModeError` | Font-change command used outside math |
| `\M@MissingCSError` | No macro for font-change command |
| `\M@BadIntegerError` | Non-integer value for font adjustment |
| `\M@NoCharmFileError` | Bad file name for `\charmfile` |
| `\M@NoFontAdjustError` | Macro used without Lua font adjustments |

Error and warning messages for keywords and shape identifiers.

```
\M@InvalidOptionEr 634 \def\M@InvalidOptionError#1{%
                   635   \PackageError{mathfont}
                   636   {Invalid^^Jkeyword "#1" on line \the\inputlineno}
                   637   {You used a character keyword that I'm\MessageBreak
                   638   not familiar with. Check that you spelled\MessageBreak
                   639   everything correctly. To resolve this\MessageBreak
                   640   error, make sure you use keywords that\MessageBreak
                   641   are listed in the documentation.^^J\@gobble}}
\M@InvalidSuboptio 642 \def\M@InvalidSuboptionError#1{%
                   643   \PackageError{mathfont}
                   644   {Invalid^^Jshape identifer "#1" on line \the\inputlineno}
                   645   {You used a suboption/shape identifier\MessageBreak
                   646   that I'm not familiar with. Check that\MessageBreak
                   647   you spelled everything correctly. To\MessageBreak
```

```
  resolve this error, make sure you use\MessageBreak
  shape identifiers that are listed in\MessageBreak
  the documentation.^^J}}
\M@MissingOptionEr \def\M@MissingOptionError{%
  \PackageError{mathfont}
  {Missing keyword on line \the\inputlineno}
  {I didn't see a character keyword\MessageBreak
  where I was expecting to. This can\MessageBreak
  happen if you type ,, or ,= by\MessageBreak
  mistake. To resolve this error,\MessageBreak
  make sure you provided a comma-\MessageBreak
  separated list of keywords.^^J}}
\M@MissingSuboptio \def\M@MissingSuboptionError{%
  \PackageError{mathfont}
  {Missing suboption/^^Jshape identifier on line \the\inputlineno}
  {I didn't see a suboption/shape identifier\MessageBreak
  where I was expecting to. This can happen\MessageBreak
  if you type ,, or =, or == by mistake. To\MessageBreak
  resolve this error, make sure that every\MessageBreak
  = sign comes before a suboption or that you\MessageBreak
  provided a comma-separated list of shape\MessageBreak
  identifiers, depending on the context.^^J}}
```

Error messages regarding arguments previously fed to `\mathfont` and friends.

```
\M@FontShapesError \def\M@FontShapesError{%
  \PackageError{mathfont}
  {^^JCan't declare new font shapes after
  \string\begin{document}}
  {This error means that you (1) requested\MessageBreak
  to use a family/series/shape combination\MessageBreak
  for a font-change command in this package\MessageBreak
  (2) after your document preamble (3) that\MessageBreak
  does not match any font you used for this\MessageBreak
  package in the preamble. To resolve this\MessageBreak
  error, try declaring more font shapes in\MessageBreak
  your preamble with \string\mathfontshapes.^^J}}
\M@LuaTeXOnlyWarni \def\M@LuaTeXOnlyWarning#1{%
  \PackageWarningNoLine{mathfont}
  {Your \string#1\space on line
  \the\inputlineno\MessageBreak
  is mainly for use in LuaTeX with font\MessageBreak
  adjustments enabled. In the current\MessageBreak
  situation, it is probably not doing\MessageBreak
```

```
anything}}
```

Error messages for the `\newmathrm`, etc. commands.

```
\M@MissingCSError \def\M@MissingCSError#1#2{%
  \PackageError{mathfont}
  {Missing control sequence^^J%
  for\string#1\space on line \the\inputlineno}
  {Your command was ignored. Instead of\MessageBreak
  "#2,"\MessageBreak
  I was expecting a single control\MessageBreak
  sequence. To resolve this error,\MessageBreak
  please use one control sequence instead.^^J}}
    \M@HModeError \def\M@HModeError#1{%
  \PackageError{mathfont}
  {Missing \string$ inserted^^J%
  on line \the\inputlineno}
  {I raised an error because you used\MessageBreak
  \string#1\space outside of math mode,\MessageBreak
  which isn't allowed. I inserted a \string$\MessageBreak
  before your control sequence, so we\MessageBreak
  should be all good now.^^J}}
```

We need error messages related to Lua-based font adjustments.

```
\M@NoFontAdjustErr \def\M@NoFontAdjustError#1{%
  \PackageError{mathfont}
  {\string#1^^J%
  is invalid without Lua-based font adjustments}
  {Your control sequence won't do anything\MessageBreak
  without Lua-based font adjustments, but\MessageBreak
  you didn't enable them. To resolve this\MessageBreak
  error, load mathfont with LuaTeX and the\MessageBreak
  package option "adjust" or remove your\MessageBreak
  control sequence.^^J}}
\M@BadIntegerError \def\M@BadIntegerError#1#2{%
  \PackageError{mathfont}
  {Bad argument^^J%
  "#2" for \string#1 on line \the\inputlineno}
  {Your command was ignored. Please make sure\MessageBreak
  that your argument for \string#1\MessageBreak
  is a nonnegative integer.^^J}}
\M@NoCharmFileErro \def\M@NoCharmFileError#1{%
  \PackageError{mathfont}{Missing Charm File}
  {You requested to read the file\MessageBreak
```

```
"#1"\MessageBreak
with \string\charmfile. I can't find that file,\MessageBreak
so I'm ignoring your command. To resolve\MessageBreak
this error, make sure the filename is\MessageBreak
correct, and double check that the file\MessageBreak
is in a directory searchable by TeX.^^J}}
```

# 4 Font Declaration

We come to the fontloader. The main font declaration macro is `\M@newfont`, and it accepts one argument that is a combination of font name and optional OpenType feature information separated by a colon. The macro accomplishes four tasks: (1) it separates the font name and features and stores them in `\@tempbase` and `\@tempfeatures`; (2) it checks whether tbe argument is present as a font name in the NFSS; (3) if not, it declares the font in the NFSS, either with the built-in font loader or using fontspec; and (4) it stores the NFSS family name(s) in `\M@f@ntn@me` and `\M@f@ntn@meb@se`. If `\M@newfont` reaches step 3 after `\begin{document}`, it raises a "can't declare new font shapes" error.

Checking the NFSS happens in two steps. First, `\M@newfont` removes the spaces from its argument and checks whether the result matches a previous call to `\M@newfont`, and if yes, `\M@newfont` uses the family name from the previous call. If no, the macro checks whether the argument with spaces removed appears in the NFSS. (Properly declared NFSS font families shouldn't have spaces in their names because LaTeX ignores spaces when scanning a font family declaration.) If `\M@newfont` finds an NFSS family, it looks at the font shapes associated with that family using the helper macro `\M@check@nfss@shapes`. This command prints a warning if the NFSS is missing any standard series/shape combinations (`m/n`, `m/it`, `b/n`, and `b/it`) for the font family. The package does not add any font shapes to the NFSS at this point in the font-loading process.

If both checks of the NFSS fail, mathfont assumes the NFSS does not contain the user's desired font and proceeds to declare it in the NFSS. For the built-in font loader, font declaration uses the helper macros `\M@fill@nfss@shapes` and `\M@declare@shape`. The first of these two macros loops through series/shape pairs and feeds them to the second macro, which acts as a wrapper around `\DeclareFontShape`. If the user requested to use fontspec for the font loader, `\M@newfont` feeds the font name and requested features to `\fontspec_set_family:Nnn`, which modifies the NFSS accordingly. Advanced users should keep in mind the difference between font declaration and font load-

ing in LaTeX. Font declaration, the subject of this section, means adding font information to the NFSS through `\DeclareFontFamily` and `\DeclareFontShape`, and font loading is when TeX reads a font file into memory via the `\font` primitive. LaTeX handles font loading automatically at a `\selectfont` command or upon entering math mode, so a call to `\M@newfont` will not actually load any fonts into memory, just prepare LaTeX to do so at a later time.

In LuaTeX, users can load fonts in one of three modes, namely `node` (the default), `base`, or `harf`. Node mode works well for text, but it has more limited capabilities for math. Harf mode uses the HarfBuzz renderer and is appropriate for more complicated scripts. Accordingly, when the engine is LuaTeX, mathfont loads fonts once in base mode for math and once in unspecified (so likely node) mode for text. If the user specifies a font family already in the NFSS, `\M@newfont` tries to find a base-mode version of the font family and raises an error if it cannot. When `\M@newfont` declares the font, it does so twice, once in unspecified mode and once in base mode. This is why we have two control sequences for font family names: `\M@f@ntn@me` is the font family, and `\M@f@nt@n@me@base` contains the NFSS family name of the base-mode version of the font. In XeTeX, these control sequences will be identical because mathfont does not load a font multiple times in that case.

During font declaration, mathfont links several pieces of information as follows:

- Given an ⟨*argument*⟩ with spaces removed, `\M@newfont` stores the corresponding NFSS family name in `\M@fontfamily@`⟨*argument*⟩
- The NFSS family name for the base-mode version of the font goes in `\M@fontfamily@base@`⟨*argument*⟩
- Given a ⟨*family name*⟩, `\M@newfont` stores the corresponding base-mode font family name in the control sequence \⟨*family name*⟩`-base`. Users who want to declare their own fonts in the NFSS prior to using them with mathfont should manually define this control sequence so that mathfont knows where your base-mode font lives.
- Each NFSS font family is assigned a unique value of `\M@count` that is stored in `\M@fontid@`⟨*family*⟩.

These macro assignments are global. The difference relative to `\M@f@ntn@me` and `\M@f@ntn@meb@se` is that `\M@f@ntn@me` and `\M@f@ntn@meb@se` are temporary and always hold the font family from the most recent call to `\M@newfont`.

The `\M@check@nfss@shapes` macro checks if a font family has shapes declared in upright, italic, bold, and bold italic. If any of those shapes are missing, we issue a warning. We store the missing series/shape pairs in `\@tempb` to print them as part of the warning message.

\M@check@nfss@shap
```
\def\M@check@nfss@shapes#1{%
  \let\@tempb\@empty
  \let\@tempwarning\@gobble
  \@for\@i:=\mddefault/\shapedefault,%
    \mddefault/\itdefault,%
    \bfdefault/\shapedefault,%
    \bfdefault/\itdefault\do{%
      \expandafter\ifx\csname TU/#1/\@i\endcsname\relax
        \def\@tempwarning{\M@NFSSShapesWarning{#1}}%
        \edef\@tempb{\@tempb, \@i}%
      \fi}%
```

We use a small hack to get everything to print correctly. If all shapes are present, then `\@tempwarning` is `\@gobble`, and the argument disappears. Otherwise, the argument becomes part of the warning message. The `\@gobble` eats the (unnecessary) first comma inside `\@tempb`.

```
  \@tempwarning{\expandafter\@gobble\@tempb}}
```

Next we have commands to add series and shape information to the NFSS for a given font family. The `\M@declare@shape` macro takes several arguments. It checks whether the series/shape pair exists in the NFSS, and if not, it adds it using `\DeclareFontShape`. The argument structure is

- `#1`—NFSS font family name
- `#2`—optional `/B` or `/I` (or `/BI`) suffix on the font name
- `#3`—a list of (default) OpenType feature tags
- `#4`—a list of (the user's) OpenType feature tags
- `#5`—NFSS series identifier
- `#6`—NFSS shape identifier

We assume that the font file reference has already been stored in `\@tempbase`.

\M@declare@shape
```
\def\M@declare@shape#1#2#3#4#5#6{%
  \ifcsname TU/#1/#5/#6\endcsname
  \else
    \DeclareFontShape{TU}{#1}{#5}{#6}{<->"\@tempbase#2:#3;#4"}{}%
  \fi}
```

The `\M@fill@nfss@shapes` command does the work of populating the NFSS with the correct shape information. The argument structure is:

- `#1`—NFSS font family name
- `#2`—a list of (default) OpenType feature tags
- `#3`—a list of (the user's) OpenType feature tags

We call `\M@declare@shape` for each combination of medium/bold series and upright/italic shape, and the result is an entry in the NFSS for each combination. We have separate declarations for regular and small caps because they have different shape identifiers in the NFSS. We manually set `smcp` to be `true` or `false` accordingly.

`\M@fill@nfss@shape`
```
\def\M@fill@nfss@shapes#1#2#3{%
  \@for\@i:={#1}{}{#2;-smcp}{#3}{\mddefault}{\shapedefault},%
    {#1}{/I}{#2;-smcp}{#3}{\mddefault}{\itdefault},%
    {#1}{/B}{#2;-smcp}{#3}{\bfdefault}{\shapedefault},%
    {#1}{/BI}{#2;-smcp}{#3}{\bfdefault}{\itdefault},%
```

And do small caps. If a small caps font face is separate from the main font file, TEX won't be able to find it automatically. In that case, you will have to write your own `fd` file or `\DeclareFontShape` commands.

```
    {#1}{}{#2;+smcp}{#3}{\mddefault}{\scdefault},%
    {#1}{/I}{#2;+smcp}{#3}{\mddefault}{\scdefault\itdefault},%
    {#1}{/B}{#2;+smcp}{#3}{\bfdefault}{\scdefault},%
    {#1}{/BI}{#2;+smcp}{#3}{\bfdefault}{\scdefault\itdefault}%
      \do{\expandafter\M@declare@shape\@i}}
```

We use `\M@split@colon` and `\M@strip@colon` for parsing the argument of `\mathfont`. If the user calls `\mathfont{`⟨*name*⟩`:`⟨*features*⟩`}`, we store the name in `\@tempbase` and the features in `\@tempfeatures`. If the user specifies a name only, then `\@tempfeatures` will be empty. Syntactically, we use `\M@strip@colon` to remove a final `:` the same way we remove a final `=` when we parse the optional argument in the next section.

`\M@split@colon`
```
\def\M@split@colon#1:#2\@nil{%
  \def\@tempbase{#1}%
  \def\@tempfeatures{#2}}
```

`\M@strip@colon`
```
\def\M@strip@colon#1:{#1}
```

The main font-loading macro. It takes a single argument, which should look like either ⟨NFSS *family*⟩ or ⟨*font name*⟩`:`⟨*optional features*⟩. The first thing `\M@newfont` does is split the font name and OpenType features and store each portion in `\@tempbase` and `\@tempfeatures`. If `\@tempfeatures` is not empty, it has an extra colon at the end, so we remove it.

`\M@newfont`
```
\def\M@newfont#1{%
  \expandafter\M@split@colon\expanded{#1}:\@nil
  \ifx\@tempfeatures\@empty\else
    \edef\@tempfeatures{\expandafter\M@strip@colon\@tempfeatures}%
  \fi
```

Then we find the font family name. We remove spaces from the argument, store it in `\@tempa`, and check whether the result matches a previous call to

`\M@newfont`. If yes, we retrieve the font family and base-mode family names and store them in the appropriate control sequences.

```
\edef@nospace\@tempa{#1}%
\ifcsname M@fontfamily@\@tempa\endcsname
  \edef\M@f@ntn@me
    {\csname M@fontfamily@\@tempa\endcsname}%
  \edef\M@f@ntn@meb@se
    {\csname M@fontfamily@base@\@tempa\endcsname}%
\else
```

Next check whether `\@tempa` appears as a font family in the NFSS.

```
  \ifcsname TU+\@tempa\endcsname% is #1 font family in the nfss?
    \let\M@f@ntn@me\@tempa
```

Check that the NFSS contains some font shapes. If any are missing, we issue a warning but do not fill them.

```
    \M@check@nfss@shapes\M@f@ntn@me
```

With LuaTEX, we want a proper base-mode version of the font. In this situation, mathfont expects to find a second font family whose NFSS identifier is stored in `\`⟨*font family*⟩`-base`, and we assume this second font was loaded with `mode=base`. If that information exists, we use it for the base-mode version.

```
    \ifdefined\directlua % if LuaTeX?
      \ifcsname\M@f@ntn@me-base\endcsname % if base-mode version
        \edef\M@f@ntn@meb@se
          {\csname\M@f@ntn@me-base\endcsname}%
```

If the package found a base-mode font, again check that it contains some font shapes, and issue a warning if not.

```
        \M@check@nfss@shapes\M@f@ntn@meb@se
      \else
```

Raise an error if we can't find a base-mode font family, and use the regular font instead.

```
        \M@NoBaseModeError\M@f@ntn@me
        \expandafter\xdef
          \csname\M@f@ntn@me-base\endcsname{\M@f@ntn@me}%
        \let\M@f@ntn@meb@se\M@f@ntn@me
      \fi
```

Base mode is a LuaTEX-only feature version of a font, so we link the base-mode identifier to the same font.

```
    \else % if XeTeX?
      \expandafter\xdef
        \csname\M@f@ntn@me-base\endcsname{\M@f@ntn@me}%
```

```
794         \let\M@f@ntn@meb@se\M@f@ntn@me
795       \fi
```

Now save the font families for reference later.

```
796       \expandafter\xdef
797         \csname M@fontfamily@\@tempa\endcsname
798         {\M@f@ntn@me}%
799       \expandafter\xdef
800         \csname M@fontfamily@base@\@tempa\endcsname
801         {\M@f@ntn@meb@se}%
802     \else                           % if #1 is not in nfss
```

If the argument does not match a known font family, we have to load it ourselves. First, we check that we are still in the document preamble, and if not, we issue an error.

```
803       \ifx\@onlypreamble\@notprerr % if after \begin{document}
804         \M@FontShapesError
805       \else                         % if in preamble
```

If we are in the document preamble, we can still add information to the NFSS. We store the font family name in `\M@f@ntn@me` and print messages in the `log` file. Then we declare the font family and call `\M@fill@nfss@shapes` to declare all the shapes.

```
806         \ifcase\M@loader % are we using default font-loader?
807           \let\M@f@ntn@me\@tempa
808           \M@FontFamilyInfo\M@f@ntn@me
809           \DeclareFontFamily{TU}{\M@f@ntn@me}{}%
810           \M@fill@nfss@shapes{\M@f@ntn@me}{\M@otf@features}
811             {\@tempfeatures}%
```

If the engine is LuaTEX, we load a separate version of the font with `mode=base` and `script=math`. We need to set the `script` because luaotfload processes math information only for fonts with the script set to `math`, and we definitely want that information loaded if present. Then we link the base-mode and regular versions.

```
812           \ifdefined\directlua
813             \edef\M@f@ntn@meb@se{\M@f@ntn@me-base}%
814             \M@FontFamilyInfo\M@f@ntn@meb@se
815             \DeclareFontFamily{TU}{\M@f@ntn@meb@se}{}%
816             \M@fill@nfss@shapes{\M@f@ntn@meb@se}{\M@otf@features}
817               {\@tempfeatures;-nomathparam;mode=base}%
818           \else
819             \let\M@f@ntn@meb@se\M@f@ntn@me
820           \fi
821         \or             % are we using fontspec as font-loader?
```

If the user requested fontspec as the font-loader, we pass the font name and features to `\fontspec_set_family:Nnn` for loading and store the NFSS family name in `\M@f@ntn@me`. In LuaTeX, we request a separate base-mode version by specifying `Renderer=Base` and `Script=Math`.

```
822           \@mathfontinfo{Passing \@tempbase\space
823             to fontspec for handling!}%
824           \csname fontspec_set_family:Nnn\endcsname\M@f@ntn@me
825             {\M@otf@features,\@tempfeatures}{\@tempbase}%
826           \ifdefined\directlua
827             \@mathfontinfo{Passing \@tempbase\space
828               with Renderer=Base to fontspec for handling!}%
829             \csname fontspec_set_family:Nnn\endcsname
830               \M@f@ntn@meb@se
831               {\M@otf@features,\@tempfeatures,%
832                 RawFeature=-nomathparam,Renderer=Base}
833               {\@tempbase}%
834           \else
835             \edef\M@f@ntn@meb@se{\M@f@ntn@me}%
836           \fi
837         \fi
```

Now link the base-mode family name and store the family names for future reference.

```
838         \expandafter\xdef\csname\M@f@ntn@me-base\endcsname
839           {\M@f@ntn@meb@se}%
840         \expandafter\xdef\csname M@fontfamily@\@tempa\endcsname
841           {\M@f@ntn@me}%
842         \expandafter\xdef
843           \csname M@fontfamily@base@\@tempa\endcsname
844             {\M@f@ntn@meb@se}%
845       \fi
846     \fi
847   \fi
```

Finally, assign `\M@count` values to the font family(ies) if needed and save their names in `\M@families`.

```
848   \ifcsname M@fontid@\M@f@ntn@me\endcsname % need new \M@count?
849   \else
850     \expandafter\xdef
851       \csname M@fontid@\M@f@ntn@me\endcsname{\the\M@count}%
852     \global\advance\M@count\@ne
853     \M@addto@families{\M@f@ntn@me}%
854   \fi
```

Same thing with the base-mode version of the font.

```
\ifcsname M@fontid@\M@f@ntn@meb@se\endcsname
\else
  \expandafter\xdef
    \csname M@fontid@\M@f@ntn@meb@se\endcsname{\the\M@count}%
  \global\advance\M@count\@ne
  \M@addto@families{\M@f@ntn@meb@se}%
\fi}
```

The font-loading commands should appear only in the preamble.

```
\@onlypreamble\M@declare@shape
\@onlypreamble\M@fill@nfss@shapes
```

# 5 Parse Input

This section provides the macros to parse the optional argument of `\mathfont`. We have two parts to this section: error checking and parsing. For parsing, we extract option and suboption information, and for error checking, we make sure that both are valid. The command `\M@check@opt` accepts a macro containing (what is hopefully) the text of a keyword-option. The macro defines `\@temperror` to be an invalid option error and loops through all possible options. If the argument matches one of the correct possibilities, mathfont changes `\@temperror` to `\relax`. The macro ends by calling `\@temperror` and issuing an error if and only if the argument is invalid. If `\M@check@opt` finds a valid keyword-option, it changes `\if@optionpresent` to true.

`\M@check@opt`

```
\def\M@check@opt#1{%
  \@optionpresentfalse          % set switch to false by default
  \ifx#1\@empty
    \M@MissingOptionError
  \else
    \let\@temperror\M@InvalidOptionError % error by default
    \@for\@j:=\M@keys\do{%
      \ifx\@j#1%
        \let\@temperror\@gobble % eliminate error
        \@optionpresenttrue     % set switch to true
      \fi}%
    \@temperror{#1}%
  \fi}
```

Now we have to parse the optional argument of `\mathfont`. The macro `\M@parse@option` carries out the following tasks:

1. Store the keyword in `\@temp@opt` and the suboption (if present) in `\@temp@sub`. Set the boolean corresponding to the presence of a suboption.
2. Check that `\@temp@opt` is actually a keyword and set the corresponding boolean.
3. Convert `\@temp@sub` into NFSS series and shape keywords.

We want to allow the user to specify options using an xkeyval-type syntax. However, we do not need the full package; a slim few lines of code will suffice. When `\mathfont` reads one segment of *text* from its optional argument, it calls `\M@parse@option`⟨*text*⟩`=\@nil`. The `\M@parse@option` macro splits the option and suboption by looking for the first `=`.

`\M@strip@equals`
`\M@parse@option`

```
\def\M@strip@equals#1={#1}
\def\M@parse@option#1=#2\@nil{%
  \def\@temp@opt{#1}%     % store option
  \def\@temp@sub{#2}%     % store suboption
```

After storing the option and suboption, check for errors. If the user specified a suboption, `\@temp@sub` contains ⟨*suboption*⟩`=`, and we use `\M@strip@equals` to get rid of the extra `=`. If the user does not specify a suboption, `\@temp@sub` will be empty. After `\M@parse@sub`, the NFSS series and shape codes for the suboption (if provided) will be stored in `\@tempseries` and `\@tempshape`.

```
  \M@check@opt\@temp@opt
```

At this point, we have three possibilities for `\@temp@sub`:

1. `\@temp@sub` is `=`, which means the user wrote something like `,`⟨*keyword*⟩`=,` and indicates a missing suboption.
2. `\@temp@sub` is empty, which indicates the user didn't provide a suboption, and we should use the default setting.
3. Otherwise, we try to extract series and shape information from `\@temp@sub`.

We process `\@temp@sub` in that order. In case 3, we expect `\@temp@sub` to look like ⟨*shape identifier*⟩`=`, so we have to strip the final `=` before processing.

```
  \begingroup
    \def\@tempa{=}%
    \expandafter
  \endgroup
  \@suboptionpresentfalse    % set switch to false by default
  \ifx\@temp@sub\@tempa      % if missing suboption
    \M@MissingSuboptionError
  \else
```

```
    \ifx\@temp@sub\@empty    % if no suboption provided
    \else                    % if suboption provided, parse it
      \@suboptionpresenttrue % set switch to true
      \edef\@temp@sub{\expandafter\M@strip@equals\@temp@sub}%
      \M@parse@sub\@temp@sub
    \fi
  \fi}
```

Now a macro to convert a shape identifier into NFSS series and shape codes. Here the `#1` argument is a control sequence such as `\@temp@sub`. The first thing we do is check whether `#1` ends in an asterisk and set `\M@base@` accordingly. This boolean is true if we use base mode (if no asterisk) and false otherwise (if asterisk). Note that `\M@parse@sub` should always be called when `\@suboptionpresent` is set to true. If we encounter a bad shape identifier in `#1`, we change `\@suboptionpresent` to false. (So it may be more accurate to name the conditional something like `\@suboptioncheck`, but I'm keeping the name as is to match `\@optionpresent`.)

`\M@parse@sub`

```
\def\M@parse@sub#1{%
  \expanded{\noexpand\in@*{#1}}%
```

If `#1` contains an asterisk, we check that it is the final character in `#1` and strip it.

```
  \ifin@
    \begingroup
      \expandafter\M@split@star#1\@nil
      \ifx\@tempb\@empty
        \expanded{\endgroup   % first branch \endgroup
          \def\noexpand#1{\@tempa}}%
        \M@mode@true
```

If the asterisk is not the final character, that probably means something went wrong. (But we'll catch the problem later.)

```
      \else
        \endgroup             % second branch \endgroup
        \M@mode@false
      \fi
  \else
    \M@mode@false
  \fi
```

If the shape identifier contains a `/`, we interpret it as NFSS identifiers and do not check further, and otherwise, we check that the argument is one of `upright`, `italic`, `bold`, or `bolditalic`. We also accept `roman` for backwards compatibility. We store NFSS information in `\@tempseries` and `\@tempshape`.

```
\expanded{\noexpand\in@/{#1}}%
\ifin@
  \expandafter\M@split@slash#1\@nil
\else
```

If the user wrote out a shape identifier, we have a bit more checking to do: we have to check whether `#1` is one of `roman`, `upright`, `italic`, `bold`, or `bolditalic`. We let `\@tempa` be equal to different strings inside a group, and for each possibility, if `#1` is that string, we set `\@tempseries` and `\@tempshape` to the correct definitions. We have multiple `\endgroup`s for the same `\begingroup` because they occur on different branches of the compound conditional.

```
    \begingroup
    \def\@tempa{roman}%
      \ifx#1\@tempa
        \endgroup         % first branch \endgroup
```

If `\@temp@sub` is `roman`, we change it to `upright`.

```
        \def#1{upright}%
        \let\@tempseries\mddefault
        \let\@tempshape\shapedefault
      \else
        \def\@tempa{upright}%
        \ifx#1\@tempa
          \endgroup       % second branch \endgroup
          \let\@tempseries\mddefault
          \let\@tempshape\shapedefault
        \else
          \def\@tempa{italic}%
          \ifx#1\@tempa
            \endgroup     % third branch \endgroup
            \let\@tempseries\mddefault
            \let\@tempshape\itdefault
          \else
            \def\@tempa{bold}%
            \ifx#1\@tempa
              \endgroup   % fourth branch \endgroup
              \let\@tempseries\bfdefault
              \let\@tempshape\shapedefault
            \else
              \def\@tempa{bolditalic}%
              \ifx#1\@tempa
                \endgroup % fifth branch \endgroup
```

```
                \let\@tempseries\bfdefault
                \let\@tempshape\itdefault
              \else
```

Otherwise, the user specified a bad suboption.

```
                \endgroup % sixth branch \endgroup
                \@suboptionpresentfalse
                \M@InvalidSuboptionError{#1}%
              \fi
            \fi
          \fi
        \fi
      \fi
    % no \fi at this level of indentation
  \fi}
```

Helper macro to parse the shape identifier if it contains a `/` character.

`\M@split@slash`

```
\def\M@split@slash#1/#2\@nil{%
  \def\@tempseries{#1}%
  \def\@tempshape{#2}}
```

Helper macros for processing asterisks in the shape identifier. The first macro here should be called inside a group since it uses `\@tempa` and `\@tempb` and therefore will mess with temporary assignments otherwise.

`\M@split@star`

```
\def\M@split@star#1*#2\@nil{%
  \def\@tempa{#1}%
  \def\@tempb{#2}}
```

`\M@strip@star`

```
\def\M@strip@star#1*{#1}
```

We code a general-purpose definition macro that defines its first argument to be the second argument fully expanded and with spaces removed.

```
\long\def\edef@nospace#1#2{%
  \edef#1{\expandafter\zap@space\expanded{#2} \@empty}}
```

Perhaps something that sets spaces to `\catcode9` and then retokenizes `#2` would be better, but I don't think it matters very much.

# 6 Default Font Changes

This section documents default font changes. We have three main user-level commands in this section: `\mathfont`, which makes changes for math mode; `\mainfont`, which makes changes for horizontal mode; and `\documentfont`, which calls both `\mathfont` and `\mainfont`. The `\mainfont` command is straightforward: set `\rmdefault` to be the font family corresponding to the

user's argument and, when necessary, call `\selectfont` to change the font family in use. (So unlike `\mathfont`, `\mainfont` can result in font loading rather than just font declaration.)

The `\mathfont` command serves as the primary font-changing command for this package and is more complicated than `\mainfont`. This command is a wrapper around `\@mathfont`, the internal command that does the actual font changing, and when a user calls `\mathfont`, the `\@mathfont` macro carries out the following tasks:

1. Call `\M@newfont` on the mandatory argument of `\mathfont`, and store `\M@count` values.
2. Loop through the optional argument of `\mathfont` and determine NFSS series and shape codes from any suboptions using macros from the previous section.
3. On each iteration, check whether mathfont added a symbol font to the NFSS that uses the series and shape corresponding to the current suboption (or the default series and shape if there is no suboption). If not, call `\DeclareSymbolFont` to add the symbol font or raise a "Can't declare new font shapes after `\begin{document}`" error.
4. Call `\M@`⟨*keyword*⟩`@set` to actually change the font.

Each `\M@`⟨*keyword*⟩`@set` macro is a wrapper around `\Umathcode` declarations and is defined in the last section of this document.

For a given ⟨*keyword*⟩, `\M@fontinfo@`⟨*keyword*⟩ stores the human-readable name of the new default font for ⟨*keyword*⟩ as well as the NFSS series and shape identifiers. (This is the name in `\@tempbase`, not `\M@f@ntn@me`.) We use this information for writing messages to the user. Additionally, `\M@`⟨*keyword*⟩`shape` holds the series and shape pair for ⟨*keyword*⟩. If the user specified a suboption, the contents of this macro come from the suboption via `\M@parse@sub`, and if the user did not specify a suboption, the information comes from `\`⟨*keyword*⟩`default`.

For a combination of ⟨*font family*⟩, ⟨*series*⟩, and ⟨*shape*⟩ identifiers, mathfont calls the associated symbol font `M`⟨*count*⟩`-`⟨*series*⟩`/`⟨*shape*⟩, where ⟨*count*⟩ is the `\M@count` value associated to the ⟨*font family*⟩, i.e. the contents of `\M@fontid@`⟨*font family*⟩. The ⟨*series*⟩ and ⟨*shape*⟩ values should be entries in the NFSS. For example, calls to `\mathfont` will often result symbol font names like `M0-m/n`. Including a count value in the symbol font name serves two purposes. First, it enables consistent formatting of symbol font names regardless of underlying NFSS family names. In particular, using the built-in fontloader vs. fontspec will result in different family names for the same font face with the same OpenType features, but the symbol font names will still match. Second,

it simplifies the symbol font names and makes them human readable regardless of the underlying NFSS family names, which may be complicated.

The last two user-level commands in this section are `\mathconstantsfont` and `\mathfontshapes`. The first of these two commands only works in LuaTEX and makes TEX use the math parameters from a given font when formatting equations. Traditional TEX expects to see extra parameters in the font(s) in \⟨*math style*⟩`font2` and \⟨*math style*⟩`font3`, and it uses those parameters to format equations. LuaTEX can pull these extra parameters from the fonts in any math family, and `\mathconstantsfont` tells LuaTEX to do so for a given font family. The command `\mathfontshapes` declares extra font shapes for the NFSS as well as extra symbol fonts. The purpose of this command is to allow the user to declare symbol fonts in the document preamble for use after `\begin{document}`.

We begin by coding `\mainfont`. This command is a wrapper around `\@mainfont`.

`\mainfont` 968
```
\protected\def\mainfont{\@testopt{\@mainfont}{rm}}
```

Now the internal `\@mainfont` command. This command doesn't do anything in math mode, so we include `\@nomath`. We check whether `#1` is one of `rm`, `sf`, `tt`, or empty. If not, we issue an error and change `\@tempa` to `rm`. We use `\@tempswa` to check whether we are storing the font family name in \⟨*type*⟩`default` macros.

`\@mainfont` 969
```
\def\@mainfont[#1]#2{%
  \@nomath\mainfont
  \M@newfont{#2}%
  \edef@nospace\@tempa{#1}%
  \@tempswatrue
  \def\@tempb{rm}%
  \ifx\@tempa\@tempb
  \else
    \def\@tempb{sf}%
    \ifx\@tempa\@tempb
    \else
      \def\@tempb{tt}%
      \ifx\@tempa\@tempb
      \else
        \ifx\@tempa\@empty
          \@tempswafalse
        \else
          \M@FamilyTypeError\@tempa
          \def\@tempa{rm}%
        \fi
```

```
      \fi
    \fi
  \fi
```

Now close the group and save the font family in `\⟨#1⟩default` and change the default family to `#1`.

```
  \if@tempswa
    \expandafter\let\csname\@tempa default\endcsname\M@f@ntn@me
\familydefault
    \edef\familydefault{\expandafter\noexpand
      \csname\@tempa default\endcsname}%
  \fi
```

If the current font is not `\nullfont` or `\M@f@ntn@me`, select `\M@f@ntn@me` as the font family.

```
  \expandafter\ifx\the\font\nullfont
  \else
    \ifx\f@family\M@f@ntn@me
    \else
      \fontfamily\M@f@ntn@me\selectfont
    \fi
  \fi}
```

Now we come to `\mathfont`. This macro is a wrapper around `\@mathfont` that we use to check for an optional argument. The default argument is `\M@defaultkeys`.

```
\protected\def\mathfont{\@testopt{\@mathfont}{\M@defaultkeys}}
```

The internal font-changing command. We call `\M@newfont` on the mandatory argument of `\mathfont`, which stores the NFSS family name(s) in `\M@f@ntn@me` and `\M@f@ntn@meb@se`. We check whether each family name corresponds to a value of `\M@newcount`, and if not, we define it. Throughout the definition of `\mathfont`, `\@tempa` stores the value of `\M@count` that corresponds to the `#1` font, and `\@tempb` stores the value of `\M@count` that corresponds to the `#1` font in base mode.

```
\def\@mathfont[#1]#2{%
  \wlog{}%
  \M@newfont{#2}%
```

Temporarily store values of `\M@count`.

```
  \edef\@tempa{\csname M@fontid@\M@f@ntn@me\endcsname}%
  \edef\@tempb{\csname M@fontid@\M@f@ntn@meb@se\endcsname}%
```

Expand, zap spaces from, and store the optional argument in `\@tempc`, and then perform the loop. (At that point, we do not need `\@tempc` anymore.) We store the current keyword-suboption pair in `\@i` and feed it to `\M@parse@option`.

```
\edef@nospace\@tempc{#1}%
\@for\@i:=\@tempc\do{%
  \expandafter\M@parse@option\@i=\@nil
  \if@optionpresent
```

If the user did not specify a suboption, parse the default option, and use that instead. We set `\@suboptionpresent` to true before calling `\M@parse@sub` so that we can check whether the default shape identifier is valid. If `\@suboptionpresent` is false after `\M@parse@sub`, we use `m/n` as the series/shape pair.

```
    \if@suboptionpresent
    \else
      \@suboptionpresenttrue
      \expandafter\M@parse@sub
        \csname\@temp@opt default\endcsname
      \if@suboptionpresent
      \else
        \let\@tempseries\mddefault
        \let\@tempshape\shapedefault
      \fi
    \fi
```

Now store the series and shape in `\M@`⟨*option*⟩`shape`.

```
    \expandafter\edef\csname M@\@temp@opt shape\endcsname{%
        \@tempseries/\@tempshape}%
```

At this point we have the information we need to declare the symbol font, namely the NFSS family (`\M@f@ntn@me` or `\M@f@ntn@meb@se`), series (`\@tempseries`), and shape (`\@tempshape`). We check if the symbol font we want to use is defined, and if not, we define it. We have two cases to consider: if `\M@base@` is true, we use the base-mode version of the font (corresponding to information in `\@tempb` and `\M@f@ntn@meb@se`), and if `\M@base@` is false, we use the default-mode version of the font (corresponding to information in `\@tempa` and `\M@f@ntn@me`). We let `\@tempc` be the count value in use for the current iteration of the loop.

```
    \ifM@mode@ % if default/node mode
      \let\@tempc\@tempa
      \ifcsname symM\@tempa-\@tempseries/\@tempshape\endcsname
```

If the symbol font has not been declared, check that we are still in the preamble. If no, issue an error message.

```
      \else
        \ifx\@onlypreamble\@notprerr
          \M@FontShapesError
```

Otherwise, we declare the symbol font.

```
1033          \else
1034            \M@SymbolFontInfo{\M@f@ntn@me}
1035              {\@tempseries}{\@tempshape}%
1036            \M@addto@symbolfonts
1037              {M\@tempa-\@tempseries/\@tempshape}
1038              {\@tempbase}{\@tempseries}{\@tempshape}%
1039            \DeclareSymbolFont
1040              {M\@tempa-\@tempseries/\@tempshape}{TU}
1041              {\M@f@ntn@me}{\@tempseries}{\@tempshape}%
1042          \fi
1043        \fi
```

Now do the same thing for default (node) mode.

```
1044      \else      % if default/node mode
1045        \let\@tempc\@tempb
1046        \ifcsname
1047          symM\@tempb-\@tempseries/\@tempshape\endcsname
1048        \else
1049          \ifx\@onlypreamble\@notprerr
1050            \M@FontShapesError
```

The only difference is we use different font family and symbol font names.

```
1051          \else
1052            \M@SymbolFontInfo{\M@f@ntn@meb@se}
1053              {\@tempseries}{\@tempshape}%
1054            \M@addto@symbolfonts
1055              {M\@tempb-\@tempseries/\@tempshape}
1056              {\@tempbase(base)}{\@tempseries}
1057              {\@tempshape}%
1058            \DeclareSymbolFont
1059              {M\@tempb-\@tempseries/\@tempshape}{TU}
1060              {\M@f@ntn@meb@se}{\@tempseries}{\@tempshape}%
1061          \fi
1062        \fi
1063      \fi
```

We store the new font information so we can write it to the `log` file `\AtBeginDocument` and send an informational message to the user.

```
1064      \expandafter\edef
1065        \csname M@fontinfo@\@temp@opt\endcsname{%
1066          {\@tempbase}{\@tempseries}{\@tempshape}}%
1067      \M@FontChangeInfo{\@temp@opt}{\@tempbase}%
```

We have extra information to keep track of when `\@temp@opt` is `bb`, `cal`, `frak`, `bcal`, or `bfrak` because then mathfont effectively creates a new local font-change command. We make sure that information gets added to `\M@localfonts` (a macro that tracks the font names used for local font changes).

```
      \@tfor\@j:={bb}{cal}{frak}{bcal}{bfrak}\do{%
        \ifx\@temp@opt\@j
          \M@addto@localfonts{\expandafter\string
              \csname math\@temp@opt\endcsname}
            {\@tempbase}{\@tempseries}{\@tempshape}%
          \@break@tfor
        \fi}%
```

And now the magic happens!

```
      \csname M@\@temp@opt @set\endcsname % set default font
      \csname M@\@temp@opt true\endcsname % set switch to true
    \fi}%
```

Display concluding messages for the user.

```
  \ifx\@tempa\@empty
    \wlog{The \string\mathfont\space command on line
      \the\inputlineno\space did not change the font
      for any characters!}%
  \else
    \typeout{:: mathfont :: Using font \@tempbase\space
      on line \the\inputlineno.}%
  \fi
  \wlog{}}
```

Using `\documentfont` calls `\mainfont`, `\mathfont`, and `\mathconstantsfont`. It also calls `\mathfontcommands` if the user is still in the preamble. The optional argument gets fed directly to `\@mainfont`.

`\documentfont`
```
\protected\def\documentfont{\@testopt{\@documentfont}{rm}}
```

The internal command.

`\@documentfont`
```
\def\@documentfont[#1]#2{%
  \@mainfont[#1]{#2}%
  \mathfont{#2}%
  \ifdefined\directlua
    \mathconstantsfont{#2}%
  \fi
  \ifx\@onlypreamble\@notprerr
  \else
    \mathfontcommands{#2}%
  \fi}
```

For backwards compatibility, we make `\setfont` expand to `\documentfont` (plus a warning message).

```
\setfont
\protected\def\setfont{%
  \PackageWarningNoLine{mathfont}
  {Using \string\setfont\space is deprecated; I\MessageBreak
  replaced it with \string\documentfont}%
  \documentfont}
```

The macro `\mathconstantsfont` chooses a font for setting math parameters. It is intended for LuaTEX when mathfont can adjust text fonts and add a MathConstants table. It issues a warning if called without font adjustments enabled. First, we check for an optional argument, which should be a shape identifier.

```
\M@SetMathConstant
\let\M@SetMathConstants\relax
\mathconstantsfont
\protected\def\mathconstantsfont{%
  \@testopt{\@mathconstantsfont}{upright}}
```

The internal command that does the processing. We begin by feeding the `#2` argument to `\M@newfont` and parsing the `#1` argument.

```
\@mathconstantsfon
\def\@mathconstantsfont[#1]#2{%
  \M@newfont{#2}%
  \edef@nospace\@tempa{#1}%
  \M@parse@sub\@tempa
```

Store the family, series, and shape information. Because it doesn't make sense to use a font that is loaded with node mode, we force use of the base-mode version of the font regardless of the value of `\ifM@mode@`.

```
\m@th@const@nts@f@
  \let\m@th@const@nts@f@mily\M@f@ntn@meb@se
\m@th@const@nts@se
  \let\m@th@const@nts@series\@tempseries
\m@th@const@nts@sh
  \let\m@th@const@nts@sh@pe\@tempshape
```

Temporarily store the value of `\M@count`.

```
  \edef\@tempa{\csname M@fontid@\m@th@const@nts@f@mily\endcsname}%
```

Now check whether the desired symbol font has been declared. If no, we declare it or issue an error.

```
  \ifcsname symM\@tempa-\@tempseries/\@tempshape\endcsname
  \else
    \ifx\@onlypreamble\@notprerr
      \M@FontShapesError
    \else
```

Declare the symbol font.

```
      \M@SymbolFontInfo{\m@th@const@nts@f@mily}
        {\@tempseries}{\@tempshape}%
      \M@addto@symbolfonts
```

```
1122        {M\@tempb-\@tempseries/\@tempshape}
1123        {\@tempbase(base)}{\@tempseries}{\@tempshape}%
1124      \DeclareSymbolFont
1125        {M\@tempa-\@tempseries/\@tempshape}{TU}
1126        {\m@th@const@nts@f@mily}{\@tempseries}{\@tempshape}%
1127    \fi
1128  \fi
```

We come to the tricky problem of making sure to use the correct MathConstants table. LuaTEX automatically initializes all math parameters based on the most recent `\textfont`, etc. assignment, so we want to tell LaTeX to reassign whatever font we're using to the correct math family right after we finish assigning other math fonts. This is possible, but the implementation is super hacky. When LaTeX enters math mode, it checks whether it needs to redo any math family assignments, typically because of a change in font size, and if so, it calls `\getanddefine@fonts` repeatedly to append `\textfont`, etc. assignments onto the macro `\math@fonts`. Usually `\math@fonts` is empty because this process always happens inside a group, so we can hook into the code by defining `\math@font` to be `\aftergroup`⟨*extra code*⟩. In this case, the *extra code* will be another call to `\getanddefine@fonts`.

We initialize `\M@SetMathConstants` to be `\relax`, and we define it the first time the user calls `\mathconstantsfont`. When that happens, mathfont begins by calling `\getanddefine@fonts` inside a group and uses as arguments the upright face of the font corresponding to `#1`. That puts the `\textfont`, `\scriptfont`, and `\scriptscriptfont` assignments corresponding to `#1` inside `\math@fonts`. Then we call `\math@fonts`, and to avoid an infinite loop, we gobble the `\aftergroup\M@SetMathConstants` macros that mathfont has inserted at the start of `\math@fonts`. Setting `\globaldefs` to 1 makes the `\textfont`, etc. assignments from `\getanddefine@fonts` global when we call `\math@fonts`.

```
1129  \ifx\M@SetMathConstants\relax
\M@SetMathConstant 1130    \protected\def\M@SetMathConstants{%
1131      \begingroup
1132      \escapechar\m@ne
1133      \expandafter\getanddefine@fonts
1134        \csname symM%
1135          \csname M@fontid@\m@th@const@nts@f@mily\endcsname
1136          -\m@th@const@nts@series/\m@th@const@nts@sh@pe
1137          \expandafter
1138        \endcsname % expands to e.g. \symM0-m/n
1139        \csname TU/\m@th@const@nts@f@mily
1140                  /\m@th@const@nts@series
```

```
                  /\m@th@const@nts@sh@pe
        \endcsname % expands to \TU/<family>/<series>/<shape>
      \globaldefs\@ne
      \expandafter\@gobbletwo\math@fonts % avoid infinite loop
      \endgroup}%
  \fi
  \ifM@adjust@font
  \else
    \M@LuaTeXOnlyWarning\mathconstantsfont
  \fi}
\math@fonts \def\math@fonts{\aftergroup\M@SetMathConstants}
```

Now `\mathfontshapes`. This macro adds extra font shapes to the NFSS and defines symbol fonts. Its purpose is to allow the user to easily declare symbol fonts in the preamble without using them right away. The user-level command is a wrapper around `\@mathfontshapes`.

```
\mathfontshapes \protected\def\mathfontshapes{\@testopt{\@mathfontshapes}
  {upright,upright*,italic,bold}}
```

For the internal command, we feed the font name to `\M@newfont` and then loop through the optional argument. For each shape identifier in the optional argument, we parse it and then use it to declare a symbol font. This macro is very similar to parts of `\@mathfont`.

```
\@mathfontshapes \protected\def\@mathfontshapes[#1]#2{%
  \wlog{}%
  \M@newfont{#2}%
```

As in `\@mathfont`, we temporarily store values of `\M@count`.

```
  \edef\@tempa{\csname M@fontid@\M@f@ntn@me\endcsname}%
  \edef\@tempb{\csname M@fontid@\M@f@ntn@meb@se\endcsname}%
```

Expand, zap spaces, and loop through the optional argument.

```
  \edef@nospace\@tempc{#1}%
  \@for\@i:=\@tempc\do{%
    \@suboptionpresenttrue
    \M@parse@sub\@i
```

Then check whether the symbol font exists and if not, declare it. We start with default/node mode. We print a message in the log file, add the information to `\M@symbolfonts`, and call `\DeclareSymbolFont`.

```
    \if@suboptionpresent
      \ifM@mode@ % if default/node mode
        \ifcsname symM\@tempa-\@tempseries/\@tempshape\endcsname
        \else
          \M@SymbolFontInfo{\M@f@ntn@me}
```

```
            {\@tempseries}{\@tempshape}%
          \M@addto@symbolfonts
            {M\@tempa-\@tempseries/\@tempshape}
            {\@tempbase}{\@tempseries}{\@tempshape}%
          \DeclareSymbolFont
            {M\@tempa-\@tempseries/\@tempshape}{TU}
            {\M@f@ntn@me}{\@tempseries}{\@tempshape}%
        \fi
```

And do the same thing for base mode.

```
      \else      % if base mode
        \ifcsname symM\@tempb-\@tempseries/\@tempshape\endcsname
        \else
          \M@SymbolFontInfo{\M@f@ntn@meb@se}
            {\@tempseries}{\@tempshape}%
          \M@addto@symbolfonts
            {M\@tempb-\@tempseries/\@tempshape}
            {\@tempbase(base)}{\@tempseries}{\@tempshape}%
          \DeclareSymbolFont
            {M\@tempb-\@tempseries/\@tempshape}{TU}
            {\M@f@ntn@meb@se}{\@tempseries}{\@tempshape}%
        \fi
      \fi
    \fi}}
\@onlypreamble\mathfontshapes
\@onlypreamble\@mathfontshapes
```

# 7 Local Font Changes

This section deals with local font changes. The main user-level macro in this section is `\newmathfontcommand`, which creates macros that change the font for math alphabet characters and is basically a wrapper around `\DeclareMathAlphabet`. Other user-level commands are a special case of this one.

We begin with two helper macros. First is `\M@check@csarg`, which accepts two arguments and handles some error checking. The `#1` argument is a user-level command that we use in error messaging, and `#2` should be a single control sequence. The way `\M@check@csarg` scans the following tokens is a bit tricky: (1) check the length of the argument (number of tokens) by seeing if `\@gobble` eats it completely; and (2) check that the argument is a control sequence. If the user specifies an argument of the form `{..}`, i.e. extra text inside braces,

the `\ifcat` will catch it and issue an error. If `\M@check@csarg` likes the input, it sets `\ifM@arg@good` to true, and otherwise, it sets `\ifM@arg@good` to false.

```
\M@check@csarg
\def\M@check@csarg#1#2{%
  \expandafter\ifx\expandafter\@nnil\@gobble#2\@nnil % good
    \ifcat\relax\noexpand#2 % good
      \M@arg@goodtrue
    \else % if #2 not a control sequence
      \M@MissingCSError#1{\detokenize{#2}}
      \M@arg@goodfalse
    \fi
  \else   % if #2 is multiple tokens
    \M@MissingCSError#1{\detokenize{#2}}
    \M@arg@goodfalse
  \fi}
```

The macro `\M@checkspecials` accepts a control sequence as its `#1` argument and a font name as its `#2` argument, and it checks whether `#1` is `\mathbb` or a related command. If yes, we assume that the user is using some variant of `\newmathrm` instead of, for example, `\mathfont[bb]`, so we do some processing analogous to what happens inside `\@mathfont`.

```
\M@checkspecials
\def\M@checkspecials#1#2{%
  \in@#1{\mathbb\mathcal\mathfrak\mathbcal\mathbfrak}
  \ifin@
```

We set `\escapechar` to $-1$ and use `\@gobblefour` to remove the `\math` from the start of `#1`. The string of `\expandafter`s hits the `\string` inside `\@tempa`, and then the `\edef` expands the `\@gobblefour`. We are left with just the keyword inside `\@tempa`.

```
    \begingroup
      \escapechar\m@ne
      \expandafter
    \endgroup
    \expandafter\edef\expandafter\@tempa\expandafter{%
      \expandafter\@gobblefour\string#1}%
```

Then write a message to the `log` file and set the corresponding boolean to true.

```
    \@mathfontinfo{Interpreting your new macro \string#1\space
      as \@tempa\space chars.}%
    \@mathfontinfo{Setting \expandafter\string
      \csname ifM@\@tempa\endcsname\space to true.}%
    \csname M@\@tempa true\endcsname
```

And store the information to write to the `log` file `\AtBeginDocument`.

```
    \expandafter\edef\csname M@fontinfo@\@tempa\endcsname{%
```

```
      {\@tempbase}{\@tempseries}{\@tempshape}}%
    \expandafter\edef\csname M@\@tempa shape\endcsname
      {\@tempseries/\@tempshape}%
  \fi}
```

Now declare the math alphabet. This macro first checks that its `#1` argument is a control sequence using `\M@check@csarg`. If yes, load the `#2` argument with `\M@newfont`, call `\DeclareMathAlphabet`, and check whether `#1` is `\mathbb` or a related command. Finally, add `#1` and `#2` to the list of local font-change commands.

```
\newmathfontcomman \protected\def\newmathfontcommand#1#2#3#4{%
  \M@check@csarg\newmathfontcommand{#1}%
  \ifM@arg@good
    \M@newfont{#2}
    \M@NewFontCommandInfo{#1}{\M@f@ntn@meb@se}{#3}{#4}
    \DeclareMathAlphabet{#1}{TU}{\M@f@ntn@meb@se}{#3}{#4}
    \M@checkspecials{#1}{\@tempbase}
    \M@addto@localfonts{\string#1}{\@tempbase}{#3}{#4}
  \fi}
\@onlypreamble\newmathfontcommand
```

Then define macros that create local font-changing commands with default series and shape information. Because they're all similar, we metacode them. We define the commands themselves with `\define@newmath@cmd`. The argument structure is:

- `#1`—`\newmath`⟨*key*⟩ macro name
- `#2`—font series
- `#3`—font shape
- `##1`—the user's control sequence
- `##2`—the user's font information (family name)

We feed `##1`, `##2`, `#2`, and `#3` to `\newmathfontcommand`, and we load `##2` with `\M@newfont`. Each `\newmath`⟨*key*⟩ macro will check its first argument using `\M@check@csarg` and then call `\newmathfontcommand` on both of its two arguments. We store the list of `\newmath`⟨*key*⟩ commands that we want to define with their series and shape information in `\M@default@newmath@cmds`, and we loop through it with `\@for`.

```
\M@define@newmath@ \def\M@define@newmath@cmd#1#2#3{%
                     \protected\def#1##1##2{%
    \M@check@csarg     \M@check@csarg{#1}{##1}
                       \newmathfontcommand{##1}{##2}{#2}{#3}}}
\M@default@newmath \def\M@default@newmath@cmds{%
```

```
  \newmathrm{\mddefault}{\shapedefault},%
  \newmathit{\mddefault}{\itdefault},%
  \newmathbf{\bfdefault}{\shapedefault},%
  \newmathbfit{\bfdefault}{\itdefault},%
  \newmathsc{\mddefault}{\scdefault},%
  \newmathscit{\mddefault}{\scdefault\itdefault},%
  \newmathbfsc{\bfdefault}{\scdefault},%
  \newmathbfscit{\bfdefault}{\scdefault\itdefault}}
\@for\@i:=\M@default@newmath@cmds\do{%
  \expandafter\M@define@newmath@cmd\@i}
\@onlypreamble\newmathrm
\@onlypreamble\newmathit
\@onlypreamble\newmathbf
\@onlypreamble\newmathbfit
\@onlypreamble\newmathsc
\@onlypreamble\newmathscit
\@onlypreamble\newmathbfsc
\@onlypreamble\newmathbfscit
\@onlypreamble\M@define@newmath@cmd
\let\M@default@newmath@cmds\@undefined
```

The command `\mathfontcommands` sets all the default local font-change commands at once.

```
\protected\def\mathfontcommands#1{%
  \newmathrm\mathrm{#1}
  \newmathit\mathit{#1}
  \newmathbf\mathbf{#1}
  \newmathbfit\mathbfit{#1}
  \newmathsc\mathsc{#1}
  \newmathscit\mathscit{#1}
  \newmathbfsc\mathbfsc{#1}
  \newmathbfscit\mathbfscit{#1}}
\@onlypreamble\mathfontcommands
```

# 8 Miscellaneous

We begin this section with the user-level macros that provide information for Lua-based font adjustments. We define a macro `\M@check@int` to determine if `#1` is a nonnegative integer and set the switch `\M@arg@good` accordingly. Checking happens in three stages inside a group. Immediately after the group, we will check whether `\@tempb` is `\@empty`, so to force the switch to be false,

we sometimes set `\@tempb` to `\@nnil` inside the group. The first stage is a check whether `#1` is empty.

```
\ifM@adjust@font
\M@check@int  \def\M@check@int#1{%
    \begingroup
      \def\@tempa{#1}%
      \ifx\@tempa\@empty
        \let\@tempb\@nnil
```

If `#1` is not empty, we check whether it is a nonnegative integer. If `#1` is a nonnegative integer, the entirety of `0#1` becomes the value of `\count@`, and `\@tempb` will end up empty. Otherwise, `\@tempb` will be nonempty. The use of `\afterassignment` here is inspired by `\@defaultunits` from the kernel.

```
      \else
        \def\@tempa##1\relax\@nil{\def\@tempb{##1}}%
        \afterassignment\@tempa
        \count@=0#1\relax\@nil
```

If `\@tempb` is nonempty, we handle the case where `#1` is an octal or hexadecimal integer. We use `\if` to check whether the first character of `#1` is `'` or `"`, and if yes, we replace it with `'0` or `"0` and repeat the same check with `\@tempb`. The `\relax` prevents `\if` from scanning past `#1` if `#1` expands to something empty. The `\remove@to@nnil` gobbles everything in `#1` (except the `'` or `"`, which is eaten by `\if`) when the test is successful, and it gobbles the `\@nnil` after the `\else` when the test is unsuccessful.

```
        \ifx\@tempb\@empty
        \else
          \expandafter\remove@to@nnil\if'#1\relax
            \@nnil
            \afterassignment\@tempa
            \count@'0\expandafter\@gobble
              \expanded{#1}\relax\@nil
          \else
            \@nnil
            \expandafter\remove@to@nnil\if"#1\relax
              \@nnil
              \afterassignment\@tempa
              \count@"0\expandafter\@gobble
                \expanded{#1}\relax\@nil
```

If `#1` is neither octal nor hexadecimal, we check whether it starts with `\numexpr`. This case is good, so we set `\@tempb` to `\@empty`.

```
            \else
```

```
            \@nnil
            \expandafter\remove@to@nnil\ifx\numexpr#1\@nnil
              \let\@tempb\@empty
```

The last possibility is if `#1` has the form `` `⟨*character*⟩ ``. To allow for the possibility of, for example, `` `\% ``, we check whether `#1` begins with `` ` `` and contains two tokens. This is not foolproof, and sufficiently bad input, such as `` `\relax ``, will break this macro. (So don't do that.) Checking the catcode of the second token in `#1` doesn't seem worth the effort. If `#1` is a single `` ` `` character, then `\@gobbletwo` will eat `#1` and `\@nnil`, so the `\ifx` will compare `\@nnil` and `\relax`.

```
            \else
              \@nnil
              \expandafter\remove@to@nnil\if`#1\relax
                \@nnil
                \expandafter\ifx\expandafter\@nnil
                  \@gobbletwo#1\@nnil\relax
                  \let\@tempb\@empty
                \else
                  \let\@tempb\@nnil
                \fi
              \else
                \@nnil
                \let\@tempb\@nnil
              \fi
            \fi
          \fi
        \fi
      \fi
      \expandafter
    \endgroup
    \ifx\@tempb\@empty
      \M@arg@goodtrue
    \else
      \M@arg@goodfalse
    \fi}
```

Making the `\rulethicknessfactor`, etc. counts accessible to the user means that we don't need `\RuleThicknessFactor` and friends anymore, but we keep them for backwards compatibility and convenience. Each of these commands uses `\M@check@int` to check its argument, then calls the appropriate other commands.

```
\RuleThicknessFact 1322  \protected\def\RuleThicknessFactor#1{%
                   1323    \M@check@int{#1}%
                   1324    \ifM@arg@good
                   1325      \rulethicknessfactor=#1\relax
                   1326    \else
                   1327      \M@BadIntegerError\RuleThicknessFactor{\detokenize{#1}}%
                   1328    \fi}
\SurdHorizontalFac 1329  \protected\def\SurdHorizontalFactor#1{%
                   1330    \M@check@int{#1}%
                   1331    \ifM@arg@good
                   1332      \hsurdfactor=#1\relax
                   1333    \else
                   1334      \M@BadIntegerError\SurdHorizontalFactor{\detokenize{#1}}%
                   1335    \fi}
\SurdVerticalFacto 1336  \protected\def\SurdVerticalFactor#1{%
                   1337    \M@check@int{#1}%
                   1338    \ifM@arg@good
                   1339      \vsurdfactor=#1\relax
                   1340    \else
                   1341      \M@BadIntegerError\SurdVerticalFactor{\detokenize{#1}}%
                   1342    \fi}
```

For the integral italic factor, we input the information to `\charmline`

```
\IntegralItalicFac 1343  \protected\def\IntegralItalicFactor#1{%
                   1344    \M@check@int{#1}%
                   1345    \ifM@arg@good
                   1346      \charmline{0x222B * * * * * * * * * *
                   1347                        * * * * * * * * * *
                   1348                        * * * * * * * * * *
                   1349                                        * * #1}%
                   1350    \else
                   1351      \M@BadIntegerError\IntegralItalicFactor{\detokenize{#1}}%
                   1352    \fi}
```

If automatic font adjustments are disabled, we should also disable the related user-level commands. In this case, each of the font-adjustment macros expands to raise an `\M@NoFontAdjustError` and gobble its argument.

```
                   1353 \else
                   1354  \@tfor\@i:=\RuleThicknessFactor\IntegralItalicFactor
                   1355    \SurdHorizontalFactor\SurdVerticalFactor\charmline
                   1356    \charmfile\CharmLine\CharmFile\CharmInfo\CharmType
                   1357      \do{%
                   1358        \protected\expandafter\edef\@i{%
```

```
1359          \noexpand\M@NoFontAdjustError\expandafter\noexpand\@i
1360          \noexpand\@gobble}}
\charminfo 1361  \protected\def\charminfo{\M@NoFontAdjustError\charminfo
1362    \begingroup
1363      \afterassignment\endgroup
1364      \count@}
\charmtype 1365  \protected\def\charmtype{\M@NoFontAdjustError\charmtype
1366    \begingroup
1367      \afterassignment\endgroup
1368      \count@}
1369 \fi
```

These commands should appear in the preamble only.

```
1370 \@onlypreamble\charmline
1371 \@onlypreamble\charmfile
1372 \@onlypreamble\CharmLine
1373 \@onlypreamble\CharmFile
1374 \@onlypreamble\RuleThicknessFactor
1375 \@onlypreamble\IntegralItalicFactor
1376 \@onlypreamble\SurdHorizontalFactor
1377 \@onlypreamble\SurdVerticalFactor
```

We use the next three macros in defining `\simeq` and `\cong`. The construction is clunky and needs the intermediate macro `\st@ck@fl@trel` because `\mathchoice` is a bit of an odd macro. It feels like it should be expandable, but it isn't. Instead, it fully typesets each of its four arguments and then takes the one corresponding to the correct style. This is due to fundamental aspects of how TEX processes math-mode material.

```
1378 \protected\gdef\clap#1{\hb@xt@\z@{\hss#1\hss}}
\stack@flatrel 1379 \protected\def\stack@flatrel#1#2{\expandafter
1380   \st@ck@fl@trel\expandafter#1\@firstofone#2}
\st@ck@fl@trel 1381 \protected\def\st@ck@fl@trel#1#2#3{%
1382   {\setbox0\hbox{$#1#2\m@th$}% contains \mathrel symbol
1383   \setbox1\hbox{$#1#3\m@th$}% gets raised over \box0
1384   \if\wd0>\wd1\relax
1385     \hb@xt@\wd0{%
1386       \hfil
1387       \clap{\raise0.7\ht0\box1}%
1388       \clap{\box0}\hfil}%
1389   \else
1390     \hb@xt@\wd1{%
1391       \hfil
1392       \clap{\raise0.7\ht0\box1}%
```

```
      \clap{\box0}\hfil}%
  \fi}}
```

Some fonts do not contain characters that mathfont can declare as math symbols. We want to make sure that if this happens, TEX prints a message in the `log` file and terminal.

```
\ifnum\tracinglostchars<\tw@
  \tracinglostchars\tw@
\fi
```

We `\typeout` a message about local font-change commands.

```
\AtBeginDocument{%
  \ifcase\M@num@localfonts
  \or
    \def\@tempa#1#2#3\@nil{#2}
    \wlog{}
    \typeout{:: mathfont :: Using
      \expandafter\@tempa\M@localfonts\@nil\space
      for local font changes.}
  \else
    \wlog{}
    \typeout{:: mathfont :: Using \the\M@num@localfonts\space
      fonts for local font changes.}
  \fi}
```

A helper macro for printing messages `\AtBeginDocument`. This macro adds characters from `#1` into `\@tempa` until the length of `\@tempa`, as measured by `\count@` becomes `#2` − 2. If `\@tempa` has too many characters to fit, `\M@addtab@count@tempa` instead ends `\@tempa` with an ellipsis. Then the macro appends spaces to the end of `\@tempa` until it is `#2` characters long, again measured using `\count@`.

```
\def\M@addtab@count@tempa#1#2{%
  \let\@tempb\@empty
  \@tempcntb\z@
  \@tfor\@i:=#1\do{%
    \ifnum\count@=\numexpr#2 - 5\relax
      \edef\@tempb{\@tempb\@i}
      \advance\@tempcntb\@ne
    \else
      \edef\@tempa{\@tempa\@i}
      \advance\count@\@ne
    \fi}
  \ifnum\@tempcntb<4\relax
    \edef\@tempa{\@tempa\@tempb}
```

```
    \advance\count@\@tempcntb
  \else
    \edef\@tempa{\@tempa...}
    \advance\count@\thr@@
  \fi
  \@whilenum\count@<#2\do{%
    \edef\@tempa{\@tempa\space}
    \advance\count@\@ne}}
```

Write to the `log` file `\AtBeginDocument` all font changes carried out by mathfont. The command `\M@fontinfo@begin` accepts accepts a keyword as its `#1` argument and prints a message on the log file showing whether mathfont set a default font for that keyword and, if yes, the name, series, and shape for that font.

`\M@fontinfo@begin`

```
\def\M@fontinfo@begin#1{%
  \expandafter\ifx % next lines are two cs to be compared
      \csname ifM@#1\expandafter\endcsname
      \csname iftrue\endcsname
    \expanded{\noexpand\M@fontinfo@begin@{#1}
      \csname M@fontinfo@#1\endcsname}
  \else
    \bgroup
      \let\@tempa\@empty
      \count@\z@
      \M@addtab@count@tempa{#1}{35}
      \M@addtab@count@tempa{No\space change}{78}
      \wlog{\@tempa}
    \egroup
  \fi}
```

Helper macro that handles printing the message. The four arguments appear sequentially on the same line in the log file: `#1` is the keyword, `#2` is the font name, `#3` is the series, and `#4` is the shape. We allocate 18, 30, 15, and 15 characters respectively, for a total of 78 characters. The default width of the log file is 80 characters, so we should fit everything on one line that way.

`\M@fontinfo@begin@`

```
\def\M@fontinfo@begin@#1#2#3#4{%
  \bgroup
    \let\@tempa\@empty
    \count@\z@
    \M@addtab@count@tempa{#1}{18}
    \M@addtab@count@tempa{font:\space#2}{48}
    \M@addtab@count@tempa{series:\space#3}{63}
    \M@addtab@count@tempa{shape:\space#4}{78}
```

And print the message.

```
    \wlog{\@tempa}
  \egroup}
```

The macro `\M@localfonts@begin` does the same thing except for the local font-change commands.

`\M@localfonts@begi`
```
\let\M@localfonts@begin\M@fontinfo@begin@
```

And for symbol fonts declared.

`\M@symbolfonts@beg`
```
\let\M@symbolfonts@begin\M@fontinfo@begin@
```

The command for font families is different because we only have two pieces of information to display.

`\M@families@begin`
```
\def\M@families@begin#1{%
  \bgroup
    \let\@tempa\@empty
    \count@\z@
    \expanded{%
      \noexpand\M@addtab@count@tempa
        {\csname M@fontid@#1\endcsname}{5}
      \noexpand\M@addtab@count@tempa{#1}{78}}
    \wlog{\@tempa}
  \egroup}
```

Now print the messages. We start with the font families that mathfont uses. We don't need to store a list of font families in `\M@families` because we can simply increment the `fontid` count until we reach a value that is large enough to not have a corresponding font.

```
\AtBeginDocument{%
  \wlog{^^J------------------ Changes made by mathfont
  in the preamble ------------------}
  \wlog{}
  \wlog{**********************^^J%
        * Font families used *^^J%
        **********************}
  \ifx\M@families\@empty
    \wlog{No font families declared by mathfont.}
  \else
    \@for\@i:=\M@families\do{%
      \expandafter\M@families@begin\expandafter{\@i}}
  \fi
  \wlog{}
```

Same thing for symbol fonts.

```
  \wlog{************************^^J%
```

```
1484        * Symbol fonts declared *^^J%
1485        *************************}
1486  \ifx\M@symbolfonts\@empty
1487    \wlog{No symbol fonts declared by mathfont.}
1488  \else
1489    \@for\@i:=\M@symbolfonts\do{%
1490      \expandafter\M@symbolfonts@begin\@i}
1491  \fi
1492  \wlog{}
```

Character keywords.

```
1493  \wlog{************^^J%
1494        * Keywords *^^J%
1495        ************}
1496  \@for\@i:=\M@keys\do{%
1497    \expandafter\M@fontinfo@begin\expandafter{\@i}}
1498  \wlog{}
```

And information in the `log` file about local font-change commands.

```
1499  \wlog{*****************************^^J%
1500        * Local font-change commands *^^J%
1501        *****************************}
1502  \ifnum\M@num@localfonts=\z@
1503    \wlog{No local font change commands declared.}
1504  \else
1505    \@for\@j:=\M@localfonts\do{%
1506      \expandafter\M@localfonts@begin\@j}
1507  \fi
1508  \wlog{}
1509  \wlog{------------------------------- End of changes
1510        -------------------------------}
1511  \wlog{}}
```

Warn the user about possible problems with a multi-word optional package argument in XeTeX.

```
1512 \ifdefined\XeTeXrevision
1513  \ifM@font@loaded
1514    \AtEndOfPackage{%
1515    \PackageWarningNoLine{mathfont}
1516    {It looks like you specified a font\MessageBreak
1517    when you loaded mathfont. If you run\MessageBreak
1518    into problems with a font whose name\MessageBreak
1519    is multiple words, try using LuaLaTeX\MessageBreak
1520    or call \string\documentfont\space or
```

```
    \string\mathfont\MessageBreak
    manually}}
  \fi
\fi
```

If the user passed a font name to mathfont, we set it as the default `\AtEndOfPackage`.

```
\ifM@font@loaded
  \AtEndOfPackage{\documentfont\M@font@load}
\fi
```

# 9 Adjust Fonts: Setup

The next three sections implement Lua-based font adjustments and apply only if the user has enabled font adjustment. Most of the implementation happens through Lua code, but we need some TEX code in case the user wants to adjust character metric information. Here is a rough outline of what happens in the next three sections:

1. Initialize a Lua table that contains new metrics for certain characters specific to math mode, such as letters with wider bounding boxes and large operator symbols.
2. Provide an interface for the user to change this metric information.
3. Write functions that accept a fontdata object and (a) change top-level math specs to indicate that we have a math function; (b) alter characters according to our Lua table of new metric information; and (c) populate a MathConstants table for the font.
4. Create callbacks that call these functions. Put a wrapper around them, and insert the wrapper-function into `luaotfload.patch_font`.

Step 2 happens on the TEX side of things and is documented next, and everything else happens inside `\directlua`. On the Lua side of things, we store all the functions and character metric information in the table `mathfont`. With the exception of a handful of integers used to track encoding slots, every entry in `mathfont` is either a function or a subtable indexed by an ⟨*integer*⟩. The *integer* is a Unicode encoding number and indicates which Unicode character the subtable corresponds to. See tables 2 and 3 for a list of the functions in `mathfont` and the fields in character subtables. See section 10 for discussion of the callbacks for editing fontdata objects.

Changing the top-level `nomath` flag in a font object is easy. Creating a MathConstants table is complicated but largely self-contained. We take a

**Table 2: Fields of Character Subtables in `mathfont`**

| Field | Type | In `a` | In `e` | Used For |
|---|---|---|---|---|
| `type` | string | Yes | Yes | Type is `a` or `e` |
| `data_rm` | table | Yes | Yes | Information for upright font shapes |
| `data_it` | table | Yes | Yes | Information for italic font shapes |
| `num_variants` | integer | No | Yes | Number of large variants |
| `smash` | integer | No | Yes | Encoding slot for smashed character |
| `next` | table | No | Yes | Encoding slots for large variants |

few parameters that the user has set, define traditional TEX math parameters based on the essential parameters of the font, and assign their values to the corresponding entries in a MathConstants table. However, editing character metrics during font loading is convoluted with many moving parts. For every glyph that we want modify, we store character metric information for that glyph as a subtable in `mathfont`. The entries of the subtable describe how to scale the bounding box, scale the glyph itself, or determine math accent placement. For characters of type `a` ("alphabet"), we specify information to stretch the bounding box (not the glyph) horizontally, so we equivalently add extra space around the character. For type `e` ("extensible"), we stretch the bounding box and glyph, so we create an ensemble of scaled versions, which we use as a family of large variants.

Here's how to think about the dynamics of our approach. We use character metric information at three different times: pre-processing, interim processing, and post-processing. In pre-processing, which we implement in this section, we assemble initial character metric information into entries in `mathfont`. In other words, pre-processing means creating the initial `mathfont` subtables and happens during package loading. Interim processing means the user altering entries in `mathfont` and happens through `\charmline` and `\charmfile`. This can occur at any point in the pramble. In post-processing, which we implement in the next section, mathfont extracts information from the current state of the `mathfont` table and uses it to alter a fontdata object. Post-processing happens through the `luaotfload.patch_font` callback and occurs once at the point when TEX loads the font file. As a rule, LATEX does not like to load fonts before it uses them, so post-processing typically happens `\AtBeginDocument` in the case of the main text font or whenever the user calls a `\text`⟨*font keyword*⟩ command or enters math mode. This is also why you cannot adjust fonts that TEX loaded before mathfont.

We set `mathnolimitsmode` to 4 to make integral signs look nice. Or at least nicer than they would otherwise.

**Table 3: Functions in `mathfont`**

| Function | Argument(s) | Used For |
|---|---|---|
| `strint` | number | Format number as a string |
| `new_type_a` | `index`, data | Add type `a` entry to `mathfont` |
| `new_type_e` | `index`, data | Add type `e` entry to `mathfont` |
| `add_to_charm` | string of charm info | Add charm info to `mathfont` |
| `parse_charm` | string of charm info | Split string, validate inputs |
| `parse_num` | numeric string | Parse numeric value |
| `empty` | none | Does nothing |
| `glyph_info` | character subtable | Get height, width, depth, italic |
| `make_a_commands` | `index`, offset | Return virtual font commands |
| `make_a_table` | `index`, charm data, fontdata | Make new subtable for type `a` character |
| `make_e_commands` | `index`, scale factors | Return virtual font commands |
| `make_e_table` | `index`, charm data, fontdata | Make new subtable for type `e` character |
| `smash_glyph` | `index`, fontdata | Make table for smashed char |
| `utf_16BE` | integer | Return UTF-16BE format |
| `adjust_font` | fontdata | Call callbacks |
| `apply_charm_info` | fontdata | Change character metrics in font |
| `math_constants` | fontdata | Create MathConstants table |
| `set_nomath_false` | fontdata | Set `nomath` (top-level flag in the font) to `false` |
| `get_font_name` | fontdata | Return font name |
| `info` | string | Write message in the `log` file |

```
\ifM@adjust@font
\mathnolimitsmode=4\relax
```

We need some error messages. We change the catcode of \ to 12 in order to use it freely as a Lua escape character. We change ~ to catcode 0 to define the macros.

```
\bgroup
  \catcode`\~=0
  ~catcode`~\=12
  ~@firstofone{%
~egroup
```

```
~let~@tempa~%
~let~%~@percentchar
~def~M@empty@ssert{"\n\n%
  Package mathfont error: Empty charm information.\n\n%
  Your argument for \\charmline is empty, or a\n%
  line in your \\charmfile is blank. Make sure\n%
  your calls to \\charmline and all lines in\n%
  your \\charmfile contain integers, floats,\n%
  and asterisks separated by commas or spaces.\n"}
~def~M@missing@ssert{"\n\n%
  Package mathfont error: Missing charm entries.\n\n%
  I'm having trouble with a character metric.\n%
  Your \\charmline or \\charmfile contains\n%
  \"".. temp_string .. ",\"\n%
  which looks to me like you provided an index\n%
  without any commas or spaces to specify the\n%
  numbers for charm values. Make sure that you\n%
  use commas or spaces to separate each entry\n%
  in your charm information.\n"}
~def~M@number@ssert{"\n\n%
  Package mathfont error: Nonnumeric charm value.\n\n%
  I'm having trouble with a character metric.\n%
  Your \\charmline or \\charmfile contains \""
  .. s .. ",\"\n%
  which is not a number. Make sure that your\n%
  charm information is all integers, floats,\n%
  and asterisks separated by commas or spaces.\n"}
~def~M@index@ssert{"\n\n%
  Package mathfont error: Invalid Unicode index.\n\n%
  The Unicode index \""
  .. temp_string .. "\" is invalid. Make sure\n%
  that the first entry in your \\charmline and in each\n%
  line of your \\charmfile is an integer between 0 and\n%
  1,114,111 (0x10FFFF), possibly with an exclamation\n%
  point or question mark.\n"}
~def~M@bound@ssert{"\n\n%
  Package mathfont error: Exceeded Unicode table.\n\n%
  You asked me to do something with an encoding slot\n%
  number that exceeds the number of slots in the\n%
  Unicode table. You are probably seeing this error\n%
  because you declared too many type e characters.\n"}
```

We previously defined \% to contain a single % with catcode 12, so when we

put `\%` in an `\edef`, it becomes easy to get `%`'s inside the definition of our next two macros. This is important because it lets us use `string.format` to create informative error and warning messages using straightforward syntax. To insert the Unicode character in the messages, we use `utf8.char` function, which is the Lua code for producing arbitrary Unicode characters.

```
~edef~M@entries@ssert{string.format("\n\n%
  Package mathfont error: Charm values too short.\n\n%
  Your charm information for U+~%X ~%s (index ~%d)\n%
  needs more entries. Right now you have ~%d\n%
  entries (besides the index), but you need at\n%
  least ~%d. If you aren't sure what to do, try\n%
  adding asterisks to your \\charmline or line in\n%
  your \\charmfile. See the user guide for more\n%
  information.\n",
  index, utf8.char(index), index,
  number_of_entries, entries_needed)}
~edef~M@entries@warning{string.format("\n\n%
  Package mathfont warning: Charm values too long.\n\n%
  Your charm information for U+~%X ~%s (index ~%d)\n%
  has more entries than it needs. Right now you\n%
  have ~%d entries (besides the index), but you\n%
  only need ~%d. This isn't a problem per se\n%
  because I can easily ignore the extra numbers,\n%
  but it may indicate confusion about Unicode\n%
  characters and charm values. See the user\n%
  guide for more information.\n\n",
  index, utf8.char(index), index,
  number_of_entries, entries_needed)}
```

Error message if the user tries to adjust the left side of the bounding box on a virtual character.

```
~edef~M@virtual@ssert{string.format("\n\n%
  Package mathfont error: Can't adjust left side\n%
  of the bounding box on a virtual character.\n\n%
  Your charm information for U+~%X ~%s (index ~%d)\n%
  instructs me to change the left side of the bounding\n%
  box around this character. However, in the font\n%
  ~%s,\n%
  that character is a virtual character, and I'm not\n%
  programmed to change the left side of the bounding\n%
  box on a virtual character. To resolve this error,\n%
  try including\n%
```

```
  \\charmline{~%d 0 * * *}\n%
  at the end of your document preamble.\n\n",
  index, utf8.char(index), index,
  mathfont.get_font_name(fontdata), index)}}
\let\%\@tempa
```

The user inputs charm information at the TEX level. We define the macros `\charmline` that interfaces with `mathfont:add_to_charm` directly and `\charmfile` that reads lines from a file and individually feeds them to `\charmline`. The macros `\charminfo` and `\charmtype` print information from `mathfont` about the charm information currently in memory for certain characters.

```
\newluafunction\addtocharm@
\newluafunction\charminfo@
\newluafunction\charmtype@
\directlua{%
  local t = lua.get_functions_table()
  t[\number\addtocharm@] = function()
    mathfont:add_to_charm(token.scan_string())
  end
```

We also define the Lua function `\charminfo@` for use in `\charminfo`. This function scans the following integer and prints the charm information for the Unicode character whose index is that integer.

```
  t[\number\charminfo@] = function()
    local temp = token.scan_int()
    temp = mathfont[temp]
```

If `mathfont` contains an entry for index `temp`, gather the charm information from that entry. We will print to TEX a string in the same format as the argument of `\charmline`. We loop through `data_rm`, and on each iteration, we add that entry to a temporary string and then add italic charm information if it is different.

```
    if temp then
      local temp_str = ""
      local temp_rm = {}
      local temp_it = {}
```

We have to possibilities depending on the type of the input. For type `a` characters, all entries in `data_rm` and `data_it` are integers, and we can loop the list without issue. For type `e` characters, `data_rm` and `data_it` contain 15 two-entry subtables and three integers, so we have to flatten the list before we use it.

```
      for k,v in pairs(temp.data_rm) do
```

```
        if type(v) == "number" then
          temp_rm[k] = v * 1000
          temp_it[k] = temp.data_it[k] * 1000
        elseif type(v) == "table" then
          temp_rm[2*k-1] = v[1] * 1000
          temp_rm[2*k]   = v[2] * 1000
          temp_it[2*k-1] = temp.data_it[k][1] * 1000
          temp_it[2*k]   = temp.data_it[k][2] * 1000
        end
      end
```

Now we loop through `temp_rm` and add the contents to `temp_str`.

```
      for k,v in pairs(temp_rm) do
        if temp_str \noexpand~= "" then
          temp_str = temp_str .. " "
        end
        temp_str = temp_str .. mathfont.strint(v)
```

Now check the corresponding charm entry for italic fonts and add that if different from `v`.

```
        if temp_it[k] \noexpand~= v then
          temp_str = temp_str .. "/" ..
            mathfont.strint(temp_it[k])
        end
      end
```

Now print the result.

```
      tex.print(temp_str)
    else
      tex.print("none")
    end
  end
```

Same thing for `\charmtype@`.

```
  t[\number\charmtype@] = function()
    local temp = token.scan_int()
    temp = mathfont[temp]
    if temp then
      tex.print(temp.type)
    else
      tex.print("none")
    end
  end}
\charmline \protected\def\charmline{\luafunction\addtocharm@}
```

The argument of `\charmfile` should be a valid filename, and we open it in `\M@Charm`. The macro processes each line of the file as a piece of charm information.

```
\charmfile 1666 \protected\def\charmfile#1{%
           1667   \IfFileExists{#1}{%
           1668     \begingroup
           1669     \endlinechar\m@ne
           1670     \immediate\openin\M@Charm{#1}
```

The macro `\next` reads a line into `#1`, feeds it to `\charmline`, and calls itself if the file has more lines. TEX adds an extra line to the end of files it reads (why??), so we check whether the current line is empty before feeding it to `\charmline`. (The last line of the file is empty because we set `\endlinechar` to −1. Otherwise, the file would have a spurious `\par` at the end.)

```
           1671     \def\next{%
           1672       \read\M@Charm to \@tempa
           1673       \ifx\@tempa\@empty
           1674       \else
           1675         \charmline\@tempa
           1676       \fi
           1677       \ifeof\M@Charm\else % if file has more lines?
           1678         \expandafter\next
           1679       \fi}
```

Call `\next`, close the file, and end the group.

```
           1680     \next
           1681     \immediate\closein\M@Charm
           1682     \endgroup}
```

If the file does not exist, raise an error.

```
           1683   {\M@NoCharmFileError{\detokenize{#1}}}}
```

Alternative names.

```
\CharmLine 1684 \let\CharmLine\charmline
\CharmFile 1685 \let\CharmFile\charmfile
```

Now the macros `\charminfo` and `\charmtype`. The structure is a bit different because we want them to be fully expandable. The macros don't have an argument. Instead, they call the appropriate `\luafunction`, which scans the next integer and processes it.

```
\charminfo 1686 \def\charminfo{\luafunction\charminfo@}
\charmtype 1687 \def\charmtype{\luafunction\charmtype@}
```

The wrapped versions are more user-friendly. For `\CharmInfo`, we first check whether `mathfont` contains an entry with index `#1`.

```
\CharmInfo 1688 \protected\def\CharmInfo#1{%
           1689   \M@check@int{#1}%
           1690   \ifM@arg@good
           1691     \begingroup                % \begingroup
           1692       \edef\@tempa{\charminfo#1}%
           1693       \def\@tempb{none}%
           1694       \edef\@tempc{\number#1}%
```

Now the actual check. If there is no entry, we print a message saying so.

```
           1695       \ifx\@tempa\@tempb
           1696         \expandafter\endgroup  % first branch \endgroup
           1697         \expanded{\showtokens{no charm info assigned to
           1698           index \@tempc}}%
```

If yes, we print the charm information to the terminal.

```
           1699       \else
           1700         \expandafter\endgroup  % second branch \endgroup
           1701         \expanded{\showtokens{index \@tempc\space
           1702           has charm info: \@tempa}}%
           1703       \fi
           1704   \else
           1705     \M@BadIntegerError\CharmInfo{\detokenize{#1}}%
           1706   \fi}
```

Now for `\CharmType`.

```
\CharmType 1707 \protected\def\CharmType#1{%
           1708   \M@check@int{#1}%
           1709   \ifM@arg@good
           1710     \begingroup
           1711       \edef\@tempa{\number#1}%
           1712     \expandafter\endgroup
           1713     \expanded{\showtokens{index \@tempa\space has type
           1714       \charmtype\numexpr\@tempa\relax}}%
           1715   \else
           1716     \M@BadIntegerError\CharmType{\detokenize{#1}}%
           1717   \fi}
```

This concludes the TEX-based portion of font adjustments. The rest of this section and the next two sections are the Lua code that adapts a text font for math mode. First, we create the `mathfont` table. We use `mathfont.encoding_slot` in `new_type_e` to keep track of the encoding slots where we will artificially add large versions of type `e` characters during loading.

```
1718 \directlua{
1719 mathfont = {}
```

```
mathfont.extra_chars   = 0xFA000
mathfont.encoding_slot = 0xFA010
mathfont.fakel = mathfont.extra_chars + 4
mathfont.faker = mathfont.extra_chars + 5
mathfont.fakell = mathfont.extra_chars + 6
mathfont.fakerr = mathfont.extra_chars + 7
```

Helper function for use in `\charminfo@`.

```
function mathfont.strint(i)
  if i == (i // 1) then
    return string.format("\@percentchar d", i)
  else
    return tostring(i)
  end
end
```

Each character whose metrics we want to change will have one of two types: `a` for alphabet or `e` for extensible. We begin with type `a`. The `index` is the base-10 Unicode encoding value of the character that we will later modify. The `data` arguments are tables with 4 entries that store sizing information and information regarding accent placement. We divide the information by 1000 as is standard in TEX.

```
function mathfont:new_type_a(index, data_rm, data_it)
  self[index] = {}
  self[index].type = "a"
  self[index].data_rm = {}
  self[index].data_it = {}
  for i = 1, 4, 1 do
    self[index].data_rm[i] = data_rm[i] / 1000
    self[index].data_it[i] = data_it[i] / 1000
  end
end
```

Initializing type `e` characters is more complicated. The `index` and `data` arguments are the same as in the type `a` case, and we process them similarly. The entries in `mathfont` for type `e` characters contain additional information. The `smash` value is a Unicode slot where we store a smashed version of the glyph with no height, depth, or width, which we need to scale the glyph correctly. The `num_variants` attribute is the number of slots in `next`, which we shorten to `v` for notational convenience.

```
function mathfont:new_type_e(index, data_rm, data_it)
  self[index] = {}
  self[index].type = "e"
```

```
1746  self[index].smash = self.encoding_slot
1747  local v = (\string# data_rm - 3) / 2
1748  self[index].num_variants = v
```

Check that the user hasn't exhausted the Unicode table. (Unlikely, but you never know.)

```
1749  if self.encoding_slot + v + 1 > 0x10FFFF then
1750    error(\M@bound@ssert)
1751  end
```

Now make lists that store encoding slots and scale factors. We have the following lists:

- `next`: list of encoding slots
- `data_rm`: scale factors for upright font shapes
- `data_it`: scale factors for italic font shapes

Start with making blank lists in the subtable in `mathfont`.

```
1752  self[index].next = {}
1753  self[index].data_rm = {}
1754  self[index].data_it = {}
```

We assemble these lists in a single loop, and they all have `v` elements. For `next`, we append consecutive integers to the list. For the scale factors, we expect `data_rm` and `data_it` to have $2\texttt{v}+3$ entries, which we consider in pairs. The $i$th pair (i.e. entries $2i-1$ and $2i$ of `data_rm` or `data_it`) encodes the horizontal and vertical scale factors for the $i$th large variant, and we add those scale factors as two-element sublists to the new lists on the $i$th iteration.

```
1755  for i = 1, v, 1 do
1756    self[index].next[i] = self.encoding_slot + i
1757    self[index].data_rm[i] = {data_rm[2*i-1] / 1000,
1758                              data_rm[2*i]   / 1000}
1759    self[index].data_it[i] = {data_it[2*i-1] / 1000,
1760                              data_it[2*i]   / 1000}
1761  end
```

The final entries of `data_rm` and `data_it` contain information about accent placement and italic correction. We add those values to the subtable as well.

```
1762  for i = 1, 3, 1 do
1763    self[index].data_rm[2*v+i] = data_rm[2*v+i] / 1000
1764    self[index].data_it[2*v+i] = data_it[2*v+i] / 1000
1765  end
```

Finally, update the `encoding_slot`.

```
1766  self.encoding_slot = self.encoding_slot + v + 1
1767 end
```

Interim processing. We let the user edit resizing and accent information for the characters in `mathfont`. The main editing function is `mathfont:add_to_charm`, which incorporates the user's information into the tables already in `mathfont`. It expects a single string of integers, floats, or asterisks separated by spaces or commas, and it immediately passes the argument to `parse_charm`, which processes it into tables that we incorporate into subtables of `mathfont`. We begin with a helper function to parse a (numeric) string. If this function returns a number, the number is properly scaled (divided by 1000).

```
function mathfont.parse_num(s)
  if s == "*" then
    return s
  else
    local temp = tonumber(s)
    if temp then
      return temp / 1000
    else
      error(\M@number@ssert)
    end
  end
end
```

Now `parse_charm`. We begin by setting up tables to store the parsed string contents. We store the Unicode index value in `index`.

```
function mathfont.parse_charm(charm_input)
  local index = 0
  local charm_string = charm_input
  local temp_string = ""
```

Some preprocessing before we parse the string. Specifically, we

1. Get rid of any duplicate spaces
2. Remove any leading or trailing space, if present
3. Remove any spaces around slashes or commas
4. Replace any remaining spaces with commas

After completion of the replacements, we should have a new string with same numeric/override information as the original `charm_input` except without any spaces and all (pairs of) entries separated by commas. Step 1: duplicate spaces.

```
  while string.find(charm_string, "\space\space") do
    charm_string = string.gsub(charm_string, "\space\space", " ")
  end
```

Step 2: leading/trailing spaces.

```
  if string.sub(charm_string, 1, 1) == " " then
```

```
    charm_string = string.sub(charm_string, 2)
  end
  if string.sub(charm_string, -1) == " " then
    charm_string = string.sub(charm_string, 1, -2)
  end
```

Step 3: space around punctuation. We don't replace `!␣` or `?␣` because the space in those cases could be separating two charm entries.

```
  charm_string = string.gsub(charm_string, ", ", ",")
  charm_string = string.gsub(charm_string, " ,", ",")
  charm_string = string.gsub(charm_string, "/ ", "/")
  charm_string = string.gsub(charm_string, " /", "/")
```

Step 4: add commas.

```
  charm_string = string.gsub(charm_string, " ", ",")
```

Check that `charm_string` is not empty.

```
  if charm_string == "" then
    error(\M@empty@ssert)
  end
```

Check that `charm_string` contains at least one comma. If it does not, we raise an error.

```
  if not string.find(charm_string, ",") then
    temp_string = charm_input
    error(\M@missing@ssert)
  end
```

We're ready to parse the entries. We remove the first entry manually since it is the index and has different formatting possibilities from the other entries.

```
  local sep = string.find(charm_string, ",")
  temp_string = string.sub(charm_string, 1, sep-1)
```

Now check that the index is a (valid) number. Handle the case of possible asterisk at the end of `temp_string`.

```
  local exc
  if string.sub(temp_string, -1, -1) == "!" then
    index = tonumber(string.sub(temp_string, 1, -2))
    exc = 1
  elseif string.sub(temp_string, -1, -1) == "?" then
    index = tonumber(string.sub(temp_string, 1, -2))
    exc = 2
  else
    index = tonumber(temp_string)
    exc = 0
  end
```

```
1818  if index then
1819    assert(index == index // 1 and
1820           index >= 0          and
1821           index <= 1114111,   \M@index@ssert)
1822  else
1823    error(\M@index@ssert)
1824  end
1825  charm_string = string.sub(charm_string, sep+1)
```

Create the lists that we will use to store the information from `charm_string`.

```
1826  local split_string_rm  = {}
1827  local split_string_it  = {}
```

We loop through `charm_string` as long as it contains characters. At each iteration, we store the location of the first comma in `sep`. We remove the portion of `charm_string` preceding the first comma and store it in `temp_string`, and we save the remaining portion of `charm_string` for processing on the next iteration of the loop. We use `i` as a dummy variable to track loop iterations.

```
1828  local i = 1
1829  while charm_string do
1830    sep = string.find(charm_string, ",")
1831    if sep then
1832      temp_string = string.sub(charm_string, 1, sep-1)
1833      charm_string = string.sub(charm_string, sep+1)
```

If the current value of `charm_string` does not contain a comma, then it must be the last portion, and we set `charm_string` to `nil`.

```
1834    else
1835      temp_string = charm_string
1836      charm_string = nil
1837    end
```

First check whether `temp_string` contains a `/` character. If yes, we have two values to process, and if not, we have one.

```
1838    sep = string.find(temp_string, "/")
1839    if sep then
```

The information for upright font shapes comes from the first portion of `temp_string`.

```
1840      split_string_rm[i] = mathfont.parse_num(
1841        string.sub(temp_string, 1, sep-1))
```

Information for italic shapes comes from the latter portion of `temp_string`.

```
1842      split_string_it[i] = mathfont.parse_num(
1843        string.sub(temp_string, sep+1))
```

For the case without a `/`, the same information goes in both sets of lists.

```
    else
      local temp = mathfont.parse_num(temp_string)
      split_string_rm[i]  = temp
      split_string_it[i]  = temp
    end
```

Increment `i`, end the loop, and return the charm information.

```
    i = i + 1
  end
  return {index, exc, split_string_rm, split_string_it}
end
```

We feed the user's charm information directly to `mathfont:add_to_charm`, which processes the information and stores it in `mathfont`. It first calls `parse_charm` to parse the input and then modifies `mathfont` accordingly. After being parsed, the user's input lives in `charm_rm` and `charm_it`. The `index` is the Unicode value of the character whose information we want to modify, and the `number_of_entries` is the length of `charm_metrics`.

```
function mathfont:add_to_charm(charm_string)
  local temp = self.parse_charm(charm_string)
  local index       = temp[1]
  local force_type  = temp[2]
  local data_rm     = temp[3]
  local data_it     = temp[4]
  local number_of_entries = \string# data_rm
```

If `mathfont` does not already have an entry for the Unicode character `index`, we create an entry with type `a` or `e` depending on the value of `force_type`.

```
  if not self[index] then
    mathfont.info(string.format("Setting up charm entries for
      U+\@percentchar X \@percentchar s (index \@percentchar d)",
      index, utf8.char(index), index))
    if force_type == 1 then
      temp = {}
      for i = 1, 30, 1 do
        temp[i] = 1000
      end
      for i = 31, 33, 1 do
        temp[i] = 0
      end
      self:new_type_e(index, temp, temp)
    else
      self:new_type_a(index, {0, 0, 0, 0}, {0, 0, 0, 0})
```

```
    end
  else
```

If `mathfont` does already have an entry for the character and `force_type` is positive, check whether the type of the entry in `mathfont` matches the value of `force_type`. If not, change the type by resetting the entry in `mathfont`. First, the case where the character has type `a`, and we want to change it to type `e`. We save the top and bottom accent values to use in setting up the new charm information.

```
    if (self[index].type == "a") and (force_type == 1) then
      local temp_rm = {}
      local temp_it = {}
      for i = 1, 30, 1 do
        temp_rm[i] = 1000
        temp_it[i] = 1000
      end
      temp_rm[31] = self[index].data_rm[3]
      temp_rm[32] = self[index].data_rm[4]
      temp_rm[33] = 0
      temp_it[31] = self[index].data_it[3]
      temp_it[32] = self[index].data_it[4]
      temp_it[33] = 0
      self:new_type_e(index, temp_rm, temp_it)
```

The process to convert to type `a` is simpler but fairly similar.

```
    elseif (self[index].type == "e") and (force_type == 2) then
      self:new_type_a(index, {0, 0,
                              self[index].data_rm[31],
                              self[index].data_rm[32]},
                             {0, 0,
                              self[index].data_it[31],
                              self[index].data_it[32]})
    end
  end
```

Handling the user's input depends on the type of entry `index`. The basic procedure is to first check the number of inputs, and if the user provided enough entries, we update each entries in the `mathfont` subtable. For every asterisk, we leave the corresponding subtable entries unaltered. For type `a`, we need four entries besides the `index`. The first two determine the left and right offset, and the last two determine accent placement.

```
  if self[index].type == "a" then
    local entries_needed = 4
```

Check number of entries. If it is too small, we issue an error, and if it is too large, we print a warning.

```
    if number_of_entries < entries_needed then
      error(\M@entries@ssert)
    elseif number_of_entries > entries_needed then
      texio.write_nl(\M@entries@warning)
    end
```

Now update the table entries. The data outputs from `parse_charm` have been properly scaled (divided by 1000), so we don't have to worry about rescaling in this function.

```
    for i = 1, 4, 1 do
      if data_rm[i] \noexpand~= "*" then
        self[index].data_rm[i] = data_rm[i]
      end
      if data_it[i] \noexpand~= "*" then
        self[index].data_it[i] = data_it[i]
      end
    end
```

Now do type `e`. The number of entries in the `data` lists must be at least $2 *$ `tot_variants` $+ 3$. We loop through the information and, for each $i$th pair of charm values, set those numbers to be the horizontal and vertical stretch information for the $i$th variant.

```
  elseif self[index].type == "e" then
    local tot_variants = self[index].num_variants
    local entries_needed = 2 * tot_variants + 3
```

Again check number of entries.

```
    if number_of_entries < entries_needed then
      error(\M@entries@ssert)
    elseif number_of_entries > entries_needed then
      texio.write_nl(\M@entries@warning)
    end
```

Now store the charm information. Again, we scaled the `data_rm` and `data_it` values in `parse_charm`, so we don't have to divide by 1000 here.

```
    for i = 1, tot_variants, 1 do
      if data_rm[2*i-1] \noexpand~= "*" then
        self[index].data_rm[i][1] = data_rm[2*i-1]
      end
      if data_rm[2*i] \noexpand~= "*" then
        self[index].data_rm[i][2] = data_rm[2*i]
      end
```

```
      if data_it[2*i-1] \noexpand~= "*" then
        self[index].data_it[i][1] = data_it[2*i-1]
      end
      if data_it[2*i] \noexpand~= "*" then
        self[index].data_it[i][2] = data_it[2*i]
      end
    end
```

The final entries for type `e` are the accent and italic correction information.

```
    for i = tot_variants + 1, tot_variants + 3, 1 do
      if data_rm[i] \noexpand~= "*" then
        self[index].data_rm[i] = data_rm[i]
      end
      if data_it[i] \noexpand~= "*" then
        self[index].data_it[i] = data_it[i]
      end
    end
  end
end
```

We end this section with three general-purpose Lua functions. The first function, `utf_16BE`, accepts a nonnegative integer and returns its representation in UTF-16 big-endian format. Let $x$ be a nonnegative integer at most 0x10FFFF. Here are the steps to convert $x$ to its big-endian representation:

1. If $x \leq$ 0xFFFF, keep $x$ as is. Its representation is a single four-digit hexadecimal number.
2. If $x \geq$ 0x10000, we represent $x$ as two four-digit hexadecimal numbers. First, subtract 0x10000 from $x$, and the result $y$ is a number between 0 and 0xFFFFF.
3. Equivalently, we can think of $y$ as a twenty-digit binary number. (Five hexadecimal digits store the same information as twenty binary digits.)
4. Take the 10 left-most digits of $y$ (integer divide by $2^{10} = 1024$), and add them to 0xD800. The result $z_1$ is the first hexadecimal number.
5. Take the 10 right-most digits of $y$ (remainder after dividing by $2^{10} = 1024$), and add them to 0xDC00. The result $z_2$ is the second hexadecimal number.
6. The big-endian representation of $x$ is the string $z_1 z_2$.

The purpose of big-endian representation is to encode Unicode characters beyond U+FFFF while still using four-digit hexadecimal numbers.

```
function mathfont.utf_16BE(integer)
  if integer > 0x10FFFF then
```

```
    error(\M@bound@ssert)
  end
  local temp = string.format("\@percentchar X", integer)
  if \string# temp <= 4 then
    while \string# temp < 4 do
      temp = "0" .. temp
    end
  else
    temp = integer - 0x10000
    local left_bits = 0xD800 + temp // 1024
    local right_bits = 0xDC00 + temp \@percentchar 1024
    temp = string.format("\@percentchar X\@percentchar X",
      left_bits, right_bits)
  end
  return temp
end
```

The `glyph_info` function does exactly what it sounds like. It accepts a character table from a font and returns the width, height, depth, and italic correction values.

```
function mathfont.glyph_info(char)
  local glyph_width = char.width or 0
  local glyph_height = char.height or 0
  local glyph_depth = char.depth or 0
  local glyph_italic = char.italic or 0
  return glyph_width, glyph_height, glyph_depth, glyph_italic
end
```

The `smash_glyph` function returns a character table that will produce a smashed version of the Unicode character with value `index`. The character has no width, height, or depth, and we typeset the glyph virtually using a `char` command.

```
function mathfont.smash_glyph(index)
  local smash_table = {}
  smash_table.width = 0
  smash_table.height = 0
  smash_table.depth = 0
  smash_table.commands = {{"char", index}}
  return smash_table
end
```

An empty function that does nothing. Used later for creating callbacks.

```
function mathfont.empty(arg)
end
```

# 10 Adjust Fonts: Changes

This section contains the Lua functions that actually modify the font during loading. The three functions `set_nomath_false`, `math_constants`, and `apply_charm_info` do most of the heavy lifting, and we set them as the default behavior for three callbacks. In total, mathfont defines seven different callbacks and calls them inside the function `adjust_font`—see table 4 for a list. Each callback accepts a fontdata object as an argument and returns nothing. You can use these callbacks to change mathfont's default modifications or to modify a fontdata object before or after mathfont looks at it. Be aware that if you add a function to any of the `disable_nomath`, `add_math_constants`, or `fix_character_metrics` callbacks, LuaTEX will not call the default `mathfont` function associated with the callback anymore. In other words, do not mess with these three callbacks unless you are duplicating the functionality of the corresponding "Default Behavior" function from table 4.

We begin with the functions that create new character subtables for inclusion in a font object, and we think of these new subtables as modified versions of characters already present in a given font. The functions for assembling character tables take three arguments. The `index` argument is the Unicode index of the modified character. The `charm_data` argument is the subtable in `mathfont` of charm information that corresponds to `index`, and the `fontdata` argument is a font object. For type `a`, we change the width of the bounding box and horizontal glyph positioning, and for type `e`, we scale the glyph to create large variants and change the italic correction. For both types, we modify accent placement. We add five categories of information into our new character tables: glyph dimensions, Unicode encoding bits, (possible) virtual font commands, accent placement dimensions, and math kerning.

The `:make_a_table` returns a character table for type `a` characters. We

**Table 4: Callbacks Created by mathfont**

| Callback Name | Default Behavior |
|---|---|
| `"mathfont.inspect_font"` | None |
| `"mathfont.pre_adjust"` | None |
| `"mathfont.disable_nomath"` | `mathfont.set_nomath_false` |
| `"mathfont.add_math_constants"` | `mathfont.math_constants` |
| `"mathfont.fix_character_metrics"` | `mathfont.apply_charm_info` |
| `"mathfont.post_adjust"` | None |
| `"mathfont.finishing_touches"` | None |

build up the subtable in the variable `a_table`, and we eventually return `a_table` at the end of the function. We let `char` be a shorthand for the subtable at `index` in `fontdata`, and `slant` is the font's `slant` parameter. In upright fonts, `slant` is generally 0, and in italic fonts, `slant` is generally positive.

```
1982 function mathfont:make_a_table(index, charm_data, fontdata)
1983   local a_table = {}
1984   local char = fontdata.characters[index] or {}
1985   local ex = fontdata.parameters.x_height or 0
1986   local slant = (fontdata.parameters.slant or 0)/ 65536
```

Get the dimensions of the character. We determine the new bounding box dimensions, horizontal glyph placement, and accent placement in terms of the character's original width (plus italic correction).

```
1987   local width, height, depth, italic = self.glyph_info(char)
```

Incorporate the italic correction into the character width.

```
1988   width = width + italic
```

We extract charm information from `charm_data` depending on whether the font is slanted or not.

```
1989   local left_stretch
1990   local right_stretch
1991   local top_accent
1992   local bot_accent
1993   if slant == 0 then
1994     left_stretch = charm_data.data_rm[1]
1995     right_stretch = charm_data.data_rm[2]
1996     top_accent = charm_data.data_rm[3]
1997     bot_accent = charm_data.data_rm[4]
1998   else
1999     left_stretch = charm_data.data_it[1]
2000     right_stretch = charm_data.data_it[2]
2001     top_accent = charm_data.data_it[3]
2002     bot_accent = charm_data.data_it[4]
2003   end
```

The new width is $1+$`left_stretch`$+$`right_stretch` times the original width. The horizontal offset that appears in the commands is the `left_stretch` portion of the new width.

```
2004   local offset = width * left_stretch
2005   a_table.width = width * (1 + left_stretch + right_stretch)
2006   a_table.height = height
2007   a_table.depth = depth
```

```
2008  a_table.italic = 0
```

Unicode information attached to the character.

```
2009  a_table.unicode = index
2010  a_table.tounicode = self.utf_16BE(index)
2011  a_table.index = char.index
```

If `left_stretch` is nonzero, we have to turn the character into a virtual character that typesets the glyph through a `char` command—that is the only way to add space on the left of the glyph. (A nonzero `right_stretch` is easier because we only have to extend the bounding box.)

```
2012  if offset \noexpand~= 0 then
2013    if char.commands then
2014      error(\M@virtual@ssert)
2015    else
2016      a_table.commands = {{"right", offset}, {"char", index}}
2017    end
2018  end
```

We calculate accent placement in two steps. The first step is to calculate a "base" accent position from character and font properties, and then we modify the base position according to charm information. If the font contains a `top_accent` value for the character, we take that value as our base accent position.

```
2019  local top_base
2020  if char.top_accent then
2021    top_base = char.top_accent
```

If the font does not contain a `top_accent` value, which for text fonts is the more likely possibility, we have to create the base accent position ourselves. We use an approach similar to how TEX positions accents in text mode. If the character is less than 1ex tall, the base accent position is halfway across the character's bounding box, i.e. any horizontal offset plus half the original width and half the italic correction. If the character is taller than 1ex, we move the base accent position right by the font's `slant` value times the character's height above 1ex. The base accent position in this approach does not change when the user changes the `left_stretch` or `right_stretch` values in the charm information.

```
2022  else
2023    top_base = offset + 0.5 * width
2024    if height > ex then
2025      top_base = top_base + slant * (height - ex)
2026    end
2027  end
```

We take a similar approach for the bottom accent. For the base accent position, we subtract the `slant` times 1ex.

```
2028  local bot_base
2029  if char.bot_accent then
2030    bot_base = char.bot_accent
2031  else
2032    bot_base = offset + 0.5 * width - slant * ex
2033  end
```

Now add the accent information to the font.

```
2034  a_table.top_accent = top_base + top_accent * width
2035  a_table.bot_accent = bot_base + bot_accent * width
```

TEX shifts superscripts (but not subscripts) right by the italic correction. We added italic correction to the character's width. Effectively, that shifts any superscript right relative to the character's original bounding box, so our changes to the character result in superscripts that behave the way we expect. However, the larger bounding box also affects subscripts, which we don't want, so we implement a mathkern table that moves any subscripts left by the italic correction. A mathkern table contains up to four subtables, one for each corner of the character. Within each subtable, we store pairs of `height` and `kern` values, where `height` means to apply `kern` to exponents at that height. In this case, we have a `kern` value of minus italic correction in the lower right corner.

```
2036  a_table.mathkern = {}
2037  a_table.mathkern.top_right = {{height = 0, kern = 0}}
2038  a_table.mathkern.bottom_right = {{height = 0, kern = -italic}}
2039  a_table.mathkern.top_left = {{height = 0, kern = 0}}
2040  a_table.mathkern.bottom_left = {{height = 0, kern = 0}}
2041  return a_table
2042 end
```

For type `e` characters, we add several virtual characters to the font, and we use `make_e_commands` to produce their commands tables. The commands tables from this function produces a scaled version of the glyph in slot `index`. The `pdf` command sends code directly to the pdf backend that handles the transformation. The `q` specification induces a linear transformation of the output, which in this case is a dilation by `h_stretch` and `v_stretch` factors. The `Q` command restores the original coordinate system.

```
2043 function mathfont.make_e_commands(index, h_stretch, v_stretch)
2044  local c_1 = {"pdf", "origin",
2045    string.format("q \@percentchar s 0 0 \@percentchar s 0 0 cm",
2046    h_stretch, v_stretch)}
2047  local c_2 = {"char", index}
```

```
  local c_3 = {"pdf", "origin", "Q"}
  return {c_1, c_2, c_3}
end
```

The function for type `e` characters returns a list of character subtables because we need to create multiple characters at once. Specifically, the function returns a new subtable for the original character plus one subtable for each larger variant. The structure is similar to `:make_type_a`, except that we scale the glyph instead of enlarging the bounding box.

```
function mathfont:make_e_table(index, charm_data, fontdata)
  local e_table = {}
  local char = fontdata.characters[index] or {}
  local ex = fontdata.parameters.x_height or 0
  local slant = (fontdata.parameters.slant or 0) / 65536
  local width, height, depth, italic = self.glyph_info(char)
```

Extract the charm information other than scale factors. Here `_sc` is short for scale.

```
  local v = charm_data.num_variants
  local smash = charm_data.smash
  local next = charm_data.next
  local top_accent_sc
  local bot_accent_sc
  local italic_sc
  if slant == 0 then
    top_accent_sc = charm_data.data_rm[2*v+1]
    bot_accent_sc = charm_data.data_rm[2*v+2]
    italic_sc = charm_data.data_rm[2*v+3]
  else
    top_accent_sc = charm_data.data_it[2*v+1]
    bot_accent_sc = charm_data.data_it[2*v+2]
    italic_sc = charm_data.data_it[2*v+3]
  end
```

Calculate accent placement for the original glyph, and rescale the italic correction. We calculate accent placement similarly to type `a`, and we change the italic correction afterwards to ensure that accent placement is independent of `italic_scale`. Unlike with type `a`, we do not enlarge the bounding box on the left side of the character, so the base accent placement does not contain `offset`.

```
  local top_base
  if char.top_accent then
    top_base = char.top_accent
```

```
  else
    top_base = 0.5 * width
    if height > ex then
      top_base = top_base + slant * (height - ex)
    end
  end
  local bot_base = char.bot_accent or (0.5 * width - slant * ex)
  local top_accent = top_base + top_accent_sc * (width + italic)
  local bot_accent = bot_base + bot_accent_sc * (width + italic)
  italic = italic + italic_sc * (width + italic)
```

Store the Unicode encoding slot for reference later

```
  local tounicode = self.utf_16BE(index)
```

We create a number of entries in `e_table` equal to one plus the number of variants we want, which is stored in `charm_data.num_variants`. We begin with the first entry. This isn't a full character subtable because for the small version of the big operator, we won't replace the subtable already in `fontdata` but rather will add the information here into that subtable.

```
  e_table[1] = {}
  e_table[1].italic = italic
  e_table[1].unicode = index
  e_table[1].tounicode = self.utf_16BE(index)
  e_table[1].top_accent = top_accent
  e_table[1].bot_accent = bot_accent
  e_table[1].next = next[1]
```

Now loop through the large variants, and add each one to `e_table`. We begin by extracting the scale factors for the particular large variant.

```
  local h_stretch
  local v_stretch
  for i = 2, v + 1, 1 do
    if slant == 0 then
      h_stretch = charm_data.data_rm[i-1][1]
      v_stretch = charm_data.data_rm[i-1][2]
    else
      h_stretch = charm_data.data_it[i-1][1]
      v_stretch = charm_data.data_it[i-1][2]
    end
```

Now add the entries for the subtable.

```
    e_table[i] = {}
    e_table[i].width = width * h_stretch
    e_table[i].height = height * v_stretch
```

```
2106      e_table[i].depth = depth * v_stretch
2107      e_table[i].italic = italic * h_stretch
```

Add the Unicode information.

```
2108      e_table[i].unicode = index
2109      e_table[i].tounicode = tounicode
```

Accent placement.

```
2110      e_table[i].top_accent = top_accent * h_stretch
2111      e_table[i].bot_accent = bot_accent * h_stretch
```

Add the commands.

```
2112      e_table[i].commands =
2113        self.make_e_commands(smash, h_stretch, v_stretch)
```

If we aren't dealing with the last entry in the table, we need to add the character's `next` field. The next larger variant after the `i`th character will the the `i` + 1st character, and we can extract the index from the `charm_information`.

```
2114      if i <= v then
2115        e_table[i].next = charm_data.next[i]
2116      end
2117    end
2118    return e_table
2119  end
```

Before we get to the main font-changing functions, we code `make_fake_angle`, which returns a character table for the fake angle brackets. The idea is to transform a guillement such that the top 90% of the original bounding box lines up with the full bounding box (height plus depth) of the left parenthesis (U+28). We make the fake angle brackets as follows:

1. Take a (single or double) guillemet, and let $h$ be its height. We assume the guillemet has no depth.
2. Let $h_p$ and $d_p$ be the height and depth respectively of the left parenthesis.
3. Our transformation of the guillement will involve a translation and a dilation. The translation should lower the basepoint of the guillemet such that if $p$ is the proportion of the guillemet that is below the baseline after translation, then $(p - 0.1\sigma)h = d_p$, where $\sigma$ is the scale factor, assuming we apply the translation before the dilation. (The dilation fixes the current position within the virtual character regardless of its relation to the baseline.) This means $ph = d_p + 0.1h\sigma$.
4. The scale factor for the dilation is

$$\sigma = \frac{h_p + d_p}{0.9h},$$

so 90% of the guillemet height becomes equal to the size of the parenthesis.

The function accepts the index of a guillemet as `index` and the index of the smashed guillemet as `smash`. The `slot` argument is the encoding slot of the angle bracket. First, extract relevant character and font dimensions.

```
function mathfont:make_fake_angle(index, smash, slot, fontdata)
  local fab_table = {}          % fab for fake angle bracket
  local chars = fontdata.characters
  local lp = chars[40] or {}    % lp for left parenthesis
  local lp_width, lp_height, lp_depth, lp_italic =
    self.glyph_info(lp)
  local g = chars[index] or {}  % g for guillemet
  local g_width, g_height, g_depth, g_italic = self.glyph_info(g)
  local ex = fontdata.parameters.x_height or 0
  local slant = (fontdata.parameters.slant or 0) / 65536
```

We compute the dimensions of our fake angle bracket.

```
  local factor
  if g_height \noexpand~= 0 then
    factor = (lp_height + lp_depth) / (0.9 * g_height)
  else
    factor = 1
  end
  local shift = lp_depth + 0.1 * factor * g_height
```

Now populate `fab_table`.

```
  fab_table.height = lp_height
  fab_table.depth = lp_depth
  fab_table.width = g_width
  fab_table.italic = g_italic
```

Unicode information.

```
  fab_table.unicode = slot
  fab_table.tounicode = self.utf_16BE(slot)
```

We calculate accent placement information the same way as we have been doing for type `a` and type `e` characters.

```
  if g.top_accent then
    fab_table.top_accent = g.top_accent
  else
    if g_height > ex then
      fab_table.top_accent = 0.5 * g_width +
                             slant * (g_height - ex)
    else
```

```
      fab_table.top_accent = 0.5 * g_width
    end
  end
  fab_table.bot_accent = g.bot_accent or
    (0.5 * g_width - slant * ex)
```

Commands.

```
  fab_table.commands = {
    {"down", shift},
    {"pdf", "origin",
      string.format("q 1 0 0 \@percentchar s 0 0 cm", factor)},
    {"char", smash},
    {"pdf", "origin", "Q"},
    {"down", -shift}}
  return fab_table
end
```

Similar function that returns the character subtable for a nabla (inverted increment symbol/capital Delta). Again, `index` is the encoding slot for an increment symbol, and `smash` is the encoding slot for a smashed version of an increment.

```
function mathfont:make_nabla(index, smash, slot, fontdata)
  local n_table = {}                    % n for nabla
  local i = fontdata.characters[index]  % i for increment
  local i_width, i_height, i_depth, i_italic = self.glyph_info(i)
```

Now populate `n_table`.

```
  n_table.width = i_width
  n_table.height = i_height
  n_table.depth = i_depth
  n_table.italic = i_italic
```

Unicode information.

```
  n_table.unicode = slot
  n_table.tounicode = self.utf_16BE(slot)
```

Take accent placement values from the increment symbol if they exist.

```
  if i.top_accent then
    n_table.bot_accent = i_width - i.top_accent
  end
  if i.bot_accent then
    n_table.top_accent = i_width - i.bot_accent
  end
```

Commands. We reflect the increment glyph horizontally and vertically.

```
  n_table.commands = {
```

```
    {"down", -i_height},
    {"right", i_width},
    {"pdf", "origin", "q -1 0 0 -1 0 0 cm"},
    {"char", smash},
    {"pdf", "origin", "Q"},
    {"right", -i_width},
    {"down", i_height}}
  return n_table
end
```

We come to the functions that modify the font. We need to accomplish three tasks, and we define separate functions for each one. First, we set the font's `nomath` entry to `false`. Second, we incorporate the modifications based on charm information into the font, i.e. set the font's character subtables using the previous functions from this section. Third, we need to add a MathConstants table. The first task is very easy.

```
function mathfont.set_nomath_false(fontdata)
  fontdata.nomath = false
  fontdata.oldmath = false
end
```

The second task is more involved. The basic idea is to loop through `mathfont`, and whenever we find an entry that is a subtable, we treat it as charm information that we use to modify the font object. We begin by storing the character information from the font in `chars` for easier reference later.

```
function mathfont.apply_charm_info(fontdata)
  local chars = fontdata.characters or {}
```

Before we loop through the charm data, we need to make a few changes to the font, namely add fake angle brackets, add nabla, and trim the bounding box on the surd. First, we add fake angle brackets. We use `mathfont.extra_chars` to track where we put the extra (virtual) characters in the font.

```
  local temp = mathfont.extra_chars
  chars[temp] = mathfont.smash_glyph(0x2039)   % smashed \lguil
  chars[temp+1] = mathfont.smash_glyph(0x203A) % smashed \rguil
  chars[temp+2] = mathfont.smash_glyph(0xAB)   % smashed \llguil
  chars[temp+3] = mathfont.smash_glyph(0xBB)   % smashed \rrguil
```

Now add the characters to the font.

```
  chars[mathfont.fakel] = mathfont:make_fake_angle(  %\fakelangle
    0x2039, temp, 0x27E8, fontdata)
  chars[mathfont.faker] = mathfont:make_fake_angle(  %\fakerangle
    0x203A, temp+1, 0x27E9, fontdata)
  chars[mathfont.fakell] = mathfont:make_fake_angle( %\fakellangle
```

```
    0xAB, temp+2, 0x27EA, fontdata)
  chars[mathfont.fakerr] = mathfont:make_fake_angle( %\fakerrangle
    0xBB, temp+3, 0x27EB, fontdata)
```

If the function doesn't have an increment character, copy a capital Delta if it exists in the font.

```
  if chars[0x394] and (not chars[0x2206]) then
    chars[0x2206] = {}
    for k,v in pairs(chars[0x394]) do
      chars[0x2206][k] = v
    end
    chars[0x2206].commands = {{"char", 0x394}}
  end
```

Add the nabla (inverted increment/capital Delta) character to the font if it is missing.

```
  if chars[0x2206] and (not chars[0x2207]) then
    chars[temp+8] = mathfont.smash_glyph(0x2206)
    chars[0x2207] = mathfont:make_nabla(
      0x2206, temp+8, 0x2207, fontdata)
  end
```

We trim the bounding box on the surd if the user requests it. Some text fonts extend the bounding box of the surd past the edge of the glyph, and we trim the edge of the box according to the values of `\vsurdfactor` and `\hsurdfactor`.

```
  if chars[0x221A] then
    local hsurd = tex.getcount("hsurdfactor") / 1000
    local vsurd = tex.getcount("vsurdfactor") / 1000
    chars[0x221A].width = hsurd * chars[0x221A].width
    chars[0x221A].height = vsurd * chars[0x221A].height
  end
```

Perform the loop. We care about entries `info` whose index is a number because those entries are the charm information that determines how we modify the font object. We ignore charm information for any characters not present in the font.

```
  for index, info in pairs(mathfont) do
    if type(index) == "number" and chars[index] then
```

One each iteration of the loop, we start by checking the `type` of the current entry because the handling of the font object varies based on the character type. For characters of type `a`, we insert our character subtable into the font object.

```
      if info.type == "a" then
        chars[index] = mathfont:make_a_table(index,info,fontdata)
```

For type `e` we have to add entries to `chars[index]` and insert multiple character subtables into the font, namely one for the smashed version of the base glyph and others corresponding to the large variants.

```
      elseif info.type == "e" then
        local variants_table =
          mathfont:make_e_table(index, info, fontdata)
```

First add entries to the subtable for the base glyph.

```
        for k,v in pairs(variants_table[1]) do
          chars[index][k] = v
        end
```

Smashed version of the glyph.

```
        chars[info.smash] = mathfont.smash_glyph(index)
```

Now add the large variants.

```
        for i = 1, info.num_variants, 1 do
          chars[info.next[i]] = variants_table[i+1]
        end
      end
    end
  end
end
```

The `populate_math_constants` function is even longer because we need to add a full MathConstants table to the font object, which has some fifty parameters that we need to set. (But the mechanics behind the function are simpler than `apply_charm_info`.) We set the font parameters in terms of traditional TEX `\fontdimen` parameters. Besides the eight essential parameters found in all fonts, TEX traditionally uses some fifteen extra parameters to typeset math formulas. To preserve whatever structure may already exist in the font object, we do not override any MathConstants that the font already contains. For brevity, we let `MC` be a shortcut for the MathConstants table.

```
function mathfont.math_constants(fontdata)
  fontdata.MathConstants = fontdata.MathConstants or {}
  local MC = fontdata.MathConstants
```

First evaluate the dimensions from the font object that we will use in determining other math parameter values. The `A_height` is the height of the capital "A" character, and the `y_depth` is the depth of the lower-case "y" character. Both will be 0 if the font does not have the correct character.

```
  local size = fontdata.size or 0
  local ex = fontdata.parameters.x_height or 0
  local em = fontdata.parameters.quad or 0
```

```
2251  local A_height = 0
2252  local y_depth = 0
2253  if fontdata.characters[65] then
2254    A_height = fontdata.characters[65].height or 0 % A
2255  end
2256  if fontdata.characters[121] then
2257    y_depth = fontdata.characters[121].depth or 0  % y
2258  end
```

We begin by setting the axis height and default rule thickness. We need to start with these parameters because we will use them to calculate other constants. We set both values to 0 initially and then change them.

```
2259  local axis = 0
2260  local rule_thickness = 0
```

Set the default rule thickness. If the font already has a value set for the parameter `FractionRuleThickness`, we take that as the default rule thickness, and otherwise we set it to be 1/18 of the font size times the adjustment factor from `\rulethicknessfactor`.

```
2261  if MC.FractionRuleThickness then
2262    rule_thickness = MC.FractionRuleThickness
2263  else
2264    local temp =
2265      tex.getcount("rulethicknessfactor") / 1000
2266    rule_thickness = (size / 18) * temp
2267    MC.FractionRuleThickness = rule_thickness
2268  end
```

If the font has an `AxisHeight`, we take that value as the `axis`. If the font does not have `AxisHeight` already set, we set the axis to be the height of a minus sign, which has position U+2212 (8722 in decimal). As a fallback, we set the axis to 0.8ex if the font does not have a character in slot U+2212.

```
2269  if MC.AxisHeight then
2270    axis = MC.AxisHeight
2271  else
2272    if fontdata.characters[8722] then
2273      axis = fontdata.characters[8722].height - rule_thickness / 2
2274    else
2275      axis = 0.8 * ex
2276    end
2277    MC.AxisHeight = axis
2278  end
```

Apart from the axis height and rule thickness, we can group the traditional mathematics `\fontdimen` parameters into three categories: four for large operators, five for fractions, and six for superscripts and subscripts. (OpenType math does not use the fifth large-operator parameter $\xi_{13}$ and the seventh script parameter $\sigma_{14}$.) We define variables with the same names as their traditional references from Appendix G in the *TEXBook*. I have taken the design approach of using twice the rule height as a standard minimum clearance, and I am assuming that script styles are roughly 70% as large as text and display styles. We begin with the parameters for large operators.

The parameter $\xi_9$ is the minimum clearance between the top of a large operator and the limit above it, and we set it to be twice the rule thickness. Before ensuring that the bottom of the upper limit is at least $\xi_9$ away from the operator character, TEX attempts to position the baseline of the limit at $\xi_{10}$ distance above the operator character, and we set $\xi_{10}$ to be slightly larger than $\xi_9$. If the upper limit has no decender, TEX will raise its baseline by $\xi_{10}$, and if it has a descener, TEX will position the bottom of the descender to be $\xi_9$ above the operator, which in practice means it will be higher than limits without descenders. This approach balances the desire for consistency in whitespace with the desire for consistency in baseline height. Similarly, we set the minimum clearance $\xi_{11}$ for the lower limit to be equal to the attempted clearance for the upper limit, and the attempted clearance $\xi_{12}$ for the lower limit will be the minimum clearance plus the average of the `\scriptfont` x-height and `\scriptfont` A-height.

```
2279  local xi_9 = 2 * rule_thickness     % top limit min clearance
2280  local xi_10 = xi_9 + 0.35 * y_depth % bottom limit try placement
2281  local xi_11 = xi_10                 % top limit min clearance
2282  local xi_12 = xi_10 + 0.35 * (A_height + ex) % bottom attempt
```

Our general approach for `\displaystyle` fractions is to place the baseline of the numerator numerator at a distance above the fraction rule of 1.5 times the rule height plus descender depth plus a small extra space. The minimum clearance will be the rule height, so we expect the numerator to strictly exceed the minimum clearance in most situations. Doing so produces consistent baselines of numerators and gives our value for $\sigma_8$, the attempted height of the numerator in `\displaystyle` fractions. For smaller styles, we use a single rule height as clearance, so we add $0.5 * \texttt{rule\_thickness} + \texttt{y\_depth}$ scaled down by 0.7 to the rule thickness. The minimum clearance for numerator and denominator are separate OpenType parameters, and we set them later. The extra 0.1 A-height in the attempted clearance relative to the minimum clearance appears because we measure attempted clearance from the axis, whereas we measure minimum clearance from the top or bottom of the fraction rule.

```
local sigma_8 = axis + 1.5 * rule_thickness + y_depth +
  0.1 * A_height
local sigma_9 = axis + 1.35 * rule_thickness + 0.7 * y_depth +
  0.07 * A_height
local sigma_10 = sigma_9
```

Our approach in the denominators is the same except that we add half the descender depth to the minimum clearance. This creates extra space below the fraction rule so that the typographical color above the rule matches that below the rule when the numerator contains descenders.

```
local sigma_11 = -axis + 1.5  * rule_thickness +
  0.5 * y_depth + 1.1 * A_height
local sigma_12 = -axis + 1.35 * rule_thickness +
  0.35 * y_depth + 0.77 * A_height
```

For superscripts we think in terms of the top of the superscript. We raise the baseline of the superscript by the desired height of the superscript top minus the `\scriptfont` A-height. Choosing $1.3 * $ `A_height` for regular styles and $1.2 * $ `A_height` for cramped styles was a design choice that worked well. The attempted drop for subscripts is one-fifth the A-height or slightly more than the y-depth, whichever is greater. This way the subscript baseline is slightly lower than any descenders, and for fonts without descenders, we still clearly lower the subscript. Setting $\sigma_{18}$ and $\sigma_{19}$ was another design choice that worked well.

```
local sigma_13 = 0.6 * A_height   % attempted superscript height
local sigma_15 = 0.5 * A_height   % attempt for cramped scripts
local sigma_16 = 1.1 * y_depth    % attempted subscript lower
if sigma_16 < 0.2 * A_height then
  sigma_16 = 0.2 * A_height
end
local sigma_17 = sigma_16         % sigma_16 when superscript
local sigma_18 = 0.5 * A_height   % superscript lower for boxed
local sigma_19 = 0.1 * A_height   % subscript lower for boxed
```

The MathConstants themselves come from the Unicode equivalents of the traditional TEX `\fontdimen` parameters where appropriate. Where not appropriate, I made design choices as indicated. Setting the next three parameters was purely a design choice.

```
if not MC.DisplayOperatorMinHeight then
  MC.DisplayOperatorMinHeight = 1.8 * A_height
end
if not MC.FractionDelimiterDisplayStyleSize then
  MC.FractionDelimiterDisplayStyleSize = 2 * size
```

```
  end
  if not MC.FractionDelimiterSize then
    MC.FractionDelimiterSize = 1.3 * size
  end
  if not MC.FractionDenominatorDisplayStyleShiftDown then
    MC.FractionDenominatorDisplayStyleShiftDown = sigma_11
  end
  if not MC.FractionDenominatorShiftDown then
    MC.FractionDenominatorShiftDown = sigma_12
  end
```

We set the minium clearance for the numerator to be twice the rule height in `\displaystyle` and the rule height in other styles. Our approach in setting the attempted height of the numerator ($\sigma_8$ and $\sigma_9$) was to add the minimum clearance plus the descender depth plus a small extra space, so in general, we do not expect the numerator to run into the minimum clearance. For the denominator, we do the same thing except add half the descender depth to the clearance, which balances the amount of color above and below the fraction rule and is similar to what we did for the lower limits on big operators when we set $\xi_{11}$ larger than $\xi_9$.

```
  if not MC.FractionDenominatorDisplayStyleGapMin then
    MC.FractionDenominatorDisplayStyleGapMin =
      rule_thickness + 0.5 * y_depth
  end % that MathConstants entry has a long name lol
  if not MC.FractionDenominatorGapMin then
    MC.FractionDenominatorGapMin =
      rule_thickness + 0.35 * y_depth
  end
  if not MC.FractionNumeratorDisplayStyleShiftUp then
    MC.FractionNumeratorDisplayStyleShiftUp = sigma_8
  end
  if not MC.FractionNumeratorShiftUp then
    MC.FractionNumeratorShiftUp = sigma_9
  end
  if not MC.FractionNumeratorDisplayStyleGapMin then
    MC.FractionNumeratorDisplayStyleGapMin = rule_thickness
  end
  if not MC.FractionNumeratorGapMin then
    MC.FractionNumeratorGapMin = rule_thickness
  end
```

The `SkewedFractionHorizontalGap` and `SkewedFractionVerticalGap` take the values that LuaTEXwould set for a traditional TEX font.

```
  if not MC.SkewedFractionHorizontalGap then
    MC.SkewedFractionHorizontalGap = 0.5 * em
  end
  if not MC.SkewedFractionVerticalGap then
    MC.SkewedFractionVerticalGap = ex
  end
```

The `UpperLimit` and `LowerLimit` dimensions correspond exactly to traditional TEX math `\fontdimen` parameters.

```
  if not MC.UpperLimitBaselineRiseMin then
    MC.UpperLimitBaselineRiseMin = xi_11
  end
  if not MC.UpperLimitGapMin then
    MC.UpperLimitGapMin = xi_9
  end
  if not MC.LowerLimitBaselineDropMin then
    MC.LowerLimitBaselineDropMin = xi_12
  end
  if not MC.LowerLimitGapMin then
    MC.LowerLimitGapMin = xi_10
  end
```

Traditional TEX doesn't have stack objects, but they are meant to be similar to large operators, so we set the same parameters.

```
  if not MC.StretchStackGapBelowMin then
    MC.StretchStackGapBelowMin = xi_10
  end
  if not MC.StretchStackTopShiftUp then
    MC.StretchStackTopShiftUp = xi_11
  end
  if not MC.StretchStackGapAboveMin then
    MC.StretchStackGapAboveMin = xi_9
  end
  if not MC.StretchStackBottomShiftDown then
    MC.StretchStackBottomShiftDown = xi_12
  end
```

For the three `Overbar` parameters, we take the approach that the bar itself should be as thick as the rule height. The gap will be twice the rule height, and the extra clearance will be a single rule height.

```
  if not MC.OverbarExtraAscender then
    MC.OverbarExtraAscender = rule_thickness
  end
  if not MC.OverbarRuleThickness then
```

```
2370    MC.OverbarRuleThickness = rule_thickness
2371  end
2372  if not MC.OverbarVerticalGap then
2373    MC.OverbarVerticalGap = 2 * rule_thickness
2374  end
```

For the radical sign, we take the same approach as with the `Overbar` parameters. We insert one rule thickness of extra space above the radical symbol and two rule thickness of extra space under it. For `\textstyle` and smaller, we reduce the space to a single rule height.

```
2375  if not MC.RadicalExtraAscender then
2376    MC.RadicalExtraAscender = rule_thickness
2377  end
2378  if not MC.RadicalRuleThickness then
2379    MC.RadicalRuleThickness = rule_thickness
2380  end
2381  if not MC.RadicalDisplayStyleVerticalGap then
2382    MC.RadicalDisplayStyleVerticalGap = 2 * rule_thickness
2383  end
2384  if not MC.RadicalVerticalGap then
2385    MC.RadicalVerticalGap = rule_thickness
2386  end
```

The final three `Radical` parameters aren't used if we handle degree placement at the macro level rather than at the font level. We set them to the default values that LuaTEX uses for traditional tfm fonts.

```
2387  if not MC.RadicalKernBeforeDegree then
2388    MC.RadicalKernBeforeDegree = (5/18) * em
2389  end
2390  if not MC.RadicalKernAfterDegree then
2391    MC.RadicalKernAfterDegree = (10/18) * em
2392  end
2393  if not MC.RadicalDegreeBottomRaisePercent then
2394    MC.RadicalDegreeBottomRaisePercent = 60
2395  end
```

The `SpaceAfterScript` is a design choice. Somewhat arbitrary.

```
2396  if not MC.SpaceAfterScript then
2397    MC.SpaceAfterScript = 0.1 * em
2398  end
```

The `Stack` parameters come from their traditional `\fontdimen` analogues.

```
2399  if not MC.StackBottomDisplayStyleShiftDown then
2400    MC.StackBottomDisplayStyleShiftDown = sigma_11
```

```
  end
  if not MC.StackBottomShiftDown then
    MC.StackBottomShiftDown = sigma_12
  end
  if not MC.StackTopDisplayStyleShiftUp then
    MC.StackTopDisplayStyleShiftUp = sigma_8
  end
  if not MC.StackTopShiftUp then
    MC.StackTopShiftUp = sigma_10
  end
```

Traditionally TEX uses an internal method rather than a parameter to determine the minimum distance between two boxes in an `\atop` stack. We set the minimum distance to be one rule thickness plus the combined minimum clearance for numerators and denominators in fractions. For `\displaystyle`, that gives us

`rule_thickness`+(2∗`rule_thickness`)+(2∗`rule_thickness`+0.5∗`y_depth`)

For smaller styles, we use single rule height values and scale down the `y_depth` by 0.7.

```
  if not MC.StackDisplayStyleGapMin then
    MC.StackDisplayStyleGapMin =
      5 * rule_thickness + 0.5 * y_depth
  end
  if not MC.StackGapMin then
    MC.StackGapMin = 3 * rule_thickness + 0.35 * y_depth
  end
```

With three exceptions, superscript and subscript parameters come from traditional TEX dimensions.

```
  if not MC.SubscriptShiftDown then
    MC.SubscriptShiftDown = sigma_16
  end
  if not MC.SubscriptBaselineDropMin then
    MC.SubscriptBaselineDropMin = sigma_19
  end
  if not MC.SubscriptShiftDownWithSuperscript then
    MC.SubscriptShiftDownWithSuperscript = sigma_17
  end
```

The top of a subscript should be less than half the A-height. This is a somewhat arbitrary design choice.

```
  if not MC.SubscriptTopMax then
```

```
2428    MC.SubscriptTopMax = 0.5 * A_height
2429  end
```

The minimum gap between superscripts and subscripts will be the height of the rule. This is less space than TEX traditionally allocates.

```
2430  if not MC.SubSuperscriptGapMin then
2431    MC.SubSuperscriptGapMin = rule_thickness
2432  end
```

We set the minimum height for the bottom of a subscript to be the height of a superscript in cramped styles minus the depth of a possible descender. Theoretically this is the lowest that any portion of a superscript should ever be if it contains only text.

```
2433  if not MC.SuperscriptBottomMin then
2434    MC.SuperscriptBottomMin = sigma_15 - 0.7 * y_depth
2435  end
2436  if not MC.SuperscriptBaselineDropMax then
2437    MC.SuperscriptBaselineDropMax = sigma_18
2438  end
2439  if not MC.SuperscriptShiftUp then
2440    MC.SuperscriptShiftUp = sigma_13
2441  end
2442  if not MC.SuperscriptShiftUpCramped then
2443    MC.SuperscriptShiftUpCramped = sigma_15
2444  end
```

If the superscript and subscript overlap, we choose the new position such that the baselines of subscripts are roughly consistent across subformulas. In this case, the bottom of the superscript box will rise at most to the point such that a subscript containing only text at 70% of the next-larger style will align with all similar subscripts. The top of the subscript will have approximate height $-\sigma_{16} + 0.7 * \texttt{A\_height}$ above the baseline, so to find our desired position for the bottom of the superscript, we add the minimum clearance of a single rule thickness. Putting this parameter in terms of the subscript sizing is necessary because we don't know how large the descender will be in a given subscript.

```
2445  if not MC.SuperscriptBottomMaxWithSubscript then
2446    MC.SuperscriptBottomMaxWithSubscript =
2447      -sigma_16 + 0.7 * A_height + rule_thickness
2448  end
```

As with the `Overbar` parameters, we set the extra clearance to be the rule height and the gap to be twice the rule height.

```
2449  if not MC.UnderbarExtraDescender then
2450    MC.UnderbarExtraDescender = rule_thickness
```

```
  end
  if not MC.UnderbarRuleThickness then
    MC.UnderbarRuleThickness = rule_thickness
  end
  if not MC.UnderbarVerticalGap then
    MC.UnderbarVerticalGap = 2 * rule_thickness
  end
```

No reason not to set `MinConnectorOverlap` to 0. It doesn't matter for our purposes because mathfont doesn't use extensibles.

```
  if not MC.MinConnectorOverlap then
    MC.MinConnectorOverlap = 0
  end
end
```

Time for callbacks! We create seven of them.

```
luatexbase.create_callback("mathfont.inspect_font",
  "simple", mathfont.empty)
luatexbase.create_callback("mathfont.pre_adjust",
  "simple", mathfont.empty)
luatexbase.create_callback("mathfont.disable_nomath",
  "simple", mathfont.set_nomath_false)
luatexbase.create_callback("mathfont.add_math_constants",
  "simple", mathfont.math_constants)
luatexbase.create_callback("mathfont.fix_character_metrics",
  "simple", mathfont.apply_charm_info)
luatexbase.create_callback("mathfont.post_adjust",
  "simple", mathfont.empty)
luatexbase.create_callback("mathfont.finishing_touches",
  "simple", mathfont.empty)
```

The functions `mathfont.info` and `mathfont.get_font_name` are used for informational messaging. The first prints a message in the `log` file, and the second returns a font name.

```
function mathfont.info(msg)
  texio.write_nl("log", "Package mathfont Info: " .. msg)
end
function mathfont.get_font_name(fontdata)
  return fontdata.psname or
         fontdata.fullname or
         fontdata.name or "<unknown font name>"
end
```

The `adjust_font` function is what actually goes in `luaotfload.patch_font`. This function calls the six callbacks at appropriate times and writes informa-

tional messages in the `log` file. We adjust the font object when `nomath` is `false` and the font is loaded in `base` mode. I am assuming that the user will usually load text-only fonts in `node` or `harf` mode, and then mathfont does not need to (and probably should not) alter that particular font. Unfortunately, there does not appear to be a better way to notate that we will use a font for text versus math when declaring it with `\DeclareFontSize`. I would ideally set `script=math` with the rest of the OpenType font features, but luaotfload ignores `script` declarations that aren't built into the font.

```
function mathfont.adjust_font(fontdata)
  luatexbase.call_callback("mathfont.inspect_font", fontdata)
  local the_font = mathfont.get_font_name(fontdata)
  if fontdata.nomath and
      fontdata.properties and
      fontdata.properties.mode and
      fontdata.properties.mode == "base" then
    luatexbase.call_callback("mathfont.pre_adjust",
      fontdata)
    luatexbase.call_callback("mathfont.disable_nomath",
      fontdata)
    luatexbase.call_callback("mathfont.add_math_constants",
      fontdata)
    luatexbase.call_callback("mathfont.fix_character_metrics",
      fontdata)
    luatexbase.call_callback("mathfont.post_adjust",
      fontdata)
  end
  luatexbase.call_callback("mathfont.finishing_touches", fontdata)
end
```

Finally, add the processing function to luaotfload's `patch_font` callback.

```
luatexbase.add_to_callback("luaotfload.patch_font",
  mathfont.adjust_font, "mathfont.adjust_font")
```

# 11 Adjust Fonts: Metrics

This section contains the default charm information for the characters that mathfont adjusts upon loading a font. We start with uppercase Latin letters. The first set of numbers applies to upright fonts, and the second set applies to italic/slanted fonts.

```
mathfont:new_type_a(0x41, {0,0,0,0},    {50,0,150,0})   %A
mathfont:new_type_a(0x42, {0,0,0,0},    {0,0,0,0})      %B
```

```
mathfont:new_type_a(0x43, {0,0,50,0},   {0,0,50,0})     %C
mathfont:new_type_a(0x44, {0,0,0,0},    {50,0,0,0})     %D
mathfont:new_type_a(0x45, {0,0,0,0},    {50,0,0,0})     %E
mathfont:new_type_a(0x46, {0,0,0,0},    {50,0,0,0})     %F
mathfont:new_type_a(0x47, {0,0,0,0},    {0,0,0,0})      %G
mathfont:new_type_a(0x48, {0,0,0,0},    {50,0,0,0})     %H
mathfont:new_type_a(0x49, {0,0,0,0},    {50,0,50,0})    %I
mathfont:new_type_a(0x4A, {0,0,100,0},  {50,0,100,0})   %J
mathfont:new_type_a(0x4B, {0,0,0,0},    {50,0,0,0})     %K
mathfont:new_type_a(0x4C, {0,0,-200,0}, {50,0,-100,0})  %L
mathfont:new_type_a(0x4D, {0,0,0,0},    {50,0,0,0})     %M
mathfont:new_type_a(0x4E, {0,0,0,0},    {50,0,0,0})     %N
mathfont:new_type_a(0x4F, {0,0,0,0},    {0,0,50,0})     %O
mathfont:new_type_a(0x50, {0,0,0,0},    {50,0,0,0})     %P
mathfont:new_type_a(0x51, {0,0,0,0},    {0,0,50,0})     %Q
mathfont:new_type_a(0x52, {0,0,0,0},    {50,0,0,0})     %R
mathfont:new_type_a(0x53, {0,0,0,0},    {0,0,0,0})      %S
mathfont:new_type_a(0x54, {0,0,0,0},    {0,0,0,0})      %T
mathfont:new_type_a(0x55, {0,0,0,0},    {0,0,0,0})      %U
mathfont:new_type_a(0x56, {0,0,0,0},    {0,0,0,0})      %V
mathfont:new_type_a(0x57, {0,0,0,0},    {0,0,0,0})      %W
mathfont:new_type_a(0x58, {0,0,0,0},    {50,0,0,0})     %X
mathfont:new_type_a(0x59, {0,0,0,0},    {0,0,0,0})      %Y
mathfont:new_type_a(0x5A, {0,0,0,0},    {50,0,0,0})     %Z
```

Lowercase Latin letters.

```
mathfont:new_type_a(0x61,  {0,0,0,0},     {0,0,0,0})    %a
mathfont:new_type_a(0x62,  {0,0,0,0},     {0,0,0,0})    %b
mathfont:new_type_a(0x63,  {0,0,0,0},     {0,0,100,0})  %c
mathfont:new_type_a(0x64,  {0,0,0,0},     {0,0,0,0})    %d
mathfont:new_type_a(0x65,  {0,0,0,0},     {0,0,100,0})  %e
mathfont:new_type_a(0x66,  {0,400,300,0}, {150,0,50,0}) %f
mathfont:new_type_a(0x67,  {0,0,0,0},     {0,0,0,0})    %g
mathfont:new_type_a(0x68,  {0,0,0,0},     {0,0,0,0})    %h
mathfont:new_type_a(0x69,  {0,0,0,0},     {0,0,0,0})    %i
mathfont:new_type_a(0x6A,  {100,0,0,0},   {200,0,0,0})  %j
mathfont:new_type_a(0x6B,  {0,0,0,0},     {0,0,0,0})    %k
mathfont:new_type_a(0x6C,  {0,0,0,0},     {0,0,0,0})    %l
mathfont:new_type_a(0x6D,  {0,0,0,0},     {0,0,50,0})   %m
mathfont:new_type_a(0x6E,  {0,0,0,0},     {0,0,50,0})   %n
mathfont:new_type_a(0x6F,  {0,0,0,0},     {0,0,100,0})  %o
mathfont:new_type_a(0x70,  {0,0,0,0},     {0,0,100,0})  %p
mathfont:new_type_a(0x71,  {0,0,0,0},     {0,0,0,0})    %q
```

```
mathfont:new_type_a(0x72,  {0,0,0,0},     {0,0,50,0})   %r
mathfont:new_type_a(0x73,  {0,0,0,0},     {0,0,0,0})    %s
mathfont:new_type_a(0x74,  {0,0,0,0},     {0,0,0,0})    %t
mathfont:new_type_a(0x75,  {0,0,0,0},     {0,0,0,0})    %u
mathfont:new_type_a(0x76,  {0,0,0,0},     {0,0,50,0})   %v
mathfont:new_type_a(0x77,  {0,0,0,0},     {0,0,50,0})   %w
mathfont:new_type_a(0x78,  {0,0,0,0},     {0,0,50,0})   %x
mathfont:new_type_a(0x79,  {0,0,0,0},     {0,0,50,0})   %y
mathfont:new_type_a(0x7A,  {0,0,0,0},     {0,0,0,0})    %z
mathfont:new_type_a(0x131, {0,0,0,0},     {0,0,50,0})   %\imath
mathfont:new_type_a(0x237, {0,0,0,0},     {200,0,50,0}) %\jmath
```

Uppercase Greek characters.

```
mathfont:new_type_a(0x391, {0,0,0,0}, {50,0,150,0})     %\Alpha
mathfont:new_type_a(0x392, {0,0,0,0}, {50,0,0,0})       %\Beta
mathfont:new_type_a(0x393, {0,0,0,0}, {50,0,0,0})       %\Gamma
mathfont:new_type_a(0x394, {0,0,0,0}, {50,0,150,0})     %\Delta
mathfont:new_type_a(0x395, {0,0,0,0}, {50,0,0,0})       %\Epsilon
mathfont:new_type_a(0x396, {0,0,0,0}, {50,0,0,0})       %\Zeta
mathfont:new_type_a(0x397, {0,0,0,0}, {50,0,0,0})       %\Eta
mathfont:new_type_a(0x398, {0,0,0,0}, {0,0,50,0})       %\Theta
mathfont:new_type_a(0x399, {0,0,0,0}, {50,0,0,0})       %\Iota
mathfont:new_type_a(0x39A, {0,0,0,0}, {50,0,0,0})       %\Kappa
mathfont:new_type_a(0x39B, {0,0,0,0}, {0,0,150,0})      %\Lambda
mathfont:new_type_a(0x39C, {0,0,0,0}, {50,0,0,0})       %\Mu
mathfont:new_type_a(0x39D, {0,0,0,0}, {50,0,0,0})       %\Nu
mathfont:new_type_a(0x39E, {0,0,0,0}, {0,0,0,0})        %\Xi
mathfont:new_type_a(0x39F, {0,0,0,0}, {0,0,50,0})       %\Omicron
mathfont:new_type_a(0x3A0, {0,0,0,0}, {50,0,0,0})       %\Pi
mathfont:new_type_a(0x3A1, {0,0,0,0}, {50,0,0,0})       %\Rho
mathfont:new_type_a(0x3A3, {0,0,0,0}, {50,0,0,0})       %\Sigma
mathfont:new_type_a(0x3A4, {0,0,0,0}, {0,0,0,0})        %\Tau
mathfont:new_type_a(0x3A5, {0,0,0,0}, {0,0,0,0})        %\Upsilon
mathfont:new_type_a(0x3A6, {0,0,0,0}, {0,0,50,0})       %\Phi
mathfont:new_type_a(0x3A7, {0,0,0,0}, {50,0,0,0})       %\Chi
mathfont:new_type_a(0x3A8, {0,0,0,0}, {0,0,0,0})        %\Psi
mathfont:new_type_a(0x3A9, {0,0,0,0}, {0,0,50,0})       %\Omega
mathfont:new_type_a(0x3F4, {0,0,0,0}, {0,0,50,0})       %\varTheta
```

Lowercase Greek characters.

```
mathfont:new_type_a(0x3B1, {0,0,0,0},    {0,0,0,0})     %\alpha
mathfont:new_type_a(0x3B2, {0,0,0,0},    {0,0,0,0})     %\beta
mathfont:new_type_a(0x3B3, {0,0,0,0},    {0,0,0,0})     %\gamma
```

```
mathfont:new_type_a(0x3B4, {0,0,0,0},    {0,0,0,0})     %\delta
mathfont:new_type_a(0x3B5, {0,0,50,0},   {0,0,50,0})    %\epsilon
mathfont:new_type_a(0x3B6, {0,0,50,0},   {0,0,0,0})     %\zeta
mathfont:new_type_a(0x3B7, {0,0,50,0},   {0,0,0,0})     %\eta
mathfont:new_type_a(0x3B8, {0,0,0,0},    {0,0,100,0})   %\theta
mathfont:new_type_a(0x3B9, {0,0,0,0},    {0,0,50,0})    %\iota
mathfont:new_type_a(0x3BA, {0,0,0,0},    {0,0,0,0})     %\kappa
mathfont:new_type_a(0x3BB, {0,0,-150,0}, {0,0,-100,0})  %\lambda
mathfont:new_type_a(0x3BC, {0,0,0,0},    {0,0,100,0})   %\mu
mathfont:new_type_a(0x3BD, {0,0,0,0},    {0,0,50,0})    %\nu
mathfont:new_type_a(0x3BE, {0,0,0,0},    {0,0,50,0})    %\xi
mathfont:new_type_a(0x3BF, {0,0,0,0},    {0,0,50,0})    %\omicron
mathfont:new_type_a(0x3C0, {0,0,0,0},    {0,0,0,0})     %\pi
mathfont:new_type_a(0x3C1, {0,0,0,0},    {0,0,50,0})    %\rho
mathfont:new_type_a(0x3C3, {0,0,0,0},    {0,0,0,0})     %\sigma
mathfont:new_type_a(0x3C4, {0,0,0,0},    {0,0,50,0})    %\tau
mathfont:new_type_a(0x3C5, {0,0,0,0},    {0,0,0,0})     %\upsilon
mathfont:new_type_a(0x3C6, {0,0,0,0},    {0,0,0,0})     %\phi
mathfont:new_type_a(0x3C7, {0,0,0,0},    {50,0,50,0})   %\chi
mathfont:new_type_a(0x3C8, {0,0,0,0},    {0,0,0,0})     %\psi
mathfont:new_type_a(0x3C9, {0,0,0,0},    {0,0,50,0})    %\omega
mathfont:new_type_a(0x3D0, {0,0,0,0},    {0,0,0,0})     %\varbeta
mathfont:new_type_a(0x3F5, {0,0,0,0},    {0,0,50,0})  %\varepsilon
mathfont:new_type_a(0x3D1, {0,0,150,0},  {0,0,100,0})   %\vartheta
mathfont:new_type_a(0x3F1, {0,0,0,0},    {0,0,50,0})    %\varrho
mathfont:new_type_a(0x3C2, {0,0,100,0},  {0,0,50,0})    %\varsigma
mathfont:new_type_a(0x3D5, {0,0,0,0},    {0,0,100,0})   %\varphi
```

We add the charm information for delimiters and other resizable characters. We divide the characters into four categories depending on how we want to magnify the base glyph to create large variants: delimiters, big operators, vertical characters, and the integral sign. We automate the process by putting charm information for each category of character into a separate table and feeding the whole thing to a wrapper around `:new_type_e`.

```
local delim_glyphs = {40,  % (
  41,                      % )
  47,                      % /
  91,                      % [
  92,                      % backslash
  93,                      % ]
  123,                     % {
  125,                     % }
  8249,                    % \lguil
```

```
  8250,                    % \rguil
  171,                     % \llguil
  187,                     % \rrguil
  mathfont.fakel,          % \fakelangle
  mathfont.faker,          % \fakerangle
  mathfont.fakell,         % \fakellangle
  mathfont.fakerr}         % \fakerrangle
local big_op_glyphs = {33, % !
  35,                      % #
  36,                      % $
  37,                      % %
  38,                      % &
  43,                      % +
  63,                      % ?
  64,                      % @
  167,                     % \S
  215,                     % \times
  247,                     % \div
  8719,                    % \prod
  8721,                    % \sum
  8720,                    % \coprod
  8897,                    % \bigvee
  8896,                    % \bigwedge
  8899,                    % \bigcup
  8898,                    % \bigcap
  10753,                   % \bigoplus
  10754,                   % \bigotimes
  10752,                   % \bigodot
  10757,                   % \bigsqcap
  10758}                   % \bigsqcup
local vert_glyphs = {124, 8730} % | and \surd
local int_glyphs = {8747,  % \intop
  8748,                    % \iint
  8749,                    % \iiint
  8750,                    % \oint
  8751,                    % \oiint
  8752}                    % \oiiint
```

Each category of type `e` character will have its own table of charm information with different magnification values. each table is initially empty.

```
local delim_scale = {}
local big_op_scale = {}
local vert_scale = {}
```

```
local int_scale = {}
```

Populate each table with magnification information. For every type `e` character we will create fifteen larger variants in the font. Delimiters stretch mostly vertically and some horzontally. Vertical characters stretch vertically only, so their horizontal scale factors are all constant. Big operators stretch the same in vertical and horizoontal directions.

```
for i = 1, 15, 1 do
  delim_scale[2*i-1] = 1000 + 100*i  % delimiters - horizontal
  delim_scale[2*i] = 1000 + 500*i    % delimiters - vertical
  vert_scale[2*i-1] = 1000
  vert_scale[2*i] = 1000 + 500*i     % vertically scaled chars
  big_op_scale[2*i-1] = 1000 + 100*i % big operators - horizontal
  big_op_scale[2*i] = 1000 + 100*i   % big operators - vertical
```

The integral sign is different. Visually, we would like an integral symbol that is larger than the large operators, which means that the integral sign should have no variants between the font's value of `\Umathoperatorsize` and the desired larger size. Accordingly, I decided it would be easiest to have large variants of the integral sign jump by large enough scale factors that the smallest variant larger than the regular size is already significantly larger than the `\Umathoperatorsize` setting in `populate_math_constants`. Effectively this means that the user should take the size of the integral operator as fixed and should set `\Umathoperatorsize` to make all other big operators the desired size.

```
  int_scale[2*i-1] = 1000 + 500*i    % integral sign - horizontal
  int_scale[2*i] = 1000 + 1500*i     % integral sign - vertical
end
```

We do not modify accent placement or italic corrections.

```
delim_scale[31] = 0
delim_scale[32] = 0
delim_scale[33] = 0
big_op_scale[31] = 0
big_op_scale[32] = 0
big_op_scale[33] = 0
vert_scale[31] = 0
vert_scale[32] = 0
vert_scale[33] = 0
int_scale[31] = 0
int_scale[32] = 0
int_scale[33] = 0
```

The wrapper for `:new_type_e`. We feed it a list of characters to create charm information for and a table of scaling information.

```
function mathfont:add_extensible_variants(char_list, scale_list)
  local variants = (\string# scale_list - 3) / 2
  for i = 1, \string# char_list, 1 do
    self:new_type_e(char_list[i], scale_list, scale_list)
  end
end
```

Add the charm information for the type `e` characters.

```
mathfont:add_extensible_variants(delim_glyphs, delim_scale)
mathfont:add_extensible_variants(big_op_glyphs, big_op_scale)
mathfont:add_extensible_variants(vert_glyphs, vert_scale)
mathfont:add_extensible_variants(int_glyphs, int_scale)
```

Finally, end the call to `\directlua` and balance the preceeding conditional.

```
}
\fi % matches previous \ifM@adjust@font
```

# 12 Unicode Hex Values

For this section, we don't want any `\endlinechar`s present when TEX scans things because we want to eliminate any extra spaces, so before anything else, we set `\endlinechar` to $-1$.

```
\count@\endlinechar
\endlinechar\m@ne
```

We have to save `\mathchar@type` to use after `\begin{document}` because LATEX feeds it to `\@onlypreamble`.

```
\let\@@mathchar@type\mathchar@type
```

We define `\M@sym@`, which is a wrapper around `\Umathchardef` or `\Umathcode` and is mathfont's version of `\DeclareMathSymbol`. Before version 3.0, mathfont used `\DeclareMathSymbol`, but we create our own version to support Unicode input. The command first checks whether `#1` is a control sequence. If yes, we define it using `\Umathchardef` and again using `\Umathcode`. Otherwise, we define it once with `\Umathcode`.

```
\def\M@sym@#1#2#3#4{
  \ifcat\relax\noexpand#1
```

Check if we're redefining a previously declared math symbol. We put `mathchar` in `\if@` so we don't have to check for `\mathchar` and `\Umathchar` separately. We use `\string` so that the letters in `mathchar` have catcode 12 when we check

for their presence in `\meaning#1` with `\in@`. If `#1` is undefined, `\in@` will set `\ifin@` to false.

```
    \expandafter\in@\expanded
      {{\expandafter\@gobble\string\mathchar}{\meaning#1}}
    \ifin@
```

Now redeclare the symbol.

```
      \Umathchardef#1=+\@@mathchar@type#2
        +\csname sym#3\endcsname+#4\relax
```

The next two lines implement Unicode input.

```
      \Umathcode   #4=+\@@mathchar@type#2
        +\csname sym#3\endcsname+#4\relax
```

If `#1` does not code for a math symbol, we check whether it is already defined. If no, we define it, and if yes, we issue an error. Again, we implement Unicode input.

```
    \else
      \ifx#1\@undefined
        \Umathchardef#1=+\@@mathchar@type#2
          +\csname sym#3\endcsname+#4\relax
        \Umathcode   #4=+\@@mathchar@type#2
          +\csname sym#3\endcsname+#4\relax
      \else
        \@latex@error{Command "\string#1" already defined}\@eha
      \fi
    \fi
```

Easy to deal with the case where `#1` is a single character.

```
  \else
    \Umathcode`#1=+\@@mathchar@type#2+\csname sym#3\endcsname
      +#4\relax
  \fi}
```

Similar deal for accents; `\M@acc@` is our version of `\DeclareMathAccent`. Newer versions of the LaTeX kernel define math accents as robust commands, so we have to incorporate a check for robustness as well. We let `\@tempswa` be true or false according to whether we can (re)define the control sequence `#1` as a math accent.

```
\def\M@acc@#1#2#3#4{
  \begingroup
    \@tempswatrue
    \ifdefined#1
      \expandafter\in@\expanded{
        {\expandafter\@gobble\string\mathaccent}
```

```
        {\meaning#1}}
      \ifin@
      \else
        \begingroup
          \escapechar\m@ne
          \expandafter
        \endgroup
        \expandafter\in@\expanded{
          {\string\mathaccent}
          {\expandafter\meaning\csname\string#1\space\endcsname}}
        \ifin@
        \else
          \@tempswafalse
        \fi
      \fi
    \fi
    \expandafter
  \endgroup
```

Now (re)define the command or issue an error.

```
  \if@tempswa
    \protected\edef#1{\Umathaccent+\@@mathchar@type#2
      +\csname sym#3\endcsname+#4\relax}
  \else
    \@latex@error{Command "\string#1" already defined}\@eha
  \fi}
```

Set upper-case Latin characters. We use an `\edef` for `\M@upper@id` because every expansion now will save LaTeX twenty-six expansions later when it evaluates each `\DeclareMathSymbol`. If the user has enabled Lua font adjustments, we set the mathcodes to use encoding slots in the Math Alphanumeric Symbols block.

`\M@upper@set`

```
\def\M@upper@set{
  \edef\M@upper@id{M\@tempc-\M@uppershape}
  \M@sym@{A}{\mathalpha}{\M@upper@id}{`A}
  \M@sym@{B}{\mathalpha}{\M@upper@id}{`B}
  \M@sym@{C}{\mathalpha}{\M@upper@id}{`C}
  \M@sym@{D}{\mathalpha}{\M@upper@id}{`D}
  \M@sym@{E}{\mathalpha}{\M@upper@id}{`E}
  \M@sym@{F}{\mathalpha}{\M@upper@id}{`F}
  \M@sym@{G}{\mathalpha}{\M@upper@id}{`G}
  \M@sym@{H}{\mathalpha}{\M@upper@id}{`H}
  \M@sym@{I}{\mathalpha}{\M@upper@id}{`I}
```

```
2766  \M@sym@{J}{\mathalpha}{\M@upper@id}{`J}
2767  \M@sym@{K}{\mathalpha}{\M@upper@id}{`K}
2768  \M@sym@{L}{\mathalpha}{\M@upper@id}{`L}
2769  \M@sym@{M}{\mathalpha}{\M@upper@id}{`M}
2770  \M@sym@{N}{\mathalpha}{\M@upper@id}{`N}
2771  \M@sym@{O}{\mathalpha}{\M@upper@id}{`O}
2772  \M@sym@{P}{\mathalpha}{\M@upper@id}{`P}
2773  \M@sym@{Q}{\mathalpha}{\M@upper@id}{`Q}
2774  \M@sym@{R}{\mathalpha}{\M@upper@id}{`R}
2775  \M@sym@{S}{\mathalpha}{\M@upper@id}{`S}
2776  \M@sym@{T}{\mathalpha}{\M@upper@id}{`T}
2777  \M@sym@{U}{\mathalpha}{\M@upper@id}{`U}
2778  \M@sym@{V}{\mathalpha}{\M@upper@id}{`V}
2779  \M@sym@{W}{\mathalpha}{\M@upper@id}{`W}
2780  \M@sym@{X}{\mathalpha}{\M@upper@id}{`X}
2781  \M@sym@{Y}{\mathalpha}{\M@upper@id}{`Y}
2782  \M@sym@{Z}{\mathalpha}{\M@upper@id}{`Z}}
```

Set lower-case Latin characters.

```
\M@lower@set 2783 \def\M@lower@set{
2784  \edef\M@lower@id{M\@tempc-\M@lowershape}
2785  \M@sym@{a}{\mathalpha}{\M@lower@id}{`a}
2786  \M@sym@{b}{\mathalpha}{\M@lower@id}{`b}
2787  \M@sym@{c}{\mathalpha}{\M@lower@id}{`c}
2788  \M@sym@{d}{\mathalpha}{\M@lower@id}{`d}
2789  \M@sym@{e}{\mathalpha}{\M@lower@id}{`e}
2790  \M@sym@{f}{\mathalpha}{\M@lower@id}{`f}
2791  \M@sym@{g}{\mathalpha}{\M@lower@id}{`g}
2792  \M@sym@{h}{\mathalpha}{\M@lower@id}{`h}
2793  \M@sym@{i}{\mathalpha}{\M@lower@id}{`i}
2794  \M@sym@{j}{\mathalpha}{\M@lower@id}{`j}
2795  \M@sym@{k}{\mathalpha}{\M@lower@id}{`k}
2796  \M@sym@{l}{\mathalpha}{\M@lower@id}{`l}
2797  \M@sym@{m}{\mathalpha}{\M@lower@id}{`m}
2798  \M@sym@{n}{\mathalpha}{\M@lower@id}{`n}
2799  \M@sym@{o}{\mathalpha}{\M@lower@id}{`o}
2800  \M@sym@{p}{\mathalpha}{\M@lower@id}{`p}
2801  \M@sym@{q}{\mathalpha}{\M@lower@id}{`q}
2802  \M@sym@{r}{\mathalpha}{\M@lower@id}{`r}
2803  \M@sym@{s}{\mathalpha}{\M@lower@id}{`s}
2804  \M@sym@{t}{\mathalpha}{\M@lower@id}{`t}
2805  \M@sym@{u}{\mathalpha}{\M@lower@id}{`u}
2806  \M@sym@{v}{\mathalpha}{\M@lower@id}{`v}
```

```
2807  \M@sym@{w}{\mathalpha}{\M@lower@id}{`w}
2808  \M@sym@{x}{\mathalpha}{\M@lower@id}{`x}
2809  \M@sym@{y}{\mathalpha}{\M@lower@id}{`y}
2810  \M@sym@{z}{\mathalpha}{\M@lower@id}{`z}
2811  \M@sym@{\imath}{\mathalpha}{\M@lower@id}{"131}
2812  \M@sym@{\jmath}{\mathalpha}{\M@lower@id}{"237}
2813  \let\hbar\@undefined
2814  \M@sym@{\hbar}{\mathord}{\M@lower@id}{"127}}
```

Set diacritics.

```
\M@diacritics@set 2815 \def\M@diacritics@set{
2816  \edef\M@diacritics@id{M\@tempc-\M@diacriticsshape}
2817  \M@acc@{\acute}   {\mathalpha}{\M@diacritics@id}{"B4}
2818  \M@acc@{\aacute}  {\mathalpha}{\M@diacritics@id}{"2DD}
2819  \M@acc@{\dot}     {\mathalpha}{\M@diacritics@id}{"2D9}
2820  \M@acc@{\ddot}    {\mathalpha}{\M@diacritics@id}{"A8}
2821  \M@acc@{\grave}   {\mathalpha}{\M@diacritics@id}{"60}
2822  \M@acc@{\breve}   {\mathalpha}{\M@diacritics@id}{"2D8}
2823  \M@acc@{\hat}     {\mathalpha}{\M@diacritics@id}{"2C6}
2824  \M@acc@{\check}   {\mathalpha}{\M@diacritics@id}{"2C7}
2825  \M@acc@{\bar}     {\mathalpha}{\M@diacritics@id}{"2C9}
2826  \M@acc@{\mathring}{\mathalpha}{\M@diacritics@id}{"2DA}
2827  \M@acc@{\tilde}   {\mathalpha}{\M@diacritics@id}{"2DC}}
```

Set capital Greek characters.

```
\M@greekupper@set 2828 \def\M@greekupper@set{
2829  \edef\M@greekupper@id{M\@tempc-\M@greekuppershape}
2830  \M@sym@{\Alpha}   {\mathalpha}{\M@greekupper@id}{"391}
2831  \M@sym@{\Beta}    {\mathalpha}{\M@greekupper@id}{"392}
2832  \M@sym@{\Gamma}   {\mathalpha}{\M@greekupper@id}{"393}
2833  \M@sym@{\Delta}   {\mathalpha}{\M@greekupper@id}{"394}
2834  \M@sym@{\Epsilon} {\mathalpha}{\M@greekupper@id}{"395}
2835  \M@sym@{\Zeta}    {\mathalpha}{\M@greekupper@id}{"396}
2836  \M@sym@{\Eta}     {\mathalpha}{\M@greekupper@id}{"397}
2837  \M@sym@{\Theta}   {\mathalpha}{\M@greekupper@id}{"398}
2838  \M@sym@{\Iota}    {\mathalpha}{\M@greekupper@id}{"399}
2839  \M@sym@{\Kappa}   {\mathalpha}{\M@greekupper@id}{"39A}
2840  \M@sym@{\Lambda}  {\mathalpha}{\M@greekupper@id}{"39B}
2841  \M@sym@{\Mu}      {\mathalpha}{\M@greekupper@id}{"39C}
2842  \M@sym@{\Nu}      {\mathalpha}{\M@greekupper@id}{"39D}
2843  \M@sym@{\Xi}      {\mathalpha}{\M@greekupper@id}{"39E}
2844  \M@sym@{\Omicron} {\mathalpha}{\M@greekupper@id}{"39F}
2845  \M@sym@{\Pi}      {\mathalpha}{\M@greekupper@id}{"3A0}
```

```
\M@sym@{\Rho}     {\mathalpha}{\M@greekupper@id}{"3A1}
\M@sym@{\Sigma}   {\mathalpha}{\M@greekupper@id}{"3A3}
\M@sym@{\Tau}     {\mathalpha}{\M@greekupper@id}{"3A4}
\M@sym@{\Upsilon} {\mathalpha}{\M@greekupper@id}{"3A5}
\M@sym@{\Phi}     {\mathalpha}{\M@greekupper@id}{"3A6}
\M@sym@{\Chi}     {\mathalpha}{\M@greekupper@id}{"3A7}
\M@sym@{\Psi}     {\mathalpha}{\M@greekupper@id}{"3A8}
\M@sym@{\Omega}   {\mathalpha}{\M@greekupper@id}{"3A9}
\M@sym@{\varTheta}{\mathalpha}{\M@greekupper@id}{"3F4}
```

Declare `\increment` and `\nabla` if they haven't already been declared in the `symbols` or `extsymbols` fonts.

```
\ifM@adjust@font
  \ifM@symbols\else
    \M@sym@{\increment}
      {\mathord}{\M@greekupper@id}{"2206}
    \M@sym@{\nabla}
      {\mathord}{\M@greekupper@id}{"2207}
  \fi
\else
  \ifM@symbols\else
    \M@sym@{\increment}
      {\mathord}{\M@greekupper@id}{"2206}
  \fi
  \ifM@extsymbols\else
    \M@sym@{\nabla}
      {\mathord}{\M@greekupper@id}{"2207}
  \fi
\fi}
```

Set minuscule Greek characters.

`\M@greeklower@set`

```
\def\M@greeklower@set{
  \edef\M@greeklower@id{M\@tempc-\M@greeklowershape}
  \M@sym@{\alpha}     {\mathalpha}{\M@greeklower@id}{"3B1}
  \M@sym@{\beta}      {\mathalpha}{\M@greeklower@id}{"3B2}
  \M@sym@{\gamma}     {\mathalpha}{\M@greeklower@id}{"3B3}
  \M@sym@{\delta}     {\mathalpha}{\M@greeklower@id}{"3B4}
  \M@sym@{\epsilon}   {\mathalpha}{\M@greeklower@id}{"3B5}
  \M@sym@{\zeta}      {\mathalpha}{\M@greeklower@id}{"3B6}
  \M@sym@{\eta}       {\mathalpha}{\M@greeklower@id}{"3B7}
  \M@sym@{\theta}     {\mathalpha}{\M@greeklower@id}{"3B8}
  \M@sym@{\iota}      {\mathalpha}{\M@greeklower@id}{"3B9}
  \M@sym@{\kappa}     {\mathalpha}{\M@greeklower@id}{"3BA}
```

```
2884  \M@sym@{\lambda}    {\mathalpha}{\M@greeklower@id}{"3BB}
2885  \M@sym@{\mu}        {\mathalpha}{\M@greeklower@id}{"3BC}
2886  \M@sym@{\nu}        {\mathalpha}{\M@greeklower@id}{"3BD}
2887  \M@sym@{\xi}        {\mathalpha}{\M@greeklower@id}{"3BE}
2888  \M@sym@{\omicron}   {\mathalpha}{\M@greeklower@id}{"3BF}
2889  \M@sym@{\pi}        {\mathalpha}{\M@greeklower@id}{"3C0}
2890  \M@sym@{\rho}       {\mathalpha}{\M@greeklower@id}{"3C1}
2891  \M@sym@{\sigma}     {\mathalpha}{\M@greeklower@id}{"3C3}
2892  \M@sym@{\tau}       {\mathalpha}{\M@greeklower@id}{"3C4}
2893  \M@sym@{\upsilon}   {\mathalpha}{\M@greeklower@id}{"3C5}
2894  \M@sym@{\phi}       {\mathalpha}{\M@greeklower@id}{"3C6}
2895  \M@sym@{\chi}       {\mathalpha}{\M@greeklower@id}{"3C7}
2896  \M@sym@{\psi}       {\mathalpha}{\M@greeklower@id}{"3C8}
2897  \M@sym@{\omega}     {\mathalpha}{\M@greeklower@id}{"3C9}
2898  \M@sym@{\varbeta}   {\mathalpha}{\M@greeklower@id}{"3D0}
2899  \M@sym@{\varepsilon}{\mathalpha}{\M@greeklower@id}{"3F5}
2900  \M@sym@{\varkappa}  {\mathalpha}{\M@greeklower@id}{"3F0}
2901  \M@sym@{\vartheta}  {\mathalpha}{\M@greeklower@id}{"3D1}
2902  \M@sym@{\varrho}    {\mathalpha}{\M@greeklower@id}{"3F1}
2903  \M@sym@{\varsigma}  {\mathalpha}{\M@greeklower@id}{"3C2}
2904  \M@sym@{\varphi}    {\mathalpha}{\M@greeklower@id}{"3D5}}
```

Set capital ancient Greek characters.

`\M@agreekupper@set`

```
2905 \def\M@agreekupper@set{
2906   \edef\M@agreekupper@id{M\@tempc-\M@agreekuppershape}
2907   \M@sym@{\Heta}      {\mathalpha}{\M@agreekupper@id}{"370}
2908   \M@sym@{\Sampi}     {\mathalpha}{\M@agreekupper@id}{"3E0}
2909   \M@sym@{\Digamma}   {\mathalpha}{\M@agreekupper@id}{"3DC}
2910   \M@sym@{\Koppa}     {\mathalpha}{\M@agreekupper@id}{"3D8}
2911   \M@sym@{\Stigma}    {\mathalpha}{\M@agreekupper@id}{"3DA}
2912   \M@sym@{\Sho}       {\mathalpha}{\M@agreekupper@id}{"3F7}
2913   \M@sym@{\San}       {\mathalpha}{\M@agreekupper@id}{"3FA}
2914   \M@sym@{\varSampi}  {\mathalpha}{\M@agreekupper@id}{"372}
2915   \M@sym@{\varDigamma}{\mathalpha}{\M@agreekupper@id}{"376}
2916   \M@sym@{\varKoppa}  {\mathalpha}{\M@agreekupper@id}{"3DE}}
```

Set minuscule ancient Greek characters.

`\M@agreeklower@set`

```
2917 \def\M@agreeklower@set{
2918   \edef\M@agreeklower@id{M\@tempc-\M@agreeklowershape}
2919   \M@sym@{\heta}      {\mathalpha}{\M@agreeklower@id}{"371}
2920   \M@sym@{\sampi}     {\mathalpha}{\M@agreeklower@id}{"3E1}
2921   \M@sym@{\digamma}   {\mathalpha}{\M@agreeklower@id}{"3DD}
2922   \M@sym@{\koppa}     {\mathalpha}{\M@agreeklower@id}{"3D9}
```

```
2923  \M@sym@{\stigma}    {\mathalpha}{\M@agreeklower@id}{"3DB}
2924  \M@sym@{\sho}       {\mathalpha}{\M@agreeklower@id}{"3F8}
2925  \M@sym@{\san}       {\mathalpha}{\M@agreeklower@id}{"3FB}
2926  \M@sym@{\varsampi}  {\mathalpha}{\M@agreeklower@id}{"373}
2927  \M@sym@{\vardigamma}{\mathalpha}{\M@agreeklower@id}{"377}
2928  \M@sym@{\varkoppa}  {\mathalpha}{\M@agreeklower@id}{"3DF}}
```

Set capital Cyrillic characters.

```
\M@cyrillicupper@s 2929 \def\M@cyrillicupper@set{
2930  \edef\M@cyrillicupper@id{M\@tempc-\M@cyrillicuppershape}
2931  \M@sym@{\cyrA}    {\mathalpha}{\M@cyrillicupper@id}{"410}
2932  \M@sym@{\cyrBe}   {\mathalpha}{\M@cyrillicupper@id}{"411}
2933  \M@sym@{\cyrVe}   {\mathalpha}{\M@cyrillicupper@id}{"412}
2934  \M@sym@{\cyrGhe}  {\mathalpha}{\M@cyrillicupper@id}{"413}
2935  \M@sym@{\cyrDe}   {\mathalpha}{\M@cyrillicupper@id}{"414}
2936  \M@sym@{\cyrIe}   {\mathalpha}{\M@cyrillicupper@id}{"415}
2937  \M@sym@{\cyrZhe}  {\mathalpha}{\M@cyrillicupper@id}{"416}
2938  \M@sym@{\cyrZe}   {\mathalpha}{\M@cyrillicupper@id}{"417}
2939  \M@sym@{\cyrI}    {\mathalpha}{\M@cyrillicupper@id}{"418}
2940  \M@sym@{\cyrKa}   {\mathalpha}{\M@cyrillicupper@id}{"41A}
2941  \M@sym@{\cyrEl}   {\mathalpha}{\M@cyrillicupper@id}{"41B}
2942  \M@sym@{\cyrEm}   {\mathalpha}{\M@cyrillicupper@id}{"41C}
2943  \M@sym@{\cyrEn}   {\mathalpha}{\M@cyrillicupper@id}{"41D}
2944  \M@sym@{\cyrO}    {\mathalpha}{\M@cyrillicupper@id}{"41E}
2945  \M@sym@{\cyrPe}   {\mathalpha}{\M@cyrillicupper@id}{"41F}
2946  \M@sym@{\cyrEr}   {\mathalpha}{\M@cyrillicupper@id}{"420}
2947  \M@sym@{\cyrEs}   {\mathalpha}{\M@cyrillicupper@id}{"421}
2948  \M@sym@{\cyrTe}   {\mathalpha}{\M@cyrillicupper@id}{"422}
2949  \M@sym@{\cyrU}    {\mathalpha}{\M@cyrillicupper@id}{"423}
2950  \M@sym@{\cyrEf}   {\mathalpha}{\M@cyrillicupper@id}{"424}
2951  \M@sym@{\cyrHa}   {\mathalpha}{\M@cyrillicupper@id}{"425}
2952  \M@sym@{\cyrTse}  {\mathalpha}{\M@cyrillicupper@id}{"426}
2953  \M@sym@{\cyrChe}  {\mathalpha}{\M@cyrillicupper@id}{"427}
2954  \M@sym@{\cyrSha}  {\mathalpha}{\M@cyrillicupper@id}{"428}
2955  \M@sym@{\cyrShcha}{\mathalpha}{\M@cyrillicupper@id}{"429}
2956  \M@sym@{\cyrHard} {\mathalpha}{\M@cyrillicupper@id}{"42A}
2957  \M@sym@{\cyrYeru} {\mathalpha}{\M@cyrillicupper@id}{"42B}
2958  \M@sym@{\cyrSoft} {\mathalpha}{\M@cyrillicupper@id}{"42C}
2959  \M@sym@{\cyrE}    {\mathalpha}{\M@cyrillicupper@id}{"42D}
2960  \M@sym@{\cyrYu}   {\mathalpha}{\M@cyrillicupper@id}{"42E}
2961  \M@sym@{\cyrYa}   {\mathalpha}{\M@cyrillicupper@id}{"42F}
2962  \M@sym@{\cyrvarI} {\mathalpha}{\M@cyrillicupper@id}{"419}}
```

Set minuscule Cyrillic characters.

```
\M@cyrilliclower@s 2963 \def\M@cyrilliclower@set{
                   2964   \edef\M@cyrilliclower@id{M\@tempc-\M@cyrilliclowershape}
                   2965   \M@sym@{\cyra}    {\mathalpha}{\M@cyrilliclower@id}{"430}
                   2966   \M@sym@{\cyrbe}   {\mathalpha}{\M@cyrilliclower@id}{"431}
                   2967   \M@sym@{\cyrve}   {\mathalpha}{\M@cyrilliclower@id}{"432}
                   2968   \M@sym@{\cyrghe}  {\mathalpha}{\M@cyrilliclower@id}{"433}
                   2969   \M@sym@{\cyrde}   {\mathalpha}{\M@cyrilliclower@id}{"434}
                   2970   \M@sym@{\cyrie}   {\mathalpha}{\M@cyrilliclower@id}{"435}
                   2971   \M@sym@{\cyrzhe}  {\mathalpha}{\M@cyrilliclower@id}{"436}
                   2972   \M@sym@{\cyrze}   {\mathalpha}{\M@cyrilliclower@id}{"437}
                   2973   \M@sym@{\cyri}    {\mathalpha}{\M@cyrilliclower@id}{"438}
                   2974   \M@sym@{\cyrka}   {\mathalpha}{\M@cyrilliclower@id}{"43A}
                   2975   \M@sym@{\cyrel}   {\mathalpha}{\M@cyrilliclower@id}{"43B}
                   2976   \M@sym@{\cyrem}   {\mathalpha}{\M@cyrilliclower@id}{"43C}
                   2977   \M@sym@{\cyren}   {\mathalpha}{\M@cyrilliclower@id}{"43D}
                   2978   \M@sym@{\cyro}    {\mathalpha}{\M@cyrilliclower@id}{"43E}
                   2979   \M@sym@{\cyrpe}   {\mathalpha}{\M@cyrilliclower@id}{"43F}
                   2980   \M@sym@{\cyrer}   {\mathalpha}{\M@cyrilliclower@id}{"440}
                   2981   \M@sym@{\cyres}   {\mathalpha}{\M@cyrilliclower@id}{"441}
                   2982   \M@sym@{\cyrte}   {\mathalpha}{\M@cyrilliclower@id}{"442}
                   2983   \M@sym@{\cyru}    {\mathalpha}{\M@cyrilliclower@id}{"443}
                   2984   \M@sym@{\cyref}   {\mathalpha}{\M@cyrilliclower@id}{"444}
                   2985   \M@sym@{\cyrha}   {\mathalpha}{\M@cyrilliclower@id}{"445}
                   2986   \M@sym@{\cyrtse}  {\mathalpha}{\M@cyrilliclower@id}{"446}
                   2987   \M@sym@{\cyrche}  {\mathalpha}{\M@cyrilliclower@id}{"447}
                   2988   \M@sym@{\cyrsha}  {\mathalpha}{\M@cyrilliclower@id}{"448}
                   2989   \M@sym@{\cyrshcha}{\mathalpha}{\M@cyrilliclower@id}{"449}
                   2990   \M@sym@{\cyrhard} {\mathalpha}{\M@cyrilliclower@id}{"44A}
                   2991   \M@sym@{\cyryeru} {\mathalpha}{\M@cyrilliclower@id}{"44B}
                   2992   \M@sym@{\cyrsoft} {\mathalpha}{\M@cyrilliclower@id}{"44C}
                   2993   \M@sym@{\cyre}    {\mathalpha}{\M@cyrilliclower@id}{"44D}
                   2994   \M@sym@{\cyryu}   {\mathalpha}{\M@cyrilliclower@id}{"44E}
                   2995   \M@sym@{\cyrya}   {\mathalpha}{\M@cyrilliclower@id}{"44F}
                   2996   \M@sym@{\cyrvari} {\mathalpha}{\M@cyrilliclower@id}{"439}}
```

Set Hebrew characters.

```
  \M@hebrew@set 2997 \def\M@hebrew@set{
                2998   \edef\M@hebrew@id{M\@tempc-\M@hebrewshape}
                2999   \M@sym@{\aleph}   {\mathalpha}{\M@hebrew@id}{"5D0}
                3000   \M@sym@{\beth}    {\mathalpha}{\M@hebrew@id}{"5D1}
                3001   \M@sym@{\gimel}   {\mathalpha}{\M@hebrew@id}{"5D2}
                3002   \M@sym@{\daleth}  {\mathalpha}{\M@hebrew@id}{"5D3}
                3003   \M@sym@{\he}      {\mathalpha}{\M@hebrew@id}{"5D4}
```

```
3004  \M@sym@{\vav}     {\mathalpha}{\M@hebrew@id}{"5D5}
3005  \M@sym@{\zayin}   {\mathalpha}{\M@hebrew@id}{"5D6}
3006  \M@sym@{\het}     {\mathalpha}{\M@hebrew@id}{"5D7}
3007  \M@sym@{\tet}     {\mathalpha}{\M@hebrew@id}{"5D8}
3008  \M@sym@{\yod}     {\mathalpha}{\M@hebrew@id}{"5D9}
3009  \M@sym@{\kaf}     {\mathalpha}{\M@hebrew@id}{"5DB}
3010  \M@sym@{\lamed}   {\mathalpha}{\M@hebrew@id}{"5DC}
3011  \M@sym@{\mem}     {\mathalpha}{\M@hebrew@id}{"5DE}
3012  \M@sym@{\nun}     {\mathalpha}{\M@hebrew@id}{"5E0}
3013  \M@sym@{\samekh}  {\mathalpha}{\M@hebrew@id}{"5E1}
3014  \M@sym@{\ayin}    {\mathalpha}{\M@hebrew@id}{"5E2}
3015  \M@sym@{\pe}      {\mathalpha}{\M@hebrew@id}{"5E4}
3016  \M@sym@{\tsadi}   {\mathalpha}{\M@hebrew@id}{"5E6}
3017  \M@sym@{\qof}     {\mathalpha}{\M@hebrew@id}{"5E7}
3018  \M@sym@{\resh}    {\mathalpha}{\M@hebrew@id}{"5E8}
3019  \M@sym@{\shin}    {\mathalpha}{\M@hebrew@id}{"5E9}
3020  \M@sym@{\tav}     {\mathalpha}{\M@hebrew@id}{"5EA}
3021  \M@sym@{\varkaf}  {\mathalpha}{\M@hebrew@id}{"5DA}
3022  \M@sym@{\varmem}  {\mathalpha}{\M@hebrew@id}{"5DD}
3023  \M@sym@{\varnun}  {\mathalpha}{\M@hebrew@id}{"5DF}
3024  \M@sym@{\varpe}   {\mathalpha}{\M@hebrew@id}{"5E3}
3025  \M@sym@{\vartsadi}{\mathalpha}{\M@hebrew@id}{"5E5}}
```

Set digits.

```
\M@digits@set 3026 \def\M@digits@set{
3027   \edef\M@digits@id{M\@tempc-\M@digitsshape}
3028   \M@sym@{0}{\mathalpha}{\M@digits@id}{`0}
3029   \M@sym@{1}{\mathalpha}{\M@digits@id}{`1}
3030   \M@sym@{2}{\mathalpha}{\M@digits@id}{`2}
3031   \M@sym@{3}{\mathalpha}{\M@digits@id}{`3}
3032   \M@sym@{4}{\mathalpha}{\M@digits@id}{`4}
3033   \M@sym@{5}{\mathalpha}{\M@digits@id}{`5}
3034   \M@sym@{6}{\mathalpha}{\M@digits@id}{`6}
3035   \M@sym@{7}{\mathalpha}{\M@digits@id}{`7}
3036   \M@sym@{8}{\mathalpha}{\M@digits@id}{`8}
3037   \M@sym@{9}{\mathalpha}{\M@digits@id}{`9}}
```

Set new operator font. We change the `\fam` to the user's requested symbol font for math operators.

```
\M@operator@set 3038 \def\M@operator@set{
 \operator@font 3039   \edef\operator@font{\mathgroup
3040     \csname symM\@tempc-\M@operatorshape\endcsname}}
```

Set delimiters.

```
\ifM@adjust@font
\M@delimiters@set  \def\M@delimiters@set{
    \edef\M@delimiters@id{M\@tempc-\M@delimitersshape}
    \edef\M@delimiters@num{
      \csname sym\M@delimiters@id\endcsname}
    \M@sym@{(}          {\mathopen} {\M@delimiters@id}{"28}
    \M@sym@{)}          {\mathclose}{\M@delimiters@id}{"29}
    \M@sym@{[}          {\mathopen} {\M@delimiters@id}{"5B}
    \M@sym@{]}          {\mathclose}{\M@delimiters@id}{"5D}
    \M@sym@{\leftbrace} {\mathopen} {\M@delimiters@id}{"7B}
    \M@sym@{\rightbrace}{\mathclose}{\M@delimiters@id}{"7D}
```

Set `\Udelcode`s for delimiters that come from individual characters.

```
    \Udelcode"28+\M@delimiters@num+"28\relax % (
    \Udelcode"29+\M@delimiters@num+"29\relax % )
    \Udelcode"2F+\M@delimiters@num+"2F\relax % /
    \Udelcode"5B+\M@delimiters@num+"5B\relax % [
    \Udelcode"5D+\M@delimiters@num+"5D\relax % ]
    \Udelcode"7C+\M@delimiters@num+"7C\relax % |
    \ifM@symbols\else
      \M@sym@{|}{\mathord}{\M@delimiters@id}{"7C}
    \fi
    \let\vert=|
```

For the delimiters that come from control sequences, we use `\edef` and `\Udelimiter`.

```
    \protected\def\backslash{
      \ifmmode\mathbackslash\else\textbackslash\fi}
    \protected\edef\mathbackslash{
      \Udelimiter+2+\M@delimiters@num+92\relax}
    \protected\edef\lbrace{
      \Udelimiter+4+\M@delimiters@num+123\relax}
    \protected\edef\rbrace{
      \Udelimiter+5+\M@delimiters@num+125\relax}
    \protected\edef\lguil{
      \Udelimiter+4+\M@delimiters@num+8249\relax}
    \protected\edef\rguil{
      \Udelimiter+5+\M@delimiters@num+8250\relax}
    \protected\edef\llguil{
      \Udelimiter+4+\M@delimiters@num+171\relax}
    \protected\edef\rrguil{
      \Udelimiter+5+\M@delimiters@num+187\relax}
    \protected\edef\fakelangle{
```

```
      \Udelimiter+4+\M@delimiters@num
        +\directlua{tex.print(mathfont.fakel)}\relax}
    \protected\edef\fakerangle{
      \Udelimiter+5+\M@delimiters@num
        +\directlua{tex.print(mathfont.faker)}\relax}
    \protected\edef\fakellangle{
      \Udelimiter+4+\M@delimiters@num
        +\directlua{tex.print(mathfont.fakell)}\relax}
    \protected\edef\fakerrangle{
      \Udelimiter+5+\M@delimiters@num
        +\directlua{tex.print(mathfont.fakerr)}\relax}}
\else
\M@delimiters@set  \def\M@delimiters@set{
    \edef\M@delimiters@id{M\@tempc-\M@delimitersshape}
    \M@sym@{(}          {\mathopen} {\M@delimiters@id}{"28}
    \M@sym@{)}          {\mathclose}{\M@delimiters@id}{"29}
    \M@sym@{[}          {\mathopen} {\M@delimiters@id}{"5B}
    \M@sym@{]}          {\mathclose}{\M@delimiters@id}{"5D}
    \M@sym@{\lguil}     {\mathopen} {\M@delimiters@id}{"2039}
    \M@sym@{\rguil}     {\mathclose}{\M@delimiters@id}{"203A}
    \M@sym@{\llguil}    {\mathopen} {\M@delimiters@id}{"AB}
    \M@sym@{\rrguil}    {\mathclose}{\M@delimiters@id}{"BB}
    \M@sym@{\leftbrace} {\mathopen} {\M@delimiters@id}{"7B}
    \M@sym@{\rightbrace}{\mathclose}{\M@delimiters@id}{"7D}}
\fi
```

Radicals. When we define `\surd` to typeset U+221A, `\M@sym@` sets the `\Umathcode` of √ to be a surd symbol. However, if we modified the font, we know the surd character in the requested font can successfully make a square root expression, so we override the definition from `\M@sym@` to turn √ to an active character in math mode.

```
\ifM@adjust@font
\M@radical@set  \def\M@radical@set{
    \edef\M@radical@id{M\@tempc-\M@radicalshape}
      \surd     \let\surd\@undefined
    \M@sym@{\surd}{\mathord}{\M@radical@id}{"221A}
```

Now set the `\mathcode` of √ to 8000. This is probably me being paranoid, but I wanted to stick to ascii characters in the sty file. We use `\directlua` to print the surd character instead. This also has the advantage of printing an active character, so we don't have to scan any tokens.

```
    \expandafter\protected\expandafter\def\directlua{
      tex.cprint(13, utf8.char(0x221A))}
```

```
3111      {\ifmmode\expandafter\sqrt\else\Uchar"221A\relax\fi}
3112    \mathcode"221A="8000\relax
\@sqrts@gn 3113    \edef\@sqrts@gn##1{\Uradical+\number
3114      \csname sym\M@radical@id\endcsname+"221A\relax{##1}}
```

We redefine `\r@@t`, which typesets the degree symbol on an $n$th root. We set the placement so that right side of the box containing the degree lies 60% of the horizontal distance across the surd symbol, and the baseline of the degree symbol is 60% of the vertical distance up the surd.

```
\r@@t 3115    \def\r@@t##1##2{
3116      \setbox\z@\hbox{$\m@th##1\sqrtsign{##2}$}
3117      \setbox\surdbox\hbox{$\m@th##1\@sqrts@gn{
3118        \hbox{\vphantom{$\m@th##1##2$}}}$}
3119      \dimen@\ht\surdbox
3120      \advance\dimen@\dp\surdbox
3121      \dimen@=0.6\dimen@
3122      \advance\dimen@-\dp\surdbox
3123      \ifdim\wd\rootbox<0.6\wd\surdbox
3124        \kern0.6\wd\surdbox
3125      \else
3126        \kern\wd\rootbox
3127      \fi
3128      \raise\dimen@\hbox{\llap{\copy\rootbox}}
3129      \kern-0.6\wd\surdbox
3130      \box\z@}
\sqrtsign 3131    \protected\def\sqrtsign##1{
3132      \@sqrts@gn{\mkern\radicandoffset##1}}}
3133 \else
\M@radical@set 3134   \def\M@radical@set{
3135     \edef\M@radical@id{M\@tempc-\M@radicalshape}
\surd 3136     \let\surd\@undefined
3137     \M@sym@{\surd}{\mathord}{\M@radical@id}{"221A}}
3138 \fi
```

Big operators.

```
\M@bigops@set 3139 \def\M@bigops@set{
3140   \edef\M@bigops@id{M\@tempc-\M@bigopsshape}
3141   \let\sum\@undefined
3142   \let\prod\@undefined
3143   \M@sym@{\sum}  {\mathop}{\M@bigops@id}{"2211}
3144   \M@sym@{\prod} {\mathop}{\M@bigops@id}{"220F}
3145   \M@sym@{\intop}{\mathop}{\M@bigops@id}{"222B}}
```

Extended big operators.

```
\M@extbigops@set 3146 \def\M@extbigops@set{
                 3147   \edef\M@extbigops@id{M\@tempc-\M@extbigopsshape}
                 3148   \let\coprod\@undefined
                 3149   \let\bigvee\@undefined
                 3150   \let\bigwedge\@undefined
                 3151   \let\bigcup\@undefined
                 3152   \let\bigcap\@undefined
                 3153   \let\bigoplus\@undefined
                 3154   \let\bigotimes\@undefined
                 3155   \let\bigodot\@undefined
                 3156   \let\bigsqcup\@undefined
                 3157   \M@sym@{\coprod}   {\mathop}{\M@extbigops@id}{"2210}
                 3158   \M@sym@{\bigvee}   {\mathop}{\M@extbigops@id}{"22C1}
                 3159   \M@sym@{\bigwedge} {\mathop}{\M@extbigops@id}{"22C0}
                 3160   \M@sym@{\bigcup}   {\mathop}{\M@extbigops@id}{"22C3}
                 3161   \M@sym@{\bigcap}   {\mathop}{\M@extbigops@id}{"22C2}
                 3162   \M@sym@{\iintop}   {\mathop}{\M@extbigops@id}{"222C}
                 3163   \M@sym@{\iiintop}  {\mathop}{\M@extbigops@id}{"222D}
                 3164   \M@sym@{\ointop}   {\mathop}{\M@extbigops@id}{"222E}
                 3165   \M@sym@{\oiintop}  {\mathop}{\M@extbigops@id}{"222F}
                 3166   \M@sym@{\oiiintop} {\mathop}{\M@extbigops@id}{"2230}
                 3167   \M@sym@{\bigoplus} {\mathop}{\M@extbigops@id}{"2A01}
                 3168   \M@sym@{\bigotimes}{\mathop}{\M@extbigops@id}{"2A02}
                 3169   \M@sym@{\bigodot}  {\mathop}{\M@extbigops@id}{"2A00}
                 3170   \M@sym@{\bigsqcap} {\mathop}{\M@extbigops@id}{"2A05}
                 3171   \M@sym@{\bigsqcup} {\mathop}{\M@extbigops@id}{"2A06}
                 3172   \protected\def\iint{\iintop\nolimits}
                 3173   \protected\def\iiint{\iiintop\nolimits}
                 3174   \protected\def\oint{\ointop\nolimits}
                 3175   \protected\def\oiint{\oiintop\nolimits}
                 3176   \protected\def\oiiint{\oiiintop\nolimits}}
```

Set symbols.

```
\M@symbols@set 3177 \def\M@symbols@set{
               3178   \edef\M@symbols@id{M\@tempc-\M@symbolsshape}
               3179   \let\colon\@undefined
               3180   \let\mathellipsis\@undefined
```

Before we start declaring symbols, specifically minus or equals signs, we have to address a minor clash with amsmath. That package defines `\relbar` and `\Relbar` as essentially a minus and equals sign respectively. However, those two control sequences are for making arrows, so they should come from the `arrows` font, not the `symbols` font. If the user already called `\mathfont` with

the `arrows` keyword, we do nothing because `\M@arrows@set` defines `\relbar` and `\Relbar` correctly. If not, we make these two control sequences be the current minus and equals sign (before the font changes in `\M@symbols@set`) because that's as good a choice as any, and we prevent amsmath from changing them to the `symbols` font. Users or package authors who want to modify `\relbar` or `\Relbar` should change `\@relbar` or `\@Relbar` respectively.

```
\ifM@arrows\else
  \Umathcharnumdef\@relbar=\Umathcodenum`\-
  \Umathcharnumdef\@Relbar=\Umathcodenum`\=
  \protected\def\relbar{\mathrel
    {\mathpalette\mathsm@sh\@relbar}}
  \protected\def\Relbar{\@Relbar}
```

We redefine stuff if amsmath gets loaded after mathfont.

```
  \@ifpackageloaded{amsmath}
    {\relax}{
      \let\@@relbar\relbar
      \let\@@Relbar\Relbar
      \AtBeginDocument{\ifM@arrows\else
          \@ifpackageloaded{amsmath}{
            \let\relbar\@@relbar
            \let\Relbar\@@Relbar}
          {\relax}
        \fi}}
\fi
```

If the user enabled Lua-based font asjustments, we declare a few more big operators for fun. For brevity, we put the `adjust@font` conditional here rather than redefining `\M@symbols@set`. Apparently, newtx defines `\bigtimes`, so in case that package gets loaded ahead of mathfont, we should make sure to clear that definition. It's important to declare the big operators before the normal versions of these characters so that `\M@sym@` defines the correct `\Umathcode` for them.

```
\ifM@adjust@font
  \let\bigtimes\@undefined
  \M@sym@{\bigat}     {\mathop}{\M@symbols@id}{"40}
  \M@sym@{\bighash}   {\mathop}{\M@symbols@id}{"23}
  \M@sym@{\bigdollar} {\mathop}{\M@symbols@id}{"24}
  \M@sym@{\bigpercent}{\mathop}{\M@symbols@id}{"25}
  \M@sym@{\bigand}    {\mathop}{\M@symbols@id}{"26}
  \M@sym@{\bigplus}   {\mathop}{\M@symbols@id}{"2B}
  \M@sym@{\bigp}      {\mathop}{\M@symbols@id}{"21}
  \M@sym@{\bigq}      {\mathop}{\M@symbols@id}{"3F}
```

```
    \M@sym@{\bigS}      {\mathop}{\M@symbols@id}{"A7}
    \M@sym@{\bigtimes}  {\mathop}{\M@symbols@id}{"D7}
    \M@sym@{\bigdiv}    {\mathop}{\M@symbols@id}{"F7}
```

Define `\nabla` here if we're adjusting the font. If we are not doing that, this declaration goes in `extsymbols`.

```
    \M@sym@{\nabla}     {\mathord}{\M@symbols@id}{"2207}
  \fi
```

The rest of the symbols.

```
  \M@sym@{.}              {\mathord}  {\M@symbols@id}{"2E}
  \M@sym@{@}              {\mathord}  {\M@symbols@id}{"40}
  \M@sym@{'}              {\mathord}  {\M@symbols@id}{"2032}
  \M@sym@{\prime}         {\mathord}  {\M@symbols@id}{"2032}
  \M@sym@{"}              {\mathord}  {\M@symbols@id}{"2033}
  \M@sym@{\mathhash}      {\mathord}  {\M@symbols@id}{"23}
  \M@sym@{\mathdollar}    {\mathord}  {\M@symbols@id}{"24}
  \M@sym@{\mathpercent}   {\mathord}  {\M@symbols@id}{"25}
  \M@sym@{\mathand}       {\mathord}  {\M@symbols@id}{"26}
  \M@sym@{\mathparagraph}{\mathord}  {\M@symbols@id}{"B6}
  \M@sym@{\mathsection}   {\mathord}  {\M@symbols@id}{"A7}
  \let\mathsterling\@undefined
  \M@sym@{\mathsterling} {\mathord}  {\M@symbols@id}{"A3}
  \M@sym@{\neg}           {\mathord}  {\M@symbols@id}{"AC}
  \M@sym@{\mid}           {\mathrel}  {\M@symbols@id}{"7C}
  \M@sym@{|}              {\mathord}  {\M@symbols@id}{"7C}
  \M@sym@{\infty}         {\mathord}  {\M@symbols@id}{"221E}
  \M@sym@{\partial}       {\mathord}  {\M@symbols@id}{"2202}
  \M@sym@{\degree}        {\mathord}  {\M@symbols@id}{"B0}
  \M@sym@{\increment}     {\mathord}  {\M@symbols@id}{"2206}
  \M@sym@{+}              {\mathbin}  {\M@symbols@id}{"2B}
  \M@sym@{-}              {\mathbin}  {\M@symbols@id}{"2212}
  \M@sym@{*}              {\mathbin}  {\M@symbols@id}{"2A}
  \M@sym@{\times}         {\mathbin}  {\M@symbols@id}{"D7}
  \M@sym@{/}              {\mathord}  {\M@symbols@id}{"2F}
  \M@sym@{\fractionslash}{\mathord}  {\M@symbols@id}{"2215}
  \M@sym@{\div}           {\mathbin}  {\M@symbols@id}{"F7}
  \M@sym@{\pm}            {\mathbin}  {\M@symbols@id}{"B1}
  \M@sym@{\bullet}        {\mathbin}  {\M@symbols@id}{"2022}
  \M@sym@{\dagger}        {\mathbin}  {\M@symbols@id}{"2020}
  \M@sym@{\ddagger}       {\mathbin}  {\M@symbols@id}{"2021}
  \M@sym@{\cdot}          {\mathbin}  {\M@symbols@id}{"2219}
  \M@sym@{\setminus}      {\mathbin}  {\M@symbols@id}{"5C}
```

```
\M@sym@{=}              {\mathrel}  {\M@symbols@id}{"3D}
\M@sym@{<}              {\mathrel}  {\M@symbols@id}{"3C}
\M@sym@{>}              {\mathrel}  {\M@symbols@id}{"3E}
\M@sym@{\leq}           {\mathrel}  {\M@symbols@id}{"2264}
\M@sym@{\geq}           {\mathrel}  {\M@symbols@id}{"2265}
\M@sym@{\sim}           {\mathrel}  {\M@symbols@id}{"7E}
\M@sym@{\approx}        {\mathrel}  {\M@symbols@id}{"2248}
\M@sym@{\equiv}         {\mathrel}  {\M@symbols@id}{"2261}
\M@sym@{\parallel}      {\mathrel}  {\M@symbols@id}{"2016}
\M@sym@{\colon}         {\mathpunct}{\M@symbols@id}{"3A}
\M@sym@{:}              {\mathrel}  {\M@symbols@id}{"3A}
\M@sym@{?}              {\mathclose}{\M@symbols@id}{"3F}
\M@sym@{!}              {\mathclose}{\M@symbols@id}{"21}
\M@sym@{\comma}         {\mathord}  {\M@symbols@id}{"2C}
\M@sym@{,}              {\mathpunct}{\M@symbols@id}{"2C}
\M@sym@{;}              {\mathpunct}{\M@symbols@id}{"3B}
\M@sym@{\mathellipsis} {\mathinner}{\M@symbols@id}{"2026}
```

Now a bit of housekeeping. We redefine `\#`, `\%`, and `\&` as `\protected` macros that expand to previously declared `\mathhash`, etc. commands in math mode and retain their standard `\char` definitions otherwise. Other commands that function in both math and horizontal modes such as `\S` or `\dag` also use this technique. Then we define macros `\cong` and `\simeq` if the user hasn't called `\mathfont` with `extsymbols`.

```
\protected\def\#{\ifmmode\mathhash\else\char"23\relax\fi}
\protected\def\%{\ifmmode\mathpercent\else\char"25\relax\fi}
\protected\def\&{\ifmmode\mathand\else\char"26\relax\fi}
\ifM@extsymbols\else
  \protected\def\simeq{
    \mathrel{\mathpalette\stack@flatrel{{-}{\sim}}}}
  \protected\def\cong{
    \mathrel{\mathpalette\stack@flatrel{{=}{\sim}}}}
\fi
\protected\def\models{\mathrel{|}\joinrel\mathrel{=}}
```

`\models`

New definition for `\not`. We define it to accept a `#1` argument, which we store in an `\hbox` in the appropriate style. Then we typeset a `/` halfway across the distance of the `\hbox` and the `\hbox` itself. This approach manually positions the `/` halfway across the `#1` subformula instead of using a character that appears to the right of a slim bounding box as in traditional TEX. In case any users want to access the old `\not` definition, we save it as `\negslash`.

```
\let\negslash\not
\protected\def\not##1{\mathrel{\mathchoice
```

`\negslash`

```
    {\setbox\@tempboxa\hbox{$\displaystyle##1\m@th$}
      \hbox{\hb@xt@\wd\@tempboxa{\hss$\displaystyle/\m@th$\hss}
        \llap{\box\@tempboxa}}}
    {\setbox\@tempboxa\hbox{$\textstyle##1\m@th$}
      \hbox{\hb@xt@\wd\@tempboxa{\hss$\textstyle/\m@th$\hss}
        \llap{\box\@tempboxa}}}
    {\setbox\@tempboxa\hbox{$\scriptstyle##1\m@th$}
      \hbox{\hb@xt@\wd\@tempboxa{\hss$\scriptstyle/\m@th$\hss}
        \llap{\box\@tempboxa}}}
    {\setbox\@tempboxa\hbox{$\scriptscriptstyle##1\m@th$}
      \hbox{\hb@xt@\wd\@tempboxa{\hss{$\scriptscriptstyle/
          \m@th$\hss}
        \llap{\box\@tempboxa}}}}}}}
```

Set extended symbols.

\M@extsymbols@set

```
\def\M@extsymbols@set{
  \edef\M@extsymbols@id{M\@tempc-\M@extsymbolsshape}
  \let\angle\@undefined
  \let\simeq\@undefined
  \let\sqsubset\@undefined
  \let\sqsupset\@undefined
  \let\bowtie\@undefined
  \let\doteq\@undefined
  \let\neq\@undefined
  \M@sym@{\wp}              {\mathord}{\M@extsymbols@id}{"2118}
  \M@sym@{\ell}             {\mathord}{\M@extsymbols@id}{"2113}
  \M@sym@{\forall}          {\mathord}{\M@extsymbols@id}{"2200}
  \M@sym@{\exists}          {\mathord}{\M@extsymbols@id}{"2203}
  \M@sym@{\emptyset}        {\mathord}{\M@extsymbols@id}{"2205}
  \M@sym@{\in}              {\mathord}{\M@extsymbols@id}{"2208}
  \M@sym@{\ni}              {\mathord}{\M@extsymbols@id}{"220B}
  \M@sym@{\mp}              {\mathord}{\M@extsymbols@id}{"2213}
  \M@sym@{\angle}           {\mathord}{\M@extsymbols@id}{"2220}
  \M@sym@{\top}             {\mathord}{\M@extsymbols@id}{"22A4}
  \M@sym@{\bot}             {\mathord}{\M@extsymbols@id}{"22A5}
  \M@sym@{\vdash}           {\mathord}{\M@extsymbols@id}{"22A2}
  \M@sym@{\dashv}           {\mathord}{\M@extsymbols@id}{"22A3}
  \M@sym@{\flat}            {\mathord}{\M@extsymbols@id}{"266D}
  \M@sym@{\natural}         {\mathord}{\M@extsymbols@id}{"266E}
  \M@sym@{\sharp}           {\mathord}{\M@extsymbols@id}{"266F}
  \M@sym@{\fflat}           {\mathord}{\M@extsymbols@id}{"1D12B}
  \M@sym@{\ssharp}          {\mathord}{\M@extsymbols@id}{"1D12A}
  \M@sym@{\bclubsuit}       {\mathord}{\M@extsymbols@id}{"2663}
```

```
\M@sym@{\bdiamondsuit}    {\mathord}{\M@extsymbols@id}{"2666}
\M@sym@{\bheartsuit}      {\mathord}{\M@extsymbols@id}{"2665}
\M@sym@{\bspadesuit}      {\mathord}{\M@extsymbols@id}{"2660}
\M@sym@{\wclubsuit}       {\mathord}{\M@extsymbols@id}{"2667}
\M@sym@{\wdiamondsuit}    {\mathord}{\M@extsymbols@id}{"2662}
\M@sym@{\wheartsuit}      {\mathord}{\M@extsymbols@id}{"2661}
\M@sym@{\wspadesuit}      {\mathord}{\M@extsymbols@id}{"2664}
  \let\spadesuit\bspadesuit
  \let\heartsuit\wheartsuit
  \let\diamondsuit\wdiamondsuit
  \let\clubsuit\bclubsuit
\M@sym@{\wedge}           {\mathbin}{\M@extsymbols@id}{"2227}
\M@sym@{\vee}             {\mathbin}{\M@extsymbols@id}{"2228}
\M@sym@{\cap}             {\mathord}{\M@extsymbols@id}{"2229}
\M@sym@{\cup}             {\mathbin}{\M@extsymbols@id}{"222A}
\M@sym@{\sqcap}           {\mathbin}{\M@extsymbols@id}{"2293}
\M@sym@{\sqcup}           {\mathbin}{\M@extsymbols@id}{"2294}
\M@sym@{\amalg}           {\mathbin}{\M@extsymbols@id}{"2A3F}
\M@sym@{\wr}              {\mathbin}{\M@extsymbols@id}{"2240}
\M@sym@{\ast}             {\mathbin}{\M@extsymbols@id}{"2217}
\M@sym@{\star}            {\mathbin}{\M@extsymbols@id}{"22C6}
\M@sym@{\diamond}         {\mathbin}{\M@extsymbols@id}{"22C4}
\M@sym@{\varcdot}         {\mathbin}{\M@extsymbols@id}{"22C5}
\M@sym@{\varsetminus}     {\mathbin}{\M@extsymbols@id}{"2216}
\M@sym@{\oplus}           {\mathbin}{\M@extsymbols@id}{"2295}
\M@sym@{\otimes}          {\mathbin}{\M@extsymbols@id}{"2297}
\M@sym@{\ominus}          {\mathbin}{\M@extsymbols@id}{"2296}
\M@sym@{\odiv}            {\mathbin}{\M@extsymbols@id}{"2A38}
\M@sym@{\oslash}          {\mathbin}{\M@extsymbols@id}{"2298}
\M@sym@{\odot}            {\mathbin}{\M@extsymbols@id}{"2299}
\M@sym@{\sqplus}          {\mathbin}{\M@extsymbols@id}{"229E}
\M@sym@{\sqtimes}         {\mathbin}{\M@extsymbols@id}{"22A0}
\M@sym@{\sqminus}         {\mathbin}{\M@extsymbols@id}{"229F}
\M@sym@{\sqdot}           {\mathbin}{\M@extsymbols@id}{"22A1}
\M@sym@{\in}              {\mathrel}{\M@extsymbols@id}{"2208}
\M@sym@{\ni}              {\mathrel}{\M@extsymbols@id}{"220B}
\M@sym@{\subset}          {\mathrel}{\M@extsymbols@id}{"2282}
\M@sym@{\supset}          {\mathrel}{\M@extsymbols@id}{"2283}
\M@sym@{\subseteq}        {\mathrel}{\M@extsymbols@id}{"2286}
\M@sym@{\supseteq}        {\mathrel}{\M@extsymbols@id}{"2287}
\M@sym@{\sqsubset}        {\mathrel}{\M@extsymbols@id}{"228F}
\M@sym@{\sqsupset}        {\mathrel}{\M@extsymbols@id}{"2290}
```

```
\M@sym@{\sqsubseteq}      {\mathrel}{\M@extsymbols@id}{"2291}
\M@sym@{\sqsupseteq}      {\mathrel}{\M@extsymbols@id}{"2292}
\M@sym@{\triangleleft}    {\mathrel}{\M@extsymbols@id}{"22B2}
\M@sym@{\triangleright}   {\mathrel}{\M@extsymbols@id}{"22B3}
\M@sym@{\trianglelefteq}  {\mathrel}{\M@extsymbols@id}{"22B4}
\M@sym@{\trianglerighteq} {\mathrel}{\M@extsymbols@id}{"22B5}
\M@sym@{\propto}          {\mathrel}{\M@extsymbols@id}{"221D}
\M@sym@{\bowtie}          {\mathrel}{\M@extsymbols@id}{"22C8}
\M@sym@{\hourglass}       {\mathrel}{\M@extsymbols@id}{"29D6}
\M@sym@{\therefore}       {\mathrel}{\M@extsymbols@id}{"2234}
\M@sym@{\because}         {\mathrel}{\M@extsymbols@id}{"2235}
\M@sym@{\ratio}           {\mathrel}{\M@extsymbols@id}{"2236}
\M@sym@{\proportion}      {\mathrel}{\M@extsymbols@id}{"2237}
\M@sym@{\ll}              {\mathrel}{\M@extsymbols@id}{"226A}
\M@sym@{\gg}              {\mathrel}{\M@extsymbols@id}{"226B}
\M@sym@{\lll}             {\mathrel}{\M@extsymbols@id}{"22D8}
\M@sym@{\ggg}             {\mathrel}{\M@extsymbols@id}{"22D9}
\M@sym@{\leqq}            {\mathrel}{\M@extsymbols@id}{"2266}
\M@sym@{\geqq}            {\mathrel}{\M@extsymbols@id}{"2267}
\M@sym@{\lapprox}         {\mathrel}{\M@extsymbols@id}{"2A85}
\M@sym@{\gapprox}         {\mathrel}{\M@extsymbols@id}{"2A86}
\M@sym@{\simeq}           {\mathrel}{\M@extsymbols@id}{"2243}
\M@sym@{\eqsim}           {\mathrel}{\M@extsymbols@id}{"2242}
\M@sym@{\simeqq}          {\mathrel}{\M@extsymbols@id}{"2245}
  \let\cong\simeqq
\M@sym@{\approxeq}        {\mathrel}{\M@extsymbols@id}{"224A}
\M@sym@{\sssim}           {\mathrel}{\M@extsymbols@id}{"224B}
\M@sym@{\seq}             {\mathrel}{\M@extsymbols@id}{"224C}
\M@sym@{\doteq}           {\mathrel}{\M@extsymbols@id}{"2250}
\M@sym@{\coloneq}         {\mathrel}{\M@extsymbols@id}{"2254}
\M@sym@{\eqcolon}         {\mathrel}{\M@extsymbols@id}{"2255}
\M@sym@{\ringeq}          {\mathrel}{\M@extsymbols@id}{"2257}
\M@sym@{\arceq}           {\mathrel}{\M@extsymbols@id}{"2258}
\M@sym@{\wedgeeq}         {\mathrel}{\M@extsymbols@id}{"2259}
\M@sym@{\veeeq}           {\mathrel}{\M@extsymbols@id}{"225A}
\M@sym@{\stareq}          {\mathrel}{\M@extsymbols@id}{"225B}
\M@sym@{\triangleeq}      {\mathrel}{\M@extsymbols@id}{"225C}
\M@sym@{\defeq}           {\mathrel}{\M@extsymbols@id}{"225D}
\M@sym@{\qeq}             {\mathrel}{\M@extsymbols@id}{"225F}
\M@sym@{\lsim}            {\mathrel}{\M@extsymbols@id}{"2272}
\M@sym@{\gsim}            {\mathrel}{\M@extsymbols@id}{"2273}
\M@sym@{\prec}            {\mathrel}{\M@extsymbols@id}{"227A}
```

```
\M@sym@{\succ}            {\mathrel}{\M@extsymbols@id}{"227B}
\M@sym@{\preceq}          {\mathrel}{\M@extsymbols@id}{"227C}
\M@sym@{\succeq}          {\mathrel}{\M@extsymbols@id}{"227D}
\M@sym@{\preceqq}         {\mathrel}{\M@extsymbols@id}{"2AB3}
\M@sym@{\succeqq}         {\mathrel}{\M@extsymbols@id}{"2AB4}
\M@sym@{\precsim}         {\mathrel}{\M@extsymbols@id}{"227E}
\M@sym@{\succsim}         {\mathrel}{\M@extsymbols@id}{"227F}
\M@sym@{\precapprox}      {\mathrel}{\M@extsymbols@id}{"2AB7}
\M@sym@{\succapprox}      {\mathrel}{\M@extsymbols@id}{"2AB8}
\M@sym@{\precprec}        {\mathrel}{\M@extsymbols@id}{"2ABB}
\M@sym@{\succsucc}        {\mathrel}{\M@extsymbols@id}{"2ABC}
\M@sym@{\asymp}           {\mathrel}{\M@extsymbols@id}{"224D}
\M@sym@{\nin}             {\mathrel}{\M@extsymbols@id}{"2209}
\M@sym@{\nni}             {\mathrel}{\M@extsymbols@id}{"220C}
\M@sym@{\nsubset}         {\mathrel}{\M@extsymbols@id}{"2284}
\M@sym@{\nsupset}         {\mathrel}{\M@extsymbols@id}{"2285}
\M@sym@{\nsubseteq}       {\mathrel}{\M@extsymbols@id}{"2288}
\M@sym@{\nsupseteq}       {\mathrel}{\M@extsymbols@id}{"2289}
\M@sym@{\subsetneq}       {\mathrel}{\M@extsymbols@id}{"228A}
\M@sym@{\supsetneq}       {\mathrel}{\M@extsymbols@id}{"228B}
\M@sym@{\nsqsubseteq}     {\mathrel}{\M@extsymbols@id}{"22E2}
\M@sym@{\nsqsupseteq}     {\mathrel}{\M@extsymbols@id}{"22E3}
\M@sym@{\sqsubsetneq}     {\mathrel}{\M@extsymbols@id}{"22E4}
\M@sym@{\sqsupsetneq}     {\mathrel}{\M@extsymbols@id}{"22E5}
\M@sym@{\neq}             {\mathrel}{\M@extsymbols@id}{"2260}
\M@sym@{\nl}              {\mathrel}{\M@extsymbols@id}{"226E}
\M@sym@{\nleq}            {\mathrel}{\M@extsymbols@id}{"2270}
\M@sym@{\ngeq}            {\mathrel}{\M@extsymbols@id}{"2271}
\M@sym@{\lneq}            {\mathrel}{\M@extsymbols@id}{"2A87}
\M@sym@{\gneq}            {\mathrel}{\M@extsymbols@id}{"2A88}
\M@sym@{\lneqq}           {\mathrel}{\M@extsymbols@id}{"2268}
\M@sym@{\gneqq}           {\mathrel}{\M@extsymbols@id}{"2269}
\M@sym@{\ntriangleleft}   {\mathrel}{\M@extsymbols@id}{"22EA}
\M@sym@{\ntriangleright}  {\mathrel}{\M@extsymbols@id}{"22EB}
\M@sym@{\ntrianglelefteq} {\mathrel}{\M@extsymbols@id}{"22EC}
\M@sym@{\ntrianglerighteq}{\mathrel}{\M@extsymbols@id}{"22ED}
\M@sym@{\nsim}            {\mathrel}{\M@extsymbols@id}{"2241}
\M@sym@{\napprox}         {\mathrel}{\M@extsymbols@id}{"2249}
\M@sym@{\nsimeq}          {\mathrel}{\M@extsymbols@id}{"2244}
\M@sym@{\nsimeqq}         {\mathrel}{\M@extsymbols@id}{"2247}
\M@sym@{\simneqq}         {\mathrel}{\M@extsymbols@id}{"2246}
\M@sym@{\nlsim}           {\mathrel}{\M@extsymbols@id}{"2274}
```

```
\M@sym@{\ngsim}           {\mathrel}{\M@extsymbols@id}{"2275}
\M@sym@{\lnsim}           {\mathrel}{\M@extsymbols@id}{"22E6}
\M@sym@{\gnsim}           {\mathrel}{\M@extsymbols@id}{"22E7}
\M@sym@{\lnapprox}        {\mathrel}{\M@extsymbols@id}{"2A89}
\M@sym@{\gnapprox}        {\mathrel}{\M@extsymbols@id}{"2A8A}
\M@sym@{\nprec}           {\mathrel}{\M@extsymbols@id}{"2280}
\M@sym@{\nsucc}           {\mathrel}{\M@extsymbols@id}{"2281}
\M@sym@{\npreceq}         {\mathrel}{\M@extsymbols@id}{"22E0}
\M@sym@{\nsucceq}         {\mathrel}{\M@extsymbols@id}{"22E1}
\M@sym@{\precneq}         {\mathrel}{\M@extsymbols@id}{"2AB1}
\M@sym@{\succneq}         {\mathrel}{\M@extsymbols@id}{"2AB2}
\M@sym@{\precneqq}        {\mathrel}{\M@extsymbols@id}{"2AB5}
\M@sym@{\succneqq}        {\mathrel}{\M@extsymbols@id}{"2AB6}
\M@sym@{\precnsim}        {\mathrel}{\M@extsymbols@id}{"22E8}
\M@sym@{\succnsim}        {\mathrel}{\M@extsymbols@id}{"22E9}
\M@sym@{\precnapprox}     {\mathrel}{\M@extsymbols@id}{"2AB9}
\M@sym@{\succnapprox}     {\mathrel}{\M@extsymbols@id}{"2ABA}
\M@sym@{\nequiv}          {\mathrel}{\M@extsymbols@id}{"2262}
```

The math-operator package renames `\Re` and `\Im` to `\varRe` and `\varIm`. To make mathfont compatible with that package, we test whether these macros contain `mathchar` in their definitions before redefining. First `\Re`.

```
\expanded{\noexpand\in@{\expandafter\@gobble\string\mathchar}
  {\meaning\Re}}
\ifin@
  \M@sym@{\Re}{\mathord}{\M@extsymbols@id}{"211C}
\else
  \expanded{\noexpand\in@{\expandafter\@gobble\string\mathchar}
    {\meaning\varRe}}
  \ifin@
    \M@sym@{\varRe}{\mathord}{\M@extsymbols@id}{"211C}
  \fi
\fi
```

And `\Im`.

```
\expanded{\noexpand\in@{\expandafter\@gobble\string\mathchar}
  {\meaning\Im}}
\ifin@
  \M@sym@{\Im}{\mathord}{\M@extsymbols@id}{"2111}
\else
  \expanded{\noexpand\in@{\expandafter\@gobble\string\mathchar}
    {\meaning\varIm}}
  \ifin@
```

```
      \M@sym@{\varIm}{\mathord}{\M@extsymbols@id}{"2111}
    \fi
  \fi
```

We handle \ng specially. The LaTeX kernel defines \ng as a text symbol, so we define \mathng like for \$, etc.

```
  \let\textng\ng
  \M@sym@{\mathng}{\mathrel}{\M@extsymbols@id}{"226F}
  \protected\def\ng{\ifmmode\mathng\else\textng\fi}
```

If we're not adjusting the font, we declare \nabla here.

```
  \ifM@adjust@font\else
    \M@sym@{\nabla}{\mathord}{\M@extsymbols@id}{"2207}
  \fi}
```

Set arrows.

\M@arrows@set
```
\def\M@arrows@set{
  \edef\M@arrows@id{M\@tempc-\M@arrowsshape}
  \let\uparrow\@undefined
  \let\Uparrow\@undefined
  \let\downarrow\@undefined
  \let\Downarrow\@undefined
  \let\updownarrow\@undefined
  \let\Updownarrow\@undefined
  \let\longrightarrow\@undefined
  \let\longleftarrow\@undefined
  \let\longleftrightarrow\@undefined
  \let\hookrightarrow\@undefined
  \let\hookleftarrow\@undefined
  \let\Longrightarrow\@undefined
  \let\Longleftarrow\@undefined
  \let\Longleftrightarrow\@undefined
  \let\rightleftharpoons\@undefined
  \M@sym@{\rightarrow}           {\mathrel}{\M@arrows@id}{"2192}
    \let\to\rightarrow
  \M@sym@{\nrightarrow}          {\mathrel}{\M@arrows@id}{"219B}
  \M@sym@{\Rightarrow}           {\mathrel}{\M@arrows@id}{"21D2}
  \M@sym@{\nRightarrow}          {\mathrel}{\M@arrows@id}{"21CF}
  \M@sym@{\Rrightarrow}          {\mathrel}{\M@arrows@id}{"21DB}
  \M@sym@{\longrightarrow}       {\mathrel}{\M@arrows@id}{"27F6}
  \M@sym@{\Longrightarrow}       {\mathrel}{\M@arrows@id}{"27F9}
  \M@sym@{\rightbararrow}        {\mathrel}{\M@arrows@id}{"21A6}
    \let\mapsto\rightbararrow
  \M@sym@{\Rightbararrow}        {\mathrel}{\M@arrows@id}{"2907}
```

```
\M@sym@{\longrightbararrow}    {\mathrel}{\M@arrows@id}{"27FC}
  \let\longmapsto\longrightbararrow
\M@sym@{\Longrightbararrow}    {\mathrel}{\M@arrows@id}{"27FE}
\M@sym@{\hookrightarrow}       {\mathrel}{\M@arrows@id}{"21AA}
\M@sym@{\rightdasharrow}       {\mathrel}{\M@arrows@id}{"21E2}
\M@sym@{\rightharpoonup}       {\mathrel}{\M@arrows@id}{"21C0}
\M@sym@{\rightharpoondown}     {\mathrel}{\M@arrows@id}{"21C1}
\M@sym@{\rightarrowtail}       {\mathrel}{\M@arrows@id}{"21A3}
\M@sym@{\rightoplusarrow}      {\mathrel}{\M@arrows@id}{"27F4}
\M@sym@{\rightwavearrow}       {\mathrel}{\M@arrows@id}{"219D}
\M@sym@{\rightsquigarrow}      {\mathrel}{\M@arrows@id}{"21DD}
\M@sym@{\longrightsquigarrow}  {\mathrel}{\M@arrows@id}{"27FF}
\M@sym@{\looparrowright}       {\mathrel}{\M@arrows@id}{"21AC}
\M@sym@{\curvearrowright}      {\mathrel}{\M@arrows@id}{"293B}
\M@sym@{\circlearrowright}     {\mathrel}{\M@arrows@id}{"21BB}
\M@sym@{\twoheadrightarrow}    {\mathrel}{\M@arrows@id}{"21A0}
\M@sym@{\rightarrowtobar}      {\mathrel}{\M@arrows@id}{"21E5}
\M@sym@{\rightwhitearrow}      {\mathrel}{\M@arrows@id}{"21E8}
\M@sym@{\rightrightarrows}     {\mathrel}{\M@arrows@id}{"21C9}
\M@sym@{\rightrightrightarrows}{\mathrel}{\M@arrows@id}{"21F6}
\M@sym@{\leftarrow}            {\mathrel}{\M@arrows@id}{"2190}
  \let\from\leftarrow
\M@sym@{\nleftarrow}           {\mathrel}{\M@arrows@id}{"219A}
\M@sym@{\Leftarrow}            {\mathrel}{\M@arrows@id}{"21D0}
\M@sym@{\nLeftarrow}           {\mathrel}{\M@arrows@id}{"21CD}
\M@sym@{\Lleftarrow}           {\mathrel}{\M@arrows@id}{"21DA}
\M@sym@{\longleftarrow}        {\mathrel}{\M@arrows@id}{"27F5}
\M@sym@{\Longleftarrow}        {\mathrel}{\M@arrows@id}{"27F8}
\M@sym@{\leftbararrow}         {\mathrel}{\M@arrows@id}{"21A4}
  \let\mapsfrom\leftbararrow
\M@sym@{\Leftbararrow}         {\mathrel}{\M@arrows@id}{"2906}
\M@sym@{\longleftbararrow}     {\mathrel}{\M@arrows@id}{"27FB}
  \let\longmapsfrom\longleftbararrow
\M@sym@{\Longleftbararrow}     {\mathrel}{\M@arrows@id}{"27FD}
\M@sym@{\hookleftarrow}        {\mathrel}{\M@arrows@id}{"21A9}
\M@sym@{\leftdasharrow}        {\mathrel}{\M@arrows@id}{"21E0}
\M@sym@{\leftharpoonup}        {\mathrel}{\M@arrows@id}{"21BC}
\M@sym@{\leftharpoondown}      {\mathrel}{\M@arrows@id}{"21BD}
\M@sym@{\leftarrowtail}        {\mathrel}{\M@arrows@id}{"21A2}
\M@sym@{\leftoplusarrow}       {\mathrel}{\M@arrows@id}{"2B32}
\M@sym@{\leftwavearrow}        {\mathrel}{\M@arrows@id}{"219C}
\M@sym@{\leftsquigarrow}       {\mathrel}{\M@arrows@id}{"21DC}
```

```
\M@sym@{\longleftsquigarrow}   {\mathrel}{\M@arrows@id}{"2B33}
\M@sym@{\looparrowleft}        {\mathrel}{\M@arrows@id}{"21AB}
\M@sym@{\curvearrowleft}       {\mathrel}{\M@arrows@id}{"293A}
\M@sym@{\circlearrowleft}      {\mathrel}{\M@arrows@id}{"21BA}
\M@sym@{\twoheadleftarrow}     {\mathrel}{\M@arrows@id}{"219E}
\M@sym@{\leftarrowtobar}       {\mathrel}{\M@arrows@id}{"21E4}
\M@sym@{\leftwhitearrow}       {\mathrel}{\M@arrows@id}{"21E6}
\M@sym@{\leftleftarrows}       {\mathrel}{\M@arrows@id}{"21C7}
\M@sym@{\leftleftleftarrows}   {\mathrel}{\M@arrows@id}{"2B31}
\M@sym@{\leftrightarrow}       {\mathrel}{\M@arrows@id}{"2194}
\M@sym@{\Leftrightarrow}       {\mathrel}{\M@arrows@id}{"21D4}
\M@sym@{\nLeftrightarrow}      {\mathrel}{\M@arrows@id}{"21CE}
\M@sym@{\longleftrightarrow}   {\mathrel}{\M@arrows@id}{"27F7}
\M@sym@{\Longleftrightarrow}   {\mathrel}{\M@arrows@id}{"27FA}
\M@sym@{\leftrightwavearrow}   {\mathrel}{\M@arrows@id}{"21AD}
\M@sym@{\leftrightarrows}      {\mathrel}{\M@arrows@id}{"21C6}
\M@sym@{\leftrightharpoons}    {\mathrel}{\M@arrows@id}{"21CB}
\M@sym@{\leftrightarrowstobar} {\mathrel}{\M@arrows@id}{"21B9}
\M@sym@{\rightleftarrows}      {\mathrel}{\M@arrows@id}{"21C4}
\M@sym@{\rightleftharpoons}    {\mathrel}{\M@arrows@id}{"21CC}
\M@sym@{\uparrow}              {\mathrel}{\M@arrows@id}{"2191}
\M@sym@{\Uparrow}              {\mathrel}{\M@arrows@id}{"21D1}
\M@sym@{\Uuparrow}             {\mathrel}{\M@arrows@id}{"290A}
\M@sym@{\upbararrow}           {\mathrel}{\M@arrows@id}{"21A5}
\M@sym@{\updasharrow}          {\mathrel}{\M@arrows@id}{"21E1}
\M@sym@{\upharpoonleft}        {\mathrel}{\M@arrows@id}{"21BF}
\M@sym@{\upharpoonright}       {\mathrel}{\M@arrows@id}{"21BE}
\M@sym@{\twoheaduparrow}       {\mathrel}{\M@arrows@id}{"219F}
\M@sym@{\uparrowtobar}         {\mathrel}{\M@arrows@id}{"2912}
\M@sym@{\upwhitearrow}         {\mathrel}{\M@arrows@id}{"21E7}
\M@sym@{\upwhitebararrow}      {\mathrel}{\M@arrows@id}{"21EA}
\M@sym@{\upuparrows}           {\mathrel}{\M@arrows@id}{"21C8}
\M@sym@{\downarrow}            {\mathrel}{\M@arrows@id}{"2193}
\M@sym@{\Downarrow}            {\mathrel}{\M@arrows@id}{"21D3}
\M@sym@{\Ddownarrow}           {\mathrel}{\M@arrows@id}{"290B}
\M@sym@{\downbararrow}         {\mathrel}{\M@arrows@id}{"21A7}
\M@sym@{\downdasharrow}        {\mathrel}{\M@arrows@id}{"21E3}
\M@sym@{\zigzagarrow}          {\mathrel}{\M@arrows@id}{"21AF}
  \let\lightningboltarrow\zigzagarrow
\M@sym@{\downharpoonleft}      {\mathrel}{\M@arrows@id}{"21C3}
\M@sym@{\downharpoonright}     {\mathrel}{\M@arrows@id}{"21C2}
\M@sym@{\twoheaddownarrow}     {\mathrel}{\M@arrows@id}{"21A1}
```

```
\M@sym@{\downarrowtobar}       {\mathrel}{\M@arrows@id}{"2913}
\M@sym@{\downwhitearrow}       {\mathrel}{\M@arrows@id}{"21E9}
\M@sym@{\downdownarrows}       {\mathrel}{\M@arrows@id}{"21CA}
\M@sym@{\updownarrow}          {\mathrel}{\M@arrows@id}{"2195}
\M@sym@{\Updownarrow}          {\mathrel}{\M@arrows@id}{"21D5}
\M@sym@{\updownarrows}         {\mathrel}{\M@arrows@id}{"21C5}
\M@sym@{\downuparrows}         {\mathrel}{\M@arrows@id}{"21F5}
\M@sym@{\updownharpoons}       {\mathrel}{\M@arrows@id}{"296E}
\M@sym@{\downupharpoons}       {\mathrel}{\M@arrows@id}{"296F}
\M@sym@{\nearrow}              {\mathrel}{\M@arrows@id}{"2197}
\M@sym@{\Nearrow}              {\mathrel}{\M@arrows@id}{"21D7}
\M@sym@{\nwarrow}              {\mathrel}{\M@arrows@id}{"2196}
\M@sym@{\Nwarrow}              {\mathrel}{\M@arrows@id}{"21D6}
\M@sym@{\searrow}              {\mathrel}{\M@arrows@id}{"2198}
\M@sym@{\Searrow}              {\mathrel}{\M@arrows@id}{"21D8}
\M@sym@{\swarrow}              {\mathrel}{\M@arrows@id}{"2199}
\M@sym@{\Swarrow}              {\mathrel}{\M@arrows@id}{"21D9}
\M@sym@{\nwsearrow}            {\mathrel}{\M@arrows@id}{"2921}
\M@sym@{\neswarrow}            {\mathrel}{\M@arrows@id}{"2922}
\M@sym@{\lcirclearrow}         {\mathrel}{\M@arrows@id}{"27F2}
\M@sym@{\rcirclearrow}         {\mathrel}{\M@arrows@id}{"27F3}
```

The commands `\relbar` and `\Relbar` produce a smashed minus and an equals sign respectively. They are helper control sequences that LaTeX uses to create other arrows. We have a small issue with amsmath because in XeTeX and LuaTeX, amsmath defines `\relbar` and `\Relbar` in terms of the `\Umathcodes` of the minus and equals signs respectively. That is a good approach in general, but it doesn't work when a package like mathfont allows users to pick different fonts for symbols and arrows. We really want `\relbar` and `\Relbar` to come from the `arrows` font, so our approach is to define the control sequences now and then redefine `\AtBeginDocument` if needed.

```
\let\@relbar\@undefined
\let\@Relbar\@undefined
\M@sym@{\@relbar}{\mathbin}{\M@arrows@id}{"2212}
\M@sym@{\@Relbar}{\mathrel}{\M@arrows@id}{"3D}
\protected\def\relbar{\mathrel{\mathpalette\mathsm@sh\@relbar}}
\protected\def\Relbar{\@Relbar}
```

We redefine stuff if amsmath gets loaded after mathfont.

```
\@ifpackageloaded{amsmath}
  {\relax}{
    \let\@@relbar\relbar
    \let\@@Relbar\Relbar
```

```
      \AtBeginDocument{\@ifpackageloaded{amsmath}{
          \let\relbar\@@relbar
          \let\Relbar\@@Relbar}
        {\relax}}}}
```

Set blackboard bold letters and numbers. The alphanumeric keywords work a bit differently from the other font-setting commands. We define `\mathbb` here, which takes a single argument and is essentially a wrapper around `\M@bb@mathcodes`. That command changes the `\Umathcode`s of letters to the Unicode hex values of corresponding blackboard-bold characters, and throughout, `\M@bb@num` stores the family number of the sumbol font for the `bb` character class. In the definition of `\mathbb`, we use `\begingroup` and `\endgroup` to avoid creating unexpected atoms. The other alphanumeric keywords work similarly.

```
\M@bb@set \def\M@bb@set{
  \mathbb   \protected\def\mathbb##1{\relax
              \ifmmode\else
                \M@HModeError\mathbb
                $
              \fi
              \begingroup
                \M@bb@mathcodes
                ##1
              \endgroup}
\M@bb@num   \edef\M@bb@num{\number
              \csname symM\@tempc-\M@bbshape\endcsname}
\M@bb@mathcodes \protected\edef\M@bb@mathcodes{
              \Umathcode`A=0+\M@bb@num"1D538\relax
              \Umathcode`B=0+\M@bb@num"1D539\relax
              \Umathcode`C=0+\M@bb@num"2102\relax
              \Umathcode`D=0+\M@bb@num"1D53B\relax
              \Umathcode`E=0+\M@bb@num"1D53C\relax
              \Umathcode`F=0+\M@bb@num"1D53D\relax
              \Umathcode`G=0+\M@bb@num"1D53E\relax
              \Umathcode`H=0+\M@bb@num"210D\relax
              \Umathcode`I=0+\M@bb@num"1D540\relax
              \Umathcode`J=0+\M@bb@num"1D541\relax
              \Umathcode`K=0+\M@bb@num"1D542\relax
              \Umathcode`L=0+\M@bb@num"1D543\relax
              \Umathcode`M=0+\M@bb@num"1D544\relax
              \Umathcode`N=0+\M@bb@num"2115\relax
              \Umathcode`O=0+\M@bb@num"1D546\relax
```

```
\Umathcode`P=0+\M@bb@num"2119\relax
\Umathcode`Q=0+\M@bb@num"211A\relax
\Umathcode`R=0+\M@bb@num"211D\relax
\Umathcode`S=0+\M@bb@num"1D54A\relax
\Umathcode`T=0+\M@bb@num"1D54B\relax
\Umathcode`U=0+\M@bb@num"1D54C\relax
\Umathcode`V=0+\M@bb@num"1D54D\relax
\Umathcode`W=0+\M@bb@num"1D54E\relax
\Umathcode`X=0+\M@bb@num"1D54F\relax
\Umathcode`Y=0+\M@bb@num"1D550\relax
\Umathcode`Z=0+\M@bb@num"2124\relax
\Umathcode`a=0+\M@bb@num"1D552\relax
\Umathcode`b=0+\M@bb@num"1D553\relax
\Umathcode`c=0+\M@bb@num"1D554\relax
\Umathcode`d=0+\M@bb@num"1D555\relax
\Umathcode`e=0+\M@bb@num"1D556\relax
\Umathcode`f=0+\M@bb@num"1D557\relax
\Umathcode`g=0+\M@bb@num"1D558\relax
\Umathcode`h=0+\M@bb@num"1D559\relax
\Umathcode`i=0+\M@bb@num"1D55A\relax
\Umathcode`j=0+\M@bb@num"1D55B\relax
\Umathcode`k=0+\M@bb@num"1D55C\relax
\Umathcode`l=0+\M@bb@num"1D55D\relax
\Umathcode`m=0+\M@bb@num"1D55E\relax
\Umathcode`n=0+\M@bb@num"1D55F\relax
\Umathcode`o=0+\M@bb@num"1D560\relax
\Umathcode`p=0+\M@bb@num"1D561\relax
\Umathcode`q=0+\M@bb@num"1D562\relax
\Umathcode`r=0+\M@bb@num"1D563\relax
\Umathcode`s=0+\M@bb@num"1D564\relax
\Umathcode`t=0+\M@bb@num"1D565\relax
\Umathcode`u=0+\M@bb@num"1D566\relax
\Umathcode`v=0+\M@bb@num"1D567\relax
\Umathcode`w=0+\M@bb@num"1D568\relax
\Umathcode`x=0+\M@bb@num"1D569\relax
\Umathcode`y=0+\M@bb@num"1D56A\relax
\Umathcode`z=0+\M@bb@num"1D56B\relax
\Umathcode`0=0+\M@bb@num"1D7D8\relax
\Umathcode`1=0+\M@bb@num"1D7D9\relax
\Umathcode`2=0+\M@bb@num"1D7DA\relax
\Umathcode`3=0+\M@bb@num"1D7DB\relax
\Umathcode`4=0+\M@bb@num"1D7DC\relax
```

```
    \Umathcode`5=0+\M@bb@num"1D7DD\relax
    \Umathcode`6=0+\M@bb@num"1D7DE\relax
    \Umathcode`7=0+\M@bb@num"1D7DF\relax
    \Umathcode`8=0+\M@bb@num"1D7E0\relax
    \Umathcode`9=0+\M@bb@num"1D7E1\relax}}
```

Set caligraphic letters.

```
\M@cal@set \def\M@cal@set{
  \mathcal   \protected\def\mathcal##1{\relax
               \ifmmode\else
                 \M@HModeError\mathcal
                 $
               \fi
               \begingroup
                 \M@cal@mathcodes
                 ##1
               \endgroup}
\M@cal@num   \edef\M@cal@num{\number
               \csname symM\@tempc-\M@calshape\endcsname}
\M@cal@mathcodes \protected\edef\M@cal@mathcodes{
               \Umathcode`A=0+\M@cal@num"1D49C\relax
               \Umathcode`B=0+\M@cal@num"212C\relax
               \Umathcode`C=0+\M@cal@num"1D49E\relax
               \Umathcode`D=0+\M@cal@num"1D49F\relax
               \Umathcode`E=0+\M@cal@num"2130\relax
               \Umathcode`F=0+\M@cal@num"2131\relax
               \Umathcode`G=0+\M@cal@num"1D4A2\relax
               \Umathcode`H=0+\M@cal@num"210B\relax
               \Umathcode`I=0+\M@cal@num"2110\relax
               \Umathcode`J=0+\M@cal@num"1D4A5\relax
               \Umathcode`K=0+\M@cal@num"1D4A6\relax
               \Umathcode`L=0+\M@cal@num"2112\relax
               \Umathcode`M=0+\M@cal@num"2133\relax
               \Umathcode`N=0+\M@cal@num"1D4A9\relax
               \Umathcode`O=0+\M@cal@num"1D4AA\relax
               \Umathcode`P=0+\M@cal@num"1D4AB\relax
               \Umathcode`Q=0+\M@cal@num"1D4AC\relax
               \Umathcode`R=0+\M@cal@num"211B\relax
               \Umathcode`S=0+\M@cal@num"1D4AE\relax
               \Umathcode`T=0+\M@cal@num"1D4AF\relax
               \Umathcode`U=0+\M@cal@num"1D4B0\relax
               \Umathcode`V=0+\M@cal@num"1D4B1\relax
               \Umathcode`W=0+\M@cal@num"1D4B2\relax
```

```
    \Umathcode`X=0+\M@cal@num"1D4B3\relax
    \Umathcode`Y=0+\M@cal@num"1D4B4\relax
    \Umathcode`Z=0+\M@cal@num"1D4B5\relax
    \Umathcode`a=0+\M@cal@num"1D4B6\relax
    \Umathcode`b=0+\M@cal@num"1D4B7\relax
    \Umathcode`c=0+\M@cal@num"1D4B8\relax
    \Umathcode`d=0+\M@cal@num"1D4B9\relax
    \Umathcode`e=0+\M@cal@num"212F\relax
    \Umathcode`f=0+\M@cal@num"1D4BB\relax
    \Umathcode`g=0+\M@cal@num"210A\relax
    \Umathcode`h=0+\M@cal@num"1D4BD\relax
    \Umathcode`i=0+\M@cal@num"1D4BE\relax
    \Umathcode`j=0+\M@cal@num"1D4BF\relax
    \Umathcode`k=0+\M@cal@num"1D4C0\relax
    \Umathcode`l=0+\M@cal@num"1D4C1\relax
    \Umathcode`m=0+\M@cal@num"1D4C2\relax
    \Umathcode`n=0+\M@cal@num"1D4C3\relax
    \Umathcode`o=0+\M@cal@num"2134\relax
    \Umathcode`p=0+\M@cal@num"1D4C5\relax
    \Umathcode`q=0+\M@cal@num"1D4C6\relax
    \Umathcode`r=0+\M@cal@num"1D4C7\relax
    \Umathcode`s=0+\M@cal@num"1D4C8\relax
    \Umathcode`t=0+\M@cal@num"1D4C9\relax
    \Umathcode`u=0+\M@cal@num"1D4CA\relax
    \Umathcode`v=0+\M@cal@num"1D4CB\relax
    \Umathcode`w=0+\M@cal@num"1D4CC\relax
    \Umathcode`x=0+\M@cal@num"1D4CD\relax
    \Umathcode`y=0+\M@cal@num"1D4CE\relax
    \Umathcode`z=0+\M@cal@num"1D4CF\relax}}
```

Set fraktur letters.

```
\M@frak@set \def\M@frak@set{
  \mathfrak   \protected\def\mathfrak##1{\relax
                \ifmmode\else
                  \M@HModeError\mathfrak
                  $
                \fi
                \begingroup
                  \M@frak@mathcodes
                  ##1
                \endgroup}
\M@frak@num   \edef\M@frak@num{\number
                \csname symM\@tempc-\M@frakshape\endcsname}
```

```
\M@frak@mathcodes 3787  \protected\edef\M@frak@mathcodes{
                  3788    \Umathcode`A=0+\M@frak@num"1D504\relax
                  3789    \Umathcode`B=0+\M@frak@num"1D505\relax
                  3790    \Umathcode`C=0+\M@frak@num"212D\relax
                  3791    \Umathcode`D=0+\M@frak@num"1D507\relax
                  3792    \Umathcode`E=0+\M@frak@num"1D508\relax
                  3793    \Umathcode`F=0+\M@frak@num"1D509\relax
                  3794    \Umathcode`G=0+\M@frak@num"1D50A\relax
                  3795    \Umathcode`H=0+\M@frak@num"210C\relax
                  3796    \Umathcode`I=0+\M@frak@num"2111\relax
                  3797    \Umathcode`J=0+\M@frak@num"1D50D\relax
                  3798    \Umathcode`K=0+\M@frak@num"1D50E\relax
                  3799    \Umathcode`L=0+\M@frak@num"1D50F\relax
                  3800    \Umathcode`M=0+\M@frak@num"1D510\relax
                  3801    \Umathcode`N=0+\M@frak@num"1D511\relax
                  3802    \Umathcode`O=0+\M@frak@num"1D512\relax
                  3803    \Umathcode`P=0+\M@frak@num"1D513\relax
                  3804    \Umathcode`Q=0+\M@frak@num"1D514\relax
                  3805    \Umathcode`R=0+\M@frak@num"211C\relax
                  3806    \Umathcode`S=0+\M@frak@num"1D516\relax
                  3807    \Umathcode`T=0+\M@frak@num"1D517\relax
                  3808    \Umathcode`U=0+\M@frak@num"1D518\relax
                  3809    \Umathcode`V=0+\M@frak@num"1D519\relax
                  3810    \Umathcode`W=0+\M@frak@num"1D51A\relax
                  3811    \Umathcode`X=0+\M@frak@num"1D51B\relax
                  3812    \Umathcode`Y=0+\M@frak@num"1D51C\relax
                  3813    \Umathcode`Z=0+\M@frak@num"2128\relax
                  3814    \Umathcode`a=0+\M@frak@num"1D51E\relax
                  3815    \Umathcode`b=0+\M@frak@num"1D51F\relax
                  3816    \Umathcode`c=0+\M@frak@num"1D520\relax
                  3817    \Umathcode`d=0+\M@frak@num"1D521\relax
                  3818    \Umathcode`e=0+\M@frak@num"1D522\relax
                  3819    \Umathcode`f=0+\M@frak@num"1D523\relax
                  3820    \Umathcode`g=0+\M@frak@num"1D524\relax
                  3821    \Umathcode`h=0+\M@frak@num"1D525\relax
                  3822    \Umathcode`i=0+\M@frak@num"1D526\relax
                  3823    \Umathcode`j=0+\M@frak@num"1D527\relax
                  3824    \Umathcode`k=0+\M@frak@num"1D528\relax
                  3825    \Umathcode`l=0+\M@frak@num"1D529\relax
                  3826    \Umathcode`m=0+\M@frak@num"1D52A\relax
                  3827    \Umathcode`n=0+\M@frak@num"1D52B\relax
                  3828    \Umathcode`o=0+\M@frak@num"1D52C\relax
```

```
    \Umathcode`p=0+\M@frak@num"1D52D\relax
    \Umathcode`q=0+\M@frak@num"1D52E\relax
    \Umathcode`r=0+\M@frak@num"1D52F\relax
    \Umathcode`s=0+\M@frak@num"1D530\relax
    \Umathcode`t=0+\M@frak@num"1D531\relax
    \Umathcode`u=0+\M@frak@num"1D532\relax
    \Umathcode`v=0+\M@frak@num"1D533\relax
    \Umathcode`w=0+\M@frak@num"1D534\relax
    \Umathcode`x=0+\M@frak@num"1D535\relax
    \Umathcode`y=0+\M@frak@num"1D536\relax
    \Umathcode`z=0+\M@frak@num"1D537\relax}}
```

Set bold caligraphic letters.

```
\M@bcal@set \def\M@bcal@set{
  \mathbcal   \protected\def\mathbcal##1{\relax
                \ifmmode\else
                  \M@HModeError\mathbcal
                  $
                \fi
                \begingroup
                  \M@bcal@mathcodes
                  ##1
                \endgroup}
\M@bcal@num     \edef\M@bcal@num{\number
                  \csname symM\@tempc-\M@bcalshape\endcsname}
\M@bcal@mathcodes \protected\edef\M@bcal@mathcodes{
                  \Umathcode`A=0+\M@bcal@num"1D4D0\relax
                  \Umathcode`B=0+\M@bcal@num"1D4D1\relax
                  \Umathcode`C=0+\M@bcal@num"1D4D2\relax
                  \Umathcode`D=0+\M@bcal@num"1D4D3\relax
                  \Umathcode`E=0+\M@bcal@num"1D4D4\relax
                  \Umathcode`F=0+\M@bcal@num"1D4D5\relax
                  \Umathcode`G=0+\M@bcal@num"1D4D6\relax
                  \Umathcode`H=0+\M@bcal@num"1D4D7\relax
                  \Umathcode`I=0+\M@bcal@num"1D4D8\relax
                  \Umathcode`J=0+\M@bcal@num"1D4D9\relax
                  \Umathcode`K=0+\M@bcal@num"1D4DA\relax
                  \Umathcode`L=0+\M@bcal@num"1D4DB\relax
                  \Umathcode`M=0+\M@bcal@num"1D4DC\relax
                  \Umathcode`N=0+\M@bcal@num"1D4DD\relax
                  \Umathcode`O=0+\M@bcal@num"1D4DE\relax
                  \Umathcode`P=0+\M@bcal@num"1D4DF\relax
                  \Umathcode`Q=0+\M@bcal@num"1D4E0\relax
```

```
3870    \Umathcode`R=0+\M@bcal@num"1D4E1\relax
3871    \Umathcode`S=0+\M@bcal@num"1D4E2\relax
3872    \Umathcode`T=0+\M@bcal@num"1D4E3\relax
3873    \Umathcode`U=0+\M@bcal@num"1D4E4\relax
3874    \Umathcode`V=0+\M@bcal@num"1D4E5\relax
3875    \Umathcode`W=0+\M@bcal@num"1D4E6\relax
3876    \Umathcode`X=0+\M@bcal@num"1D4E7\relax
3877    \Umathcode`Y=0+\M@bcal@num"1D4E8\relax
3878    \Umathcode`Z=0+\M@bcal@num"1D4E9\relax
3879    \Umathcode`a=0+\M@bcal@num"1D4EA\relax
3880    \Umathcode`b=0+\M@bcal@num"1D4EB\relax
3881    \Umathcode`c=0+\M@bcal@num"1D4EC\relax
3882    \Umathcode`d=0+\M@bcal@num"1D4ED\relax
3883    \Umathcode`e=0+\M@bcal@num"1D4EE\relax
3884    \Umathcode`f=0+\M@bcal@num"1D4EF\relax
3885    \Umathcode`g=0+\M@bcal@num"1D4F0\relax
3886    \Umathcode`h=0+\M@bcal@num"1D4F1\relax
3887    \Umathcode`i=0+\M@bcal@num"1D4F2\relax
3888    \Umathcode`j=0+\M@bcal@num"1D4F3\relax
3889    \Umathcode`k=0+\M@bcal@num"1D4F4\relax
3890    \Umathcode`l=0+\M@bcal@num"1D4F5\relax
3891    \Umathcode`m=0+\M@bcal@num"1D4F6\relax
3892    \Umathcode`n=0+\M@bcal@num"1D4F7\relax
3893    \Umathcode`o=0+\M@bcal@num"1D4F8\relax
3894    \Umathcode`p=0+\M@bcal@num"1D4F9\relax
3895    \Umathcode`q=0+\M@bcal@num"1D4FA\relax
3896    \Umathcode`r=0+\M@bcal@num"1D4FB\relax
3897    \Umathcode`s=0+\M@bcal@num"1D4FC\relax
3898    \Umathcode`t=0+\M@bcal@num"1D4FD\relax
3899    \Umathcode`u=0+\M@bcal@num"1D4FE\relax
3900    \Umathcode`v=0+\M@bcal@num"1D4FF\relax
3901    \Umathcode`w=0+\M@bcal@num"1D500\relax
3902    \Umathcode`x=0+\M@bcal@num"1D501\relax
3903    \Umathcode`y=0+\M@bcal@num"1D502\relax
3904    \Umathcode`z=0+\M@bcal@num"1D503\relax}}
```

Set bold fraktur letters.

```
\M@bfrak@set 3905 \def\M@bfrak@set{
  \mathbfrak 3906   \protected\def\mathbfrak##1{\relax
             3907     \ifmmode\else
             3908       \M@HModeError\mathbfrak
             3909       $
             3910     \fi
```

```
    \begingroup
      \M@bfrak@mathcodes
      ##1
    \endgroup}
\M@bfrak@num
  \edef\M@bfrak@num{\number
    \csname symM\@tempc-\M@bfrakshape\endcsname}
\M@bfrak@mathcodes
  \protected\edef\M@bfrak@mathcodes{
    \Umathcode`A=0+\M@bfrak@num"1D56C\relax
    \Umathcode`B=0+\M@bfrak@num"1D56D\relax
    \Umathcode`C=0+\M@bfrak@num"1D56E\relax
    \Umathcode`D=0+\M@bfrak@num"1D56F\relax
    \Umathcode`E=0+\M@bfrak@num"1D570\relax
    \Umathcode`F=0+\M@bfrak@num"1D571\relax
    \Umathcode`G=0+\M@bfrak@num"1D572\relax
    \Umathcode`H=0+\M@bfrak@num"1D573\relax
    \Umathcode`I=0+\M@bfrak@num"1D574\relax
    \Umathcode`J=0+\M@bfrak@num"1D575\relax
    \Umathcode`K=0+\M@bfrak@num"1D576\relax
    \Umathcode`L=0+\M@bfrak@num"1D577\relax
    \Umathcode`M=0+\M@bfrak@num"1D578\relax
    \Umathcode`N=0+\M@bfrak@num"1D579\relax
    \Umathcode`O=0+\M@bfrak@num"1D57A\relax
    \Umathcode`P=0+\M@bfrak@num"1D57B\relax
    \Umathcode`Q=0+\M@bfrak@num"1D57C\relax
    \Umathcode`R=0+\M@bfrak@num"1D57D\relax
    \Umathcode`S=0+\M@bfrak@num"1D57E\relax
    \Umathcode`T=0+\M@bfrak@num"1D57F\relax
    \Umathcode`U=0+\M@bfrak@num"1D580\relax
    \Umathcode`V=0+\M@bfrak@num"1D581\relax
    \Umathcode`W=0+\M@bfrak@num"1D582\relax
    \Umathcode`X=0+\M@bfrak@num"1D583\relax
    \Umathcode`Y=0+\M@bfrak@num"1D584\relax
    \Umathcode`Z=0+\M@bfrak@num"1D585\relax
    \Umathcode`a=0+\M@bfrak@num"1D586\relax
    \Umathcode`b=0+\M@bfrak@num"1D587\relax
    \Umathcode`c=0+\M@bfrak@num"1D588\relax
    \Umathcode`d=0+\M@bfrak@num"1D589\relax
    \Umathcode`e=0+\M@bfrak@num"1D58A\relax
    \Umathcode`f=0+\M@bfrak@num"1D58B\relax
    \Umathcode`g=0+\M@bfrak@num"1D58C\relax
    \Umathcode`h=0+\M@bfrak@num"1D58D\relax
    \Umathcode`i=0+\M@bfrak@num"1D58E\relax
```

```
3953     \Umathcode`j=0+\M@bfrak@num"1D58F\relax
3954     \Umathcode`k=0+\M@bfrak@num"1D590\relax
3955     \Umathcode`l=0+\M@bfrak@num"1D591\relax
3956     \Umathcode`m=0+\M@bfrak@num"1D592\relax
3957     \Umathcode`n=0+\M@bfrak@num"1D593\relax
3958     \Umathcode`o=0+\M@bfrak@num"1D594\relax
3959     \Umathcode`p=0+\M@bfrak@num"1D595\relax
3960     \Umathcode`q=0+\M@bfrak@num"1D596\relax
3961     \Umathcode`r=0+\M@bfrak@num"1D597\relax
3962     \Umathcode`s=0+\M@bfrak@num"1D598\relax
3963     \Umathcode`t=0+\M@bfrak@num"1D599\relax
3964     \Umathcode`u=0+\M@bfrak@num"1D59A\relax
3965     \Umathcode`v=0+\M@bfrak@num"1D59B\relax
3966     \Umathcode`w=0+\M@bfrak@num"1D59C\relax
3967     \Umathcode`x=0+\M@bfrak@num"1D59D\relax
3968     \Umathcode`y=0+\M@bfrak@num"1D59E\relax
3969     \Umathcode`z=0+\M@bfrak@num"1D59F\relax}}
```

Reset `\endlinechar`.

```
3970 \endlinechar\count@
```

And that's everything!

# Version History

New features and updates with each version. Listed in no particular order.

**1.1b** ..... July 2018
—initial release

**1.2** ..... August 2018
—minor bug fix for `\mathfrak`
—eliminated redundant batchfile

**1.3** ..... January 2019
—added `symbols` keyword
—created `mathfont_example.pdf`
—corrected the description of the mathastext package
—font-change `\message` added to `\mathfont`

**1.4** ..... April 2019
—`\setfont` command added
—`\mathfont` optional argument can parse spaces
—`no-operators` now default package optional argument
—added `\comma` command
—new fancy fatal error message
—improved messaging for `\mathfont`
—internal command `\mathpound` changed to `\mathhash`
—added a missing `#1` after `\char`\"` in the example code redefining `"` in the user guide

**1.5** ..... April 2019
—separated `\increment` and `\Delta`
—version history added
—initial off-the-shelf use insert added

**1.6** ..... December 2019
—separated implementation and user documentation
—created `mathfont_heading.tex`
—created `mathfont_doc_patch.tex` for use with the index
—changed `mathfont_greek.pdf` to `mathfont_symbol_list.pdf`
—eliminated `mathfont_example.pdf`
—eliminated `operators` package option
—eliminated `packages` package option
—font name can be package option
—added Hebrew and Cyrillic characters
—separated ancient Greek from modern Greek characters
—created new keywords: `extsymbols`, `delimiters`, `arrows`, `diacritics`, `bigops`, `extbigops`
—improved messaging
—improved internal code for local font-change commands
—improved space parsing for the optional argument of `\mathfont`
—bug fix for `\#`, etc. commands
—bad input for `\mathbb`, etc. now gives a warning
—improved error checking for `\newmathrm`, etc. commands
—`\mathfont` now ignores bad options (on top of issuing an error)
—iternal commands now begin with `\M@…`

—added Easter Egg!
—improved indexing
—`mathfont.dtx` renamed as `mathfont_code.dtx`
—`\newmathbold` renamed as `\newmathbf`
—default local font changes now use `\updefault`, etc.
—added fatal error for missing fontspec
—fatal errors result in `\endinput` rather than `\@@end`

**2.0** . . . . . December 2021

**Big Change:** Font adjustments for LuaTEX: new glyph boundaries for Latin letters in math mode, resizable delimiters, big operators, MathConstants table based on font metrics.

—added `\charmline` and `\charmfile`
—added `\mathconstantsfont`
—certain dimensions in equations are now adjustable when typesetting with LuaTEX
—added `adjust` and `no-adjust` package options
—automatic generation of `ind` file
—fixed symbols for `\leftharpoonup`, `\leftharpoondown`, and fraktur R
—cleaned up internal code and documentation
—font names for `\mathfont` stored to avoid multiple symbol font declarations with the same font
—more information about nfss family names stored and provided
—added option `empty`
—raised upper bound on `\DeclareSymbolFont` to 256
—reintroduced `mathfont_example.tex` with different contents
—changed several symbol-commands to `\protected` rather than robust macros
—many user-level commands are now `\protected`
—`\updefault` changed to `\shapedefault`
—eliminated `\catcode` change for space characters when scanning optional argument of `\mathfont`
—improved messaging for `\mathfont`
—removed dependence on fontspec and added internal font-loader
—switched `\epsilon` and `\varepsilon`
—switched `\phi` and `\varphi`
—changed `/` to produce a solidus in math mode and added `\fractionslash`
—removed `\restoremathinternals` from the user guide
—`\setfont` now sets `\mathrm`, etc.
—added `\newmathsc`, other math alphabet commands for small caps

**2.1** . . . . . November 2022
—`\mathbb`, etc. commands change `\Umathcode`s of letters instead of `\M@`⟨*bb,etc.*⟩`@`⟨*letter*⟩ commands
—removed warnings about non-letter contents of `\mathbb`, etc.

—fonts loaded twice, once in unspecified mode (for text) and once in base mode (for math)
—`mathconstantsfont` accepts "upright" or "italic" as optional argument

**2.2** . . . . . December 2022
—changed the easter egg text
—updated patch for `\DeclareSymbolFont` to work with changes to the kernel (fixed the `\M@p@tch@decl@re` error message)
—calling Plain TEX on `mathfont_code.dtx` produces sty file and no pdf file

**2.2a** . . . . . December 2022
—bug fix for `\mathconstantsfont`
—bug fix for `\M@check@int`
—added `doc2` option to `ltxdoc` in `mathfont_code.dtx`

**2.2b** . . . . . August 2023
—minor changes to code and documentation
—`\ng` now works in math (as not greater than symbol) and text (as pronounciation symbol)

**2.3** . . . . . September 2023
—solidus and `\fractionslash` are `\mathord` instead of `\mathbin`
—removed `\mathfont{fontspec}` functionality
—redesigned font-loader
—added package options `default-loader` and `fontspec-loader`

**2.4** . . . . . April 2025
—`\colon` is `\mathpunct` instead of `\mathord`
—moved `\relbar` and `\Relbar` to `arrows`
—reformatted `mathfont_code.pdf`
—made compatible with `\mathbb`, etc. commands from other packages
—renamed `set_nomath_true` to `set_nomath_false`
—improved messaging `\AtBeginDocument`
—removed deprecated package options, `\newmathbold`, `\restoremathinternals`
—more Easter egg messages

**2.4a** . . . . . June 2025
—bug fix involving nil value and `the_font`
—changed underscores in file names to hyphens

**3.0** . . . . . January 2026
—`\mathfont` and `\setfont` no longer restricted to preamble
—effects of `\mathfont` are local instead of global
—`\mathfont` now overrides previous calls to `\mathfont`
—added `\mathfontshapes`
—added `\mainfont`
—changed `\setfont` to `\documentfont` (but kept old name for backwards compatibility)
—changed `\setmathfontcommands` to `\mathfontcommands` and added documentation in the user guide

—added support for arbitrary NFSS series/shape codes in `\mathfont`
—separated macros for default shapes from macros for shapes used in `\mathfont`
—formalized language of keywords and shape identifiers
—font-loader is better about using argument with spaces removed for checking NFSS font family
—font-loader now uses `\@tempshape` and `\@tempseries` when parsing argument
—moved assignment of `\M@count` values into `\M@newfont`
—changed `\CharmLine` to `\charmline` (but kept old name for backwards compatibility)
—`\charmline` now uses `token.scan_string()` for scanning and expanding its argument
—changed `\CharmFile` to `\charmfile` (but kept old name for backwards compatibility)
—added `\charminfo`
—added `\charmtype`
—renamed counts for `\SurdHorizontalFactor`, etc. to be at user level
—formalized language of user-friendly versions of LuaTEX-only commands in the user guide
—corrected axis height
—font adjustments now happen only if `mode=base`
—no more virtual Latin letters in extra encoding slots (package now modifies Latin letters in their regular encoding slots for base-mode fonts)
—added `bold` and `bolditalic` shape identifiers
—no more type `u` (became type `a`)
—added separate charm information for upright and italic fonts (specified using `/` character)
—reworked default charm values
—possible to force type `a` and type `e` characters using `?` and `!` in `\charmline`
—added extra field in type `e` for italic correction
—clarified role of `bot_accent`, which does nothing, in the user guide
—cleaned up Lua code
—added `finishing_touches` callback
—bug fix involving the handling of italic correction in Lua font adjustments
—bug fix involving characters in operator font not displaying
—corrected UTF-16BE information added to character subtables in the font
—new definition for `\not`
—added `\negslash`
—added `script=math` to OpenType features for fonts if intended for math (so math fonts load with built-in math features)
—added support for Unicode input
—switched `\epsilon` and `\varepsilon` (so `\epsilon` is Unicode epsilon and `\varepsilon` is Unicode lunate epsilon)
—surd is now an active character

in math mode (`\mathcode"8000`)
—cleaned up error messages
—improved messaging
`\AtBeginDocument`
—better error checking in
`\M@check@int`

**3.0a** . . . . . February 2026
—removed spurious space in
`\mainfont`
—for base-mode version of fonts,
replaced `script=math` with
`-nomathparam`
—added `script=latn` to default
font features

# Index

Entries refer to lines in the code. Bold means a definition.

## M

**N**

**O**



**P**

**T**



**U**



**V**