The lualatex-math package*

Philipp Stephani
p.stephani2@gmail.com

2012/08/23

Contents

1 Introduction

LuaTeX brings major improvements to all areas of TeX typesetting and programming. They are made available through new primitives or the embedded Lua interpreter, and combining them with existing LaTeX 2ε packages is not a task the average LaTeX user should have to care about. Therefore a multitude of LaTeX 2ε packages have been written to bridge the gap between documents and the new features. The lualatex-math package focuses on the additional possibilities for mathematical typesetting. The most eminent of the new features is the ability to use Unicode and OpenType fonts, as provided by Will Robertson's unicode-math package. However, there is a smaller group of changes unrelated to Unicode: these are

*This document corresponds to lualatex-math v0.3c, dated 2012/08/23.

to be dealt with in this package. While in principle most TEX documents written for traditional engines should work just fine with LuaTEX, there is a small number of breaking changes that require the attention of package authors. The lualatex-math package tries to fix some of the issues encountered while porting traditional macro packages to LuaLATEX.

The decision to write patches for existing macro packages should not be made lightly: monkey patching done by somebody different from the original package author ties the patching package to the implementation details of the patched functionality and breaks all rules of encapsulation. However, due to the lack of alternatives, it has become an accepted way of providing new functionality in LATEX. To keep the negative impact as small as possible, the lualatex-math package patches only the LATEX 2_ε kernel and a small number of popular packages. In general, this package should be regarded as a temporary kludge that should be removed once the math-related packages are updated to be usable with LuaTEX. By its very nature, the package is likely to cause problems; in such cases, please refer to the issue tracker[1].

2 Interface

The lualatex-math package can be loaded with `\usepackage` or `\RequirePackage`, as usual. It has no options and no public interface; the patching is always done when the package is loaded and cannot be controlled. As a matter of course, the lualatex-math package needs LuaLATEX to function; it will produce error messages and refuse to load under other engines and formats. The package depends on the expl3 bundle, the etoolbox package, the luatexbase bundle and the filehook package. The lualatex-math package is independent of the unicode-math package; the fixes provided here are valid for both Unicode and legacy math typesetting.

Currently patches for the LATEX 2_ε kernel and the amsmath, mathtools and icomma packages are provided. It is not relevant whether you load these packages before or after lualatex-math. They should work as expected (and ideally you shouldn't notice anything), but if you load other packages that by themselves overwrite commands patched by this package, bad things may happen, as it is usual with LATEX.

`\mathstyle`, `\luatexmathstyle`

One user-visible change is that the new `\mathstyle` primitive (usually called `\luatexmathstyle` in LuaLATEX) should work in all cases after the lualatex-math package has been loaded, provided you use the high-level macros `\frac`, `\binom`, and `\genfrac`. The fraction-like TEX primitives like `\over` or `\atopwithdelims` and the plain TEX leftovers like `\brack` or `\choose` cannot be patched, and you shouldn't use them.

`\frac`, `\binom`, `\genfrac`

3 Implementation of the LATEX 2_ε package

3.1 Requirements

```
1 ⟨*package⟩
2 \NeedsTeXFormat{LaTeX2e}[2009/09/24]
3 \RequirePackage{expl3}[2012/08/14]
4 \ProvidesExplPackage{lualatex-math}{2012/08/23}{0.3c}%
5   {Patches for mathematics typesetting with LuaLaTeX}
6 \RequirePackage { etoolbox } [ 2007/10/08 ]
7 \RequirePackage { luatexbase } [ 2010/05/27 ]
8 \RequirePackage { filehook } [ 2011/03/09 ]
```

[1] https://github.com/phst/lualatex-math/issues

```
\RequireLuaModule { lualatex-math } [ 2011/05/05 ]
```

\lltxmath_restore_catcode:N Executing the exhaustive expansion of `\lltxmath_restore_catcode:N`⟨*character token*⟩ restores the category code of the ⟨*character token*⟩ to its current value.

```
\cs_new_nopar:Npn \lltxmath_restore_catcode:N #1 {
  \char_set_catcode:nn { \int_eval:n { `#1 } }
    { \char_value_catcode:n { `#1 } }
}
```

We use the macro defined above to restore the category code of the dollar sign. There are packages that make the dollar sign active; hopefully they get loaded after the packages we are trying to patch.

```
\exp_args:Nx \AtEndOfPackage {
  \lltxmath_restore_catcode:N \$
}
\char_set_catcode_math_toggle:N \$
```

3.2 Messages

luatex-required Issued when not running under LuaTEX.

```
\msg_new:nnn { lualatex-math } { luatex-required } {
  The~ lualatex-math~ package~ requires~ LuaTeX. \\
  I~ will~ stop~ loading~ now.
}
```

different-meanings Issued when two control sequences have different meanings, but should not.

```
\msg_new:nnnn { lualatex-math } { different-meanings } {
  I've~ expected~ the~ control~ sequences \\
  #1~ and~ #3 \\
  to~ have~ the~ same~ meaning,~ but~ their~ meanings~ are~ different.
} {
  The~ meaning~ of~ #1~ is: \\
  #2 \\
  The~ meaning~ of~ #3~ is: \\
  #4
}
```

macro-expected Issued when trying to patch a non-macro. The first argument must be the detokenized macro name.

```
\msg_new:nnn { lualatex-math } { macro-expected } {
  I've~ expected~ that~ #1~ is~ a~ macro,~ but~ it~ isn't.
}
```

wrong-meaning Issued when trying to patch a macro with an unexpected meaning. The first argument must be the detokenized macro name; the second argument must be the actual detokenized meaning; and the thied argument must be the expected detokenized meaning.

```
\msg_new:nnn { lualatex-math } { wrong-meaning } {
  I've~ expected~ #1~ to~ have~ the~ meaning \\
  #3, \\
  but~ it~ has~ the~ meaning \\
  #2.
}
```

patch-macro Issued when a macro is patched. The first argument must be the detokenized macro name.

```
\msg_new:nnn { lualatex-math } { patch-macro } {
  I'm~ going~ to~ patch~ macro~ #1.
}
```

3.3 Initialization

Unless we are running under LuaTeX, we issue an error and quit immediately. Loading the luatexbase module will already have produced an error, but we issue another one for clarity.

```
\luatex_if_engine:F {
  \msg_error:nn { lualatex-math } { luatex-required }
  \endinput
}
```

3.4 Patching

\lltxmath_temp:w A scratch macro.

```
\cs_new_eq:NN \lltxmath_temp:w \prg_do_nothing:
```

\luatexUmathcode \luatexUmathcodenum \luatexUmathchardef We need the extended versions of `\mathcode` and `\mathchardef`. The command `\luatexbase@ensure@primitive{`⟨*name*⟩`}` makes sure that the LuaTeX primitive `\`⟨*name*⟩ is available under the qualified name `\luatex`⟨*name*⟩.

```
\luatexbase@ensure@primitive { Umathcode }
\luatexbase@ensure@primitive { Umathcodenum }
\luatexbase@ensure@primitive { Umathchardef }
```

\lltxmath_assert_eq:NN The macro `\lltxmath_assert_eq:NN`⟨*first command*⟩⟨*second command*⟩ tests whether the control sequences ⟨*first command*⟩ and ⟨*second command*⟩ have the same meaning, and prints an error message if they do not.

```
\cs_new_protected_nopar:Npn \lltxmath_assert_eq:NN #1 #2 {
  \cs_if_eq:NNF #1 #2 {
    \msg_error:nnxxxx { lualatex-math } { different-meanings }
      { \token_to_str:N #1 } { \token_to_meaning:N #1 }
      { \token_to_str:N #2 } { \token_to_meaning:N #2 }
  }
}
```

\lltxmath_patch:NNnnn \lltxmath_patch:cNnnn The auxiliary macro `\lltxmath_patch:NNnnn`⟨*command*⟩⟨*factory command*⟩`{`⟨*parameter text*⟩`}{`⟨*expected replacement text*⟩`}{`⟨*new replacement text*⟩`}` tries to patch ⟨*command*⟩. If ⟨*command*⟩ is undefined, do nothing. Otherwise it must be a macro with the given ⟨*parameter text*⟩ and ⟨*expected replacement text*⟩, created by the given ⟨*factory command*⟩ or equivalent. In this case it will be overwritten using the ⟨*parameter text*⟩ and the ⟨*new replacement text*⟩. Otherwise issue a warning and don't overwrite.

```
\cs_new_protected_nopar:Npn \lltxmath_patch:NNnnn #1 #2 #3 #4 #5 {
  \cs_if_exist:NT #1 {
    \token_if_macro:NTF #1 {
      \group_begin:
      #2 \lltxmath_temp:w #3 { #4 }
      \cs_if_eq:NNTF #1 \lltxmath_temp:w {
        \msg_info:nnx { lualatex-math } { patch-macro }
          { \token_to_str:N #1 }
        \group_end:
        #2 #1 #3 { #5 }
      } {
        \msg_warning:nnxxx { lualatex-math } { wrong-meaning }
          { \token_to_str:N #1 } { \token_to_meaning:N #1 }
          { \token_to_meaning:N \lltxmath_temp:w }
        \group_end:
      }
    } {
```

```
      \msg_warning:nnx { lualatex-math } { macro-expected }
        { \token_to_str:N #1 }
    }
  }
}
\cs_generate_variant:Nn \lltxmath_patch:NNnnn { c }
```

\lltxmath_set_mathchar:NN The macro `\lltxmath_set_mathchar:NN`⟨*control sequence*⟩⟨*token*⟩ defines the ⟨*control sequence*⟩ as an extended mathematical character shorthand whose mathematical code is given by the mathematical code of the character `⟨*token*⟩. Since there is no `\Umathcharnumdef` primitive, we have to extract the class, family, and slot numbers separately.

```
\cs_new_protected_nopar:Npn \lltxmath_set_mathchar:NN #1 #2 {
  \luatexUmathchardef #1
  \lua_now_x:n {
    lualatex.math.print_class_fam_slot( \int_eval:n { `#2 } )
  }
  \scan_stop:
}
```

3.5 LaTeX 2ε kernel

In LuaTeX, we have 256 math families at our disposal. Therefore we modify the LaTeX allocation macros `\newfam` and `\new@mathgroup` accordingly.

First we test whether `\newfam` and `\new@mathgroup` are equal.

```
\lltxmath_assert_eq:NN \newfam \new@mathgroup
```

\new@mathgroup It is enough to modify the maximum number of families known to the allocation system; the macro `\alloc@` takes care of the rest. This would work even if the etex package weren't loaded.

```
\lltxmath_patch:NNnnn \new@mathgroup \cs_set_nopar:Npn { } {
  \alloc@ 8 \mathgroup \chardef \sixt@@n
} {
  \alloc@ 8 \mathgroup \chardef \c_two_hundred_fifty_six
}
```

\newfam We have to reset `\newfam` to equal `\new@mathgroup`.

```
\cs_set_eq:NN \newfam \new@mathgroup
```

LuaTeX enables access to the current mathematical style via the `\mathstyle` primitive. For this to work, fraction-like constructs (e. g., ⟨*numerator*⟩ `\over` ⟨*denominator*⟩) have to be enclosed in a `\Ustack` group. `\frac` can be patched to do this, but the plain TeX remnants `\choose`, `\brack` and `\brace` should be discouraged.

\luatexUstack First we make sure that we can use the `\Ustack` primitive (under the name `\luatexUstack`).

```
\luatexbase@ensure@primitive { Ustack }
```

\frac Here we assume that nobody except amsmath redefines `\frac`. This is obviously not the case, but we ignore other packages (e. g., nath) for the moment. We only patch the LaTeX 2ε kernel definition if the amsmath package is not loaded; the corresponding patch for amsmath follows below.

```
\AtEndPreamble {
  \@ifpackageloaded { amsmath } { } {
    \lltxmath_patch:NNnnn \frac \cs_set_nopar:Npn { #1 #2 } {
```

```
      {
        \begingroup #1 \endgroup \over #2
      }
    } {
```

To do: do we need the additional set of braces around \Ustack?

```
      {
        \luatexUstack { \group_begin: #1 \group_end: \over #2 }
      }
    }
  }
}
```

3.6 amsmath

The popular amsmath package is subject to three LuaTEX-related problems:

- The \mathcode primitive is used several times, which fails for Unicode math characters. \Umathcode should be used instead.
- Legacy font dimensions are used for constructing stacks in the \substack command and the subarray environment. This doesn't work if a Unicode math font is selected.
- The fraction commands \frac and \genfrac don't use the \Ustack primitive.

\luatexalignmark
\luatexUstartmath
\luatexUstopmath

We use the primitives corresponding to the alignment mark (#) and to the inline math switches; this is more semantical and might lead to better error messages.

```
\luatexbase@ensure@primitive { alignmark }
\luatexbase@ensure@primitive { Ustartmath }
\luatexbase@ensure@primitive { Ustopmath }
```

\luatexUmathstacknumup
\luatexUmathstackdenomdown
\luatexUmathstackvgap

Now we require the font parameters we will use.

```
\luatexbase@ensure@primitive { Umathstacknumup }
\luatexbase@ensure@primitive { Umathstackdenomdown }
\luatexbase@ensure@primitive { Umathstackvgap }
```

\c_lltxmath_std_minus_mathcode_int
\c_lltxmath_std_equal_mathcode_int

These constants contain the standard TEX mathematical codes for the minus and the equal signs. We temporarily set the math codes to these constants before loading the amsmath package so that it can request the legacy math code without error.

```
\int_const:Nn \c_lltxmath_std_minus_mathcode_int { "2200 }
\int_const:Nn \c_lltxmath_std_equal_mathcode_int { "303D }
```

\lltxmath_char_dim:NN

The macro \lltxmath_char_dim:NN⟨*primitive*⟩⟨*token*⟩ expands to a ⟨*dimen*⟩ whose value is the metric of the mathematical character corresponding to the character `⟨*token*⟩ specified by ⟨*primitive*⟩, which must be one of \fontcharwd, \fontcharht or \fontchardp, in the currently selected text style font.

```
\cs_new_nopar:Npn \lltxmath_char_dim:NN #1 #2 {
  #1 \textfont
  \lua_now_x:n {
    lualatex.math.print_fam_slot( \int_eval:n { `#2 } )
  }
}
```

\l_lltxmath_minus_mathchar
\l_lltxmath_equal_mathchar

These mathematical characters are saved before amsmath is loaded so that we can temporarily assign the TEX values to the mathematical codes of the minus and equals signs. The amsmath package queries these codes, and if they represent

Unicode characters, the package loading will fail. If amsmath has already been loaded, there is nothing we can do, therefore we use the non-starred version of `\AtBeginOfPackageFile`.

```
\chk_if_free_cs:N \l_lltxmath_minus_mathchar
\chk_if_free_cs:N \l_lltxmath_equal_mathchar
\AtBeginOfPackageFile { amsmath } {
  \lltxmath_set_mathchar:NN \l_lltxmath_minus_mathchar \-
  \lltxmath_set_mathchar:NN \l_lltxmath_equal_mathchar \=
```

Now we temporarily reset the mathematical codes.

```
  \char_set_mathcode:nn { `\- } { \c_lltxmath_std_minus_mathcode_int }
  \char_set_mathcode:nn { `\= } { \c_lltxmath_std_equal_mathcode_int }
  \AtEndOfPackageFile { amsmath } {
```

\std@minus
\std@equals

The amsmath package defines the control sequences `\std@minus` and `\std@equal` as mathematical character shorthands while loading, but uses our restored mathematical codes, which must be fixed.

```
    \cs_set_eq:NN \std@minus \l_lltxmath_minus_mathchar
    \cs_set_eq:NN \std@equal \l_lltxmath_equal_mathchar
```

Finally, we restore the original mathematical codes of the two signs.

```
    \luatexUmathcodenum `\- \l_lltxmath_minus_mathchar
    \luatexUmathcodenum `\= \l_lltxmath_equal_mathchar
  }
}
```

All of the following fixes work even if amsmath is already loaded.

\@begindocumenthook

amsmath repeats the definiton of `\std@minus` and `\std@equal` at the beginning of the document, so we also have to patch the internal kernel macro `\@begindocumenthook` which contains the hook code.

```
\AtEndOfPackageFile * { amsmath } {
  \tl_replace_once:Nnn \@begindocumenthook {
    \mathchardef \std@minus \mathcode `\- \relax
    \mathchardef \std@equal \mathcode `\= \relax
  } {
    \lltxmath_set_mathchar:NN \std@minus \-
    \lltxmath_set_mathchar:NN \std@equal \=
  }
```

\resetMathstrut@

amsmath uses the box `\Mathstrutbox@` for struts in mathematical mode. This box is defined to have the height and depth of the opening parenthesis taken from the current text font. The command `\resetMathstrut@` is executed whenever the mathematical fonts are changed and has to restore the correct dimensions. The original definition uses a temporary mathematical character shorthand definition whose meaning is queried to extract the family and slot. We can do this in Lua; furthermore we can avoid a temporary box because ε-TEX allows us to query glyph metrics directly.

```
  \lltxmath_patch:NNnnn \resetMathstrut@ \cs_set_nopar:Npn { } {
    \setbox \z@ \hbox {
      \mathchardef \@tempa \mathcode `\( \relax % \)
      \def \@tempb ##1 "##2 ##3 { \the \textfont "##3 \char" }
      \expandafter \@tempb \meaning \@tempa \relax
    }
    \ht \Mathstrutbox@ \ht \z@
    \dp \Mathstrutbox@ \dp \z@
  } {
```

```
    \box_set_ht:Nn \Mathstrutbox@ {
      \lltxmath_char_dim:NN \fontcharht \( % \)
    }
    \box_set_dp:Nn \Mathstrutbox@ {
      \lltxmath_char_dim:NN \fontchardp \)
    }
  }
```

subarray The `subarray` environment uses legacy font dimensions. We simply patch it to use LuaTEX font parameters (and LATEX3 expressions instead of TEX arithmetic). Since subscript arrays are conceptually vertical stacks, we use the sum of top and bottom shift for the default vertical baseline distance (`\baselineskip`) and the minimum vertical gap for stack for the minimum baseline distance (`\lineskip`).

```
  \lltxmath_patch:NNnnn \subarray \cs_set:Npn { #1 } {
    \vcenter
    \bgroup
    \Let@
    \restore@math@cr
    \default@tag
    \baselineskip \fontdimen 10~ \scriptfont \tw@
    \advance \baselineskip \fontdimen 12~ \scriptfont \tw@
    \lineskip \thr@@ \fontdimen 8~ \scriptfont \thr@@
    \lineskiplimit \lineskip
    \ialign
    \bgroup
    \ifx c #1 \hfil \fi
    $ \m@th \scriptstyle ## $
    \hfil
    \crcr
  } {
    \vcenter
    \c_group_begin_token
    \Let@
    \restore@math@cr
    \default@tag
    \skip_set:Nn \baselineskip {
      \luatexUmathstacknumup \scriptstyle
      + \luatexUmathstackdenomdown \scriptstyle
    }
    \lineskip \luatexUmathstackvgap \scriptstyle
    \lineskiplimit \lineskip
    \ialign
    \c_group_begin_token
    \token_if_eq_meaning:NNT c #1 { \hfil }
    \luatexUstartmath
    \m@th
    \scriptstyle
    \luatexalignmark \luatexalignmark
    \luatexUstopmath
    \hfil
    \crcr
  }
```

\frac Since `\frac` is declared by `\DeclareRobustCommand`, we must patch the macro `\frac␣`.

```
  \lltxmath_patch:cNnnn { frac~ } \cs_set:Npn { #1 #2 } {
    {
      \begingroup #1 \endgroup \@@over #2
```

```
    }
  } {
    {
      \luatexUstack { \group_begin: #1 \group_end: \@@over #2 }
    }
  }
```

\@genfrac Generalized fractions are typeset by the internal `\@genfrac` command.

```
  \lltxmath_patch:NNnnn \@genfrac \cs_set_nopar:Npn {
    #1 #2 #3 #4 #5
  } {
    {
      #1 { \begingroup #4 \endgroup #2 #3 \relax #5 }
    }
  } {
    {
      #1 {
        \luatexUstack {
          \group_begin: #4 \group_end: #2 #3 \scan_stop: #5
        }
      }
    }
  }
}
```

3.7 mathtools

mathtools' `\cramped` command and others that make use of its internal version use a hack involving a null radical. LuaTEX has primitives for setting material in cramped mode, so we make use of them.

\luatexcrampeddisplaystyle \luatexcrampedtextstyle \luatexcrampedscriptstyle \luatexcrampedscriptscriptstyle First we make sure that the needed primitives for cramped styles are available.

```
\luatexbase@ensure@primitive { crampeddisplaystyle }
\luatexbase@ensure@primitive { crampedtextstyle }
\luatexbase@ensure@primitive { crampedscriptstyle }
\luatexbase@ensure@primitive { crampedscriptscriptstyle }
```

\MT_cramped_internal:Nn The macro `\MT_cramped_internal:Nn`⟨*style*⟩`{`⟨*expression*⟩`}` typesets the ⟨*expression*⟩ in the cramped style corresponding to the given ⟨*style*⟩ (`\displaystyle` etc.); all we have to do in LuaTEX is to select the correct primitive. Rewriting the user-level `\cramped` command and employing `\mathstyle` would be possible as well, but we avoid this way since we want to patch only a single command.

```
\AtEndOfPackageFile * { mathtools } {
  \lltxmath_patch:NNnnn \MT_cramped_internal:Nn
    \cs_set_nopar:Npn { #1 #2 } {
    \sbox \z@ {
      $
      \m@th
      #1
      \nulldelimiterspace = \z@
      \radical \z@ { #2 }
      $
    }
    \ifx #1 \displaystyle
      \dimen@ = \fontdimen 8 \textfont 3
      \advance \dimen@ .25 \fontdimen 5 \textfont 2
    \else
```

```
      \dimen@ = 1.25 \fontdimen 8
      \ifx #1 \textstyle
        \textfont
      \else
        \ifx #1 \scriptstyle
          \scriptfont
        \else
          \scriptscriptfont
        \fi
      \fi
      3
    \fi
    \advance \dimen@ -\ht\z@
    \ht\z@ = -\dimen@
    \box\z@
  } {
```

Here the additional set of braces is absolutely necessary, otherwise the changed mathematical style would be applied to the material after the `\mathchoice` construct.

```
    {
      \use:c { luatexcramped \cs_to_str:N #1 } #2
    }
  }
}
```

3.8 icomma

The icomma package uses `\mathchardef` to save the mathematical code of the comma character. This breaks for Unicode fonts. The incompatibility was noticed by Peter Breitfeld.[2]

`\mathcomma` icomma defines the mathemathical character shorthand `\icomma` at the beginning of the document, therefore we again patch `\@begindocumenthook`.

```
\AtEndOfPackageFile * { icomma } {
  \tl_replace_once:Nnn \@begindocumenthook {
    \mathchardef \mathcomma \mathcode `\,
  } {
    \lltxmath_set_mathchar:NN \mathcomma \,
  }
}
```

⟨/package⟩

4 Implementation of the LuaLaTeX module

For the Lua module, we use the standard luatexbase-modutils template and the `module` function.

```
⟨*lua⟩
require("luatexbase.modutils")
require("luatexbase.cctb")
local err, warn, info, log = luatexbase.provides_module({
  name = "lualatex-math",
  date = "2011/05/05",
  version = 0.1,
  description = "Patches for mathematics typesetting with LuaLaTeX",
```

[2] https://groups.google.com/d/topic/de.comp.text.tex/Cputk-AJS5I/discussion

```
  author = "Philipp Stephani",
  licence = "LPPL v1.3+"
})
local unpack = unpack
local string = string
local tex = tex
local cctb = luatexbase.catcodetables
module("lualatex.math")
```

print_fam_slot The function `print_fam_slot` takes one argument which must be a number. It interprets the argument as a Unicode code point whose mathematical code is printed in the form ⟨*family*⟩␣⟨*slot*⟩, suitable for the right-hand side of e. g. `\fontcharht\textfont`.

```
function print_fam_slot(char)
  local code = tex.getmathcode(char)
  local class, family, slot = unpack(code)
  local result = string.format("%i %i ", family, slot)
  tex.sprint(cctb.string, result)
end
```

print_class_fam_slot The function `print_class_fam_slot` takes one argument which must be a number. It interprets the argument as a Unicode code point whose mathematical code is printed in the form ⟨*class*⟩␣⟨*family*⟩␣⟨*slot*⟩, suitable for the right-hand side of `\Umathchardef`.

```
function print_class_fam_slot(char)
  local code = tex.getmathcode(char)
  local class, family, slot = unpack(code)
  local result = string.format("%i %i %i ", class, family, slot)
  tex.sprint(cctb.string, result)
end
⟨/lua⟩
```

5 Test files

Finally six small test files—but not a real test suite.

5.1 Common definitions

```
⟨*test⟩
\documentclass[pagesize=auto]{scrartcl}
```

Only xparse starting with 2008/08/03 has `\NewDocumentCommand`.

```
\usepackage{xparse}[2008/08/03]
\usepackage{luacode}
\ExplSyntaxOn
\AtBeginDocument { \errorcontextlines = \c_fifteen }
```

pass This message is issued when a test passed.

```
\msg_new:nnn { test } { pass } { #1 }
```

\test_pass:x The macro `\test_pass:x{`⟨*text*⟩`}` issues the `pass` message with description ⟨*text*⟩.

```
\cs_new_protected_nopar:Npn \test_pass:x #1 {
  \msg_info:nnx { test } { pass } { #1 }
}
```

fail This message is issued when a test failed.

```
\msg_new:nnn { test } { fail } { #1 }
```

\test_fail:x The macro `\test_fail:x{`⟨*text*⟩`}` issues the `fail` message with description ⟨*text*⟩.

```
\cs_new_protected_nopar:Npn \test_fail:x #1 {
  \msg_error:nnx { test } { fail } { #1 }
}
```

\tl_const:Nx We need expanding constants.

```
\cs_generate_variant:Nn \tl_const:Nn { Nx }
```

\c_test_equal_tl Two shorthands for pretty-printing test results.

\c_test_not_equal_tl

```
\tl_const:Nx \c_test_equal_tl { \c_space_tl == \c_space_tl }
\tl_const:Nx \c_test_not_equal_tl { \c_space_tl != \c_space_tl }
```

\test_equal_pass:nxnx The macro `\test_equal_pass:nxnx{`⟨*first expression*⟩`}{`⟨*first value*⟩`}{`⟨*second expression*⟩`}{`⟨*second value*⟩`}` is called when the two values arising from the two expressions are equal.

```
\cs_new_protected_nopar:Npn \test_equal_pass:nxnx #1 #2 #3 #4 {
  \test_pass:x {
    \exp_not:n { #1 }
    \c_test_equal_tl
    #2
    \c_test_equal_tl
    #4
    \c_test_equal_tl
    \exp_not:n { #3 }
  }
}
```

\test_equal_fail:nxnx The macro `\test_equal_pass:nxnx{`⟨*first expression*⟩`}{`⟨*first value*⟩`}{`⟨*second expression*⟩`}{`⟨*second value*⟩`}` is called when the two values arising from the two expressions are not equal.

```
\cs_new_protected_nopar:Npn \test_equal_fail:nxnx #1 #2 #3 #4 {
  \test_fail:x {
    \exp_not:n { #1 }
    \c_test_equal_tl
    #2
    \c_test_not_equal_tl
    #4
    \c_test_equal_tl
    \exp_not:n { #3 }
  }
}
```

\test_assert_equal:NNNNNnn \test_assert_equal:cccccnn The macro `\test_assert_equal:NNNNNnn`⟨*set command*⟩⟨*use command*⟩⟨*compare command*⟩⟨*first temporary command*⟩⟨*second temporary command*⟩`{`⟨*first expression*⟩`}{`⟨*second expression*⟩`}` asserts that the two expressions are equal. The ⟨*set command*⟩ must have the argument specification `Nn`, the ⟨*use command*⟩ `N`, and the ⟨*compare command*⟩ `nNnTF`.

```
\cs_new_protected_nopar:Npn
\test_assert_equal:NNNNNnn #1 #2 #3 #4 #5 #6 #7 {
  #1 #4 { #6 }
  #1 #5 { #7 }
  #3 { #4 } = { #5 } {
    \test_equal_pass:nxnx { #6 } { #2 #4 } { #7 } { #2 #5 }
  } {
    \test_equal_fail:nxnx { #6 } { #2 #4 } { #7 } { #2 #5 }
  }
}
\cs_generate_variant:Nn \test_assert_equal:NNNNNnn { ccccc }
```

\test_assert_equal:nnn The macro `\test_assert_equal:nnn{`⟨*data type*⟩`}{`⟨*first expression*⟩`}{`⟨*second expression*⟩`}` is a simplified version of `\test_assert_equal:NNNNNnn` for data types following the LaTeX3 naming conventions; ⟨*data type*⟩ must be `int`, `dim`, etc.

```
\cs_new_protected_nopar:Npn \test_assert_equal:nnn #1 #2 #3 {
  \test_assert_equal:cccccnn
    { #1 _set:Nn } { #1 _use:N } { #1 _compare:nNnTF }
    { l_test_tmpa_ #1 } { l_test_tmpb_ #1 } { #2 } { #3 }
}
```

\l_test_tmpa_int Scratch registers for numbers.
\l_test_tmpb_int

```
\int_new:N \l_test_tmpa_int
\int_new:N \l_test_tmpb_int
```

\AssertIntEqual The command `\AssertIntEqual{`⟨*first expression*⟩`}{`⟨*second expression*⟩`}` asserts that the two integral expressions are equal.

```
\NewDocumentCommand \AssertIntEqual { m m } {
  \test_assert_equal:nnn { int } { #1 } { #2 }
}
```

\l_test_tmpa_int Scratch registers for dimensions.
\l_test_tmpb_int

```
\dim_new:N \l_test_tmpa_dim
\dim_new:N \l_test_tmpb_dim
```

\AssertDimEqual The command `\AssertDimEqual{`⟨*first expression*⟩`}{`⟨*second expression*⟩`}` asserts that the two dimension expressions are equal.

```
\NewDocumentCommand \AssertDimEqual { m m } {
  \test_assert_equal:nnn { dim } { #1 } { #2 }
}
```

\AssertMathStyle The command `\AssertMathStyle{`⟨*expression*⟩`}` asserts that the current mathematical style is equal to the value of the integral ⟨*expression*⟩.

```
\NewDocumentCommand \AssertMathStyle { m } {
  \AssertIntEqual { \luatexmathstyle } { #1 }
}
```

\test_assert_cramped:Nx The macro `\test_assert_cramped:Nn`⟨*predicate*⟩`{`⟨*name*⟩`}` asserts that we are in math mode and that the current style fulfills the ⟨*predicate*⟩ (identified by the ⟨*name*⟩) which must have the argument specification `n`.

```
\cs_new_protected_nopar:Npn \test_assert_cramped:Nx #1 #2 {
  \int_set:Nn \l_test_tmpa_int { \luatexmathstyle }
  \bool_if:nTF {
    \int_compare_p:nNn { \l_test_tmpa_int } > { \c_minus_one }
    &&
    #1 { \l_test_tmpa_int }
  } {
    \test_pass:x {
      \exp_not:N \luatexmathstyle
      \c_test_equal_tl
      \int_use:N \l_test_tmpa_int
      \c_space_tl
      is~ a~ #2~ style
    }
  } {
    \test_fail:x {
      \exp_not:N \luatexmathstyle
      \c_test_equal_tl
      \int_use:N \l_test_tmpa_int
```

```
      \c_space_tl
      is~ not~ a~ #2~ style
    }
  }
}
```

\AssertNoncrampedStyle The command `\AssertNoncrampedStyle` asserts that the current mathematical style is one of the non-cramped styles.

```
\NewDocumentCommand \AssertNoncrampedStyle { } {
  \test_assert_cramped:Nx \int_if_even_p:n { non-cramped }
}
```

\AssertCrampedStyle The command `\AssertCrampedStyle` asserts that the current mathematical style is one of the cramped styles.

```
\NewDocumentCommand \AssertCrampedStyle { } {
  \test_assert_cramped:Nx \int_if_odd_p:n { cramped }
}
```

\l_test_tmpa_box Scratch registers for box constructions.
\l_test_tmpb_box

```
\box_new:N \l_test_tmpa_box
\box_new:N \l_test_tmpb_box
```

contains_space The function `contains_space(head, width)` returns `true` if the node list starting at `head` or any of its sublists contain a glue or kern node of width `width` (or any glue or kern node if `width` is `nil`).

```
\begin{luacode*}
function contains_space(head, width)
  for n in node.traverse(head) do
    local id = n.id
    if id == 10 or id == 11 then
      if width then
        if (id == 10 and n.spec.width == width)
        or (id == 11 and n.kern == width) then
          return true
        end
      else
        return true
      end
    elseif id == 0 or id == 1 then
      if contains_space(n.head, width) then
        return true
      end
    end
  end
  return false
end
\end{luacode*}
```

\AssertNoSpace The command `\AssertNoSpace{`⟨*text*⟩`}` asserts that the node list that is the result of typesetting ⟨*text*⟩ contains no glue or kern nodes.

```
\NewDocumentCommand \AssertNoSpace { m } {
  \hbox_set:Nn \l_test_tmpa_box { #1 }
  \int_if_odd:nTF {
    \lua_now_x:n {
      local~ b = tex.getbox(\int_use:N \l_test_tmpa_box)
      if~ contains_space(b.head) then~
        tex.sprint("0")
```

```
      else~
        tex.sprint("1")
      end
    }
  } {
    \test_pass:x {
      \tl_to_str:n { #1 } ~
      contains~ no~ skip~ or~ kern~ node
    }
  } {
    \test_fail:x {
      \tl_to_str:n { #1 } ~
      contains~ a~ skip~ or~ kern~ node
    }
  }
}
```

\AssertMuSpace The command `\AssertMuSpace{`⟨*text*⟩`}{`⟨*muskip*⟩`}` asserts that the node list that is the result of typesetting ⟨*text*⟩ contains at least one glue or kern node of with ⟨*muskip*⟩.

```
\makeatletter
\NewDocumentCommand \AssertMuSpace { m m } {
  \hbox_set:Nn \l_test_tmpa_box { #1 }
  \hbox_set:Nn \l_test_tmpb_box { $ \mskip #2 \m@th $ }
  \int_if_odd:nTF {
    \lua_now_x:n {
      local~ b = tex.getbox(\int_use:N \l_test_tmpa_box)
      local~ s = tex.getbox(\int_use:N \l_test_tmpb_box)
      if~ contains_space(b.head, s.width) then~
        tex.sprint("1")
      else~
        tex.sprint("0")
      end
    }
  } {
    \test_pass:x {
      \tl_to_str:n { #1 } ~
      contains~ a~ skip~ or~ kern~ node~ of~ width~
      \tl_to_str:n { #2 }
    }
  } {
    \test_fail:x {
      \tl_to_str:n { #1 } ~
      contains~ no~ skip~ or~ kern~ node~ of~ width~
      \tl_to_str:n { #2 }
    }
  }
}
\makeatother
\ExplSyntaxOff
```

⟨/test⟩

5.2 LaTeX 2ε kernel, allocation of math families

The LaTeX 2ε kernel itself allocates four families (also known as "math groups" in LaTeX parlance). Therefore we should still be able to allocate 252 families. We do this alternately with `\newfam`, `\new@mathgroup` and `\DeclareSymbolFont`.

```
⟨*test-kernel-alloc⟩
\usepackage{lualatex-math}
\makeatletter
\ExplSyntaxOn
\prg_stepwise_inline:nnnn { \c_four } { \c_one } {
  \c_two_hundred_fifty_five - \c_one
} {
  \prg_case_int:nnn { \int_mod:nn { #1 } { \c_three } } {
    { \c_zero } {
      \chk_if_free_cs:N \g_test_family_int
      \newfam \g_test_family_int
      \AssertIntEqual { \g_test_family_int } { #1 }
      \cs_undefine:N \g_test_family_int
    }
    { \c_one } {
      \chk_if_free_cs:N \g_test_mathgroup_int
      \new@mathgroup \g_test_mathgroup_int
      \AssertIntEqual { \g_test_mathgroup_int } { #1 }
      \cs_undefine:N \g_test_mathgroup_int
    }
    { \c_two } {
      \DeclareSymbolFont { Test #1 } { OT1 } { cmr } { m } { n }
      \exp_args:Nc \AssertIntEqual { sym Test #1 } { #1 }
    }
  } {
    \test_fail:x { This~ cannot~ happen }
  }
}
\DeclareSymbolFont { Test 255 } { OT1 } { cmr } { bx } { it }
\DeclareSymbolFontAlphabet { \TestAlphabet } { Test 255 }
\exp_args:Nc \AssertIntEqual { sym Test 255 }
  { \c_two_hundred_fifty_five }
\ExplSyntaxOff
\makeatother
\begin{document}
\[
\TestAlphabet{
  abc
  \AssertIntEqual{\fam}{255}
  \AssertIntEqual{\mathgroup}{255}
}
\]
\end{document}
⟨/test-kernel-alloc⟩
```

5.3 LaTeX 2ε kernel, \mathstyle primitive

Here we only check whether different fractions and other style-changing commands result in the correct mathematical style.

```
⟨*test-kernel-style⟩
\usepackage{lualatex-math}
\begin{document}
\begin{displaymath}
  \AssertMathStyle{0} \sqrt{\AssertMathStyle{1}}
  \frac{\AssertMathStyle{2}}{\AssertMathStyle{3}}
  a^{\frac{\AssertMathStyle{6}}{\AssertMathStyle{7}}}
  \sqrt{\frac{\AssertMathStyle{3}}{\AssertMathStyle{3}}}
  \displaystyle
```

```
  \frac{\AssertMathStyle{2}}{\AssertMathStyle{3}}
  \luatexcrampeddisplaystyle
  \frac{\AssertMathStyle{3}}{\AssertMathStyle{3}}
  \textstyle
  \frac{\AssertMathStyle{4}}{\AssertMathStyle{5}}
  \luatexcrampedtextstyle
  \frac{\AssertMathStyle{5}}{\AssertMathStyle{5}}
  \scriptstyle
  \frac{\AssertMathStyle{6}}{\AssertMathStyle{7}}
  \luatexcrampedscriptstyle
  \frac{\AssertMathStyle{7}}{\AssertMathStyle{7}}
\end{displaymath}
\begin{math}
  \AssertMathStyle{2} \sqrt{\AssertMathStyle{3}}
  \frac{\AssertMathStyle{4}}{\AssertMathStyle{5}}
  a^{\frac{\AssertMathStyle{6}}{\AssertMathStyle{7}}}
  \sqrt{\frac{\AssertMathStyle{5}}{\AssertMathStyle{5}}}
  \displaystyle
  \frac{\AssertMathStyle{2}}{\AssertMathStyle{3}}
  \luatexcrampeddisplaystyle
  \frac{\AssertMathStyle{3}}{\AssertMathStyle{3}}
  \textstyle
  \frac{\AssertMathStyle{4}}{\AssertMathStyle{5}}
  \luatexcrampedtextstyle
  \frac{\AssertMathStyle{5}}{\AssertMathStyle{5}}
  \scriptstyle
  \frac{\AssertMathStyle{6}}{\AssertMathStyle{7}}
  \luatexcrampedscriptstyle
  \frac{\AssertMathStyle{7}}{\AssertMathStyle{7}}
\end{math}
\end{document}
⟨/test-kernel-style⟩
```

5.4 amsmath and mathtools

Since mathtools loads amsmath anyway, we test both in one file.

\testbox First a scratch box register.

```
⟨*test-amsmath⟩
\usepackage{lualatex-math}
\newsavebox{\testbox}
```

We set the mathematical code for the minus sign to some arbitrary Unicode value to test whether the load-time patch works.

```
\luatexUmathcode`\-="2 "33 "44444 \relax
\usepackage{amsmath}
\AssertIntEqual{\luatexUmathcode`\-}{"33444444}
\makeatletter
\AssertIntEqual{\std@minus}{"33444444}
\makeatother
\usepackage{mathtools}
```

The same for the document begin hook.

```
\luatexUmathcode`\="5 "66 "77777 \relax
\begin{document}
\AssertIntEqual{\luatexUmathcode`\=}{"66A77777}
\makeatletter
\AssertIntEqual{\std@equal}{"66A77777}
\makeatother
```

Here we test whether the strut box has the correct height and depth.

```
\sbox{\testbox}{$($} % )
\makeatletter
\AssertDimEqual{\ht\Mathstrutbox@}{\ht\testbox}
\AssertDimEqual{\dp\Mathstrutbox@}{\dp\testbox}
\makeatother
```

Here we test for the various amsmath features that have to be patched: sub-arrays and various kind of fraction-like objects. The `\substack` command and `subarray` environment aren't really tested since it is hard to check whether the outcome looks right in an automated way. All tests are done in both inline and display mode.

```
\begin{equation*}
  \AssertMathStyle{0} \sqrt{\AssertMathStyle{1}}
  \sum_{
    \substack{\frac12 \\ \frac34 \\ \frac56}
  }
  \sum_{
    \begin{subarray}{l} \frac12 \\ \frac34 \\ \frac56 \end{subarray}
  }
  \frac{\AssertMathStyle{2}}{\AssertMathStyle{3}}
  a^{\frac{\AssertMathStyle{6}}{\AssertMathStyle{7}}}
  \dfrac{\AssertMathStyle{2}}{\AssertMathStyle{3}}
  \tfrac{\AssertMathStyle{4}}{\AssertMathStyle{5}}
  \binom{\AssertMathStyle{2}}{\AssertMathStyle{3}}
  a^{\binom{\AssertMathStyle{6}}{\AssertMathStyle{7}}}
  \dbinom{\AssertMathStyle{2}}{\AssertMathStyle{3}}
  \tbinom{\AssertMathStyle{4}}{\AssertMathStyle{5}}
  \genfrac{}{}{}{}{\AssertMathStyle{2}}{\AssertMathStyle{3}}
  \genfrac{<}{/}{0pt}{0}{\AssertMathStyle{2}}{\AssertMathStyle{3}}
  \genfrac{}{}{}{1}{\AssertMathStyle{4}}{\AssertMathStyle{5}}
  \genfrac{|}{]}{4pt}{2}{\AssertMathStyle{6}}{\AssertMathStyle{7}}
  \genfrac{}{}{}{3}{\AssertMathStyle{6}}{\AssertMathStyle{7}}
\end{equation*}
\begin{math}
  \AssertMathStyle{2} \sqrt{\AssertMathStyle{3}}
  \sum_{
    \substack{\frac12 \\ \frac34 \\ \frac56}
  }
  \sum_{
    \begin{subarray}{l} \frac12 \\ \frac34 \\ \frac56 \end{subarray}
  }
  \frac{\AssertMathStyle{4}}{\AssertMathStyle{5}}
  a^{\frac{\AssertMathStyle{6}}{\AssertMathStyle{7}}}
  \dfrac{\AssertMathStyle{2}}{\AssertMathStyle{3}}
  \tfrac{\AssertMathStyle{4}}{\AssertMathStyle{5}}
  \binom{\AssertMathStyle{4}}{\AssertMathStyle{5}}
  a^{\binom{\AssertMathStyle{6}}{\AssertMathStyle{7}}}
  \dbinom{\AssertMathStyle{2}}{\AssertMathStyle{3}}
  \tbinom{\AssertMathStyle{4}}{\AssertMathStyle{5}}
  \genfrac{}{}{}{}{\AssertMathStyle{4}}{\AssertMathStyle{5}}
  \genfrac{<}{/}{0pt}{0}{\AssertMathStyle{2}}{\AssertMathStyle{3}}
  \genfrac{}{}{}{1}{\AssertMathStyle{4}}{\AssertMathStyle{5}}
  \genfrac{|}{]}{4pt}{2}{\AssertMathStyle{6}}{\AssertMathStyle{7}}
  \genfrac{}{}{}{3}{\AssertMathStyle{6}}{\AssertMathStyle{7}}
\end{math}
```

Since mathtools' `\cramped` command uses `\mathchoice`, we cannot test for a single mathematical style since all of them are executed; instead, we just verify that all styles encountered are cramped.

```
\begin{equation*}
  \AssertMathStyle{0}
  a^{\AssertMathStyle{4} a}
  \cramped{\AssertCrampedStyle a^{\AssertCrampedStyle a}}
  a^{
    \AssertMathStyle{4}
    a^a
    \cramped{\AssertCrampedStyle a^{\AssertCrampedStyle a}}
    a^a
    \AssertMathStyle{4}
  }
  a^{
    a^{
      \AssertMathStyle{6}
      a^a
      \cramped{\AssertCrampedStyle a^{\AssertCrampedStyle a}}
      a^a
      \AssertMathStyle{6}
    }
  }
  a^{\AssertMathStyle{4} a}
  \AssertMathStyle{0}
\end{equation*}
\begin{math}
  \AssertMathStyle{2}
  a^{\AssertMathStyle{4} a}
  \cramped{\AssertCrampedStyle a^{\AssertCrampedStyle a}}
  a^{
    \AssertMathStyle{4}
    a^a
    \cramped{\AssertCrampedStyle a^{\AssertCrampedStyle a}}
    a^a
    \AssertMathStyle{4}
  }
  a^{
    a^{
      \AssertMathStyle{6}
      a^a
      \cramped{\AssertCrampedStyle a^{\AssertCrampedStyle a}}
      a^a
      \AssertMathStyle{6}
    }
  }
  a^{\AssertMathStyle{4} a}
  \AssertMathStyle{2}
\end{math}
\end{document}
⟨/test-amsmath⟩
```

5.5 unicode-math

This test file loads both amsmath and unicode-math. The latter package contains fixes that somewhat overlap with ours. We have to take care in all packages that no attempt is made to patch a single macro twice. Therefore we treat warnings (that occur when trying to patch a macro with an unknown meaning) as errors here. However, the auxiliary package fontspec-patches uses `\RenewDocumentCommand` from the xparse package, which generates a warning that we don't want to turn into an error.

Therefore we treat the offending message `redefine-command` specially.

```
⟨*test-unicode⟩
\ExplSyntaxOn
\msg_redirect_class:nn { warning } { error }
\msg_redirect_name:nnn { LaTeX } { xparse / redefine-command } { info }
\ExplSyntaxOff
\usepackage{amsmath}
\usepackage{unicode-math}[2011/05/05]
\setmathfont{XITS Math}
\usepackage{lualatex-math}
\begin{document}
\begin{equation*}
  \AssertMathStyle{0} \sqrt{\AssertMathStyle{1}}
  \frac{\AssertMathStyle{2}}{\AssertMathStyle{3}}
  a^{\frac{\AssertMathStyle{6}}{\AssertMathStyle{7}}}
  \dfrac{\AssertMathStyle{2}}{\AssertMathStyle{3}}
  \tfrac{\AssertMathStyle{4}}{\AssertMathStyle{5}}
\end{equation*}
\end{document}
⟨/test-unicode⟩
```

5.6 icomma without unicode-math

This test file loads only icomma to test whether our patch works for Computer Modern.

```
⟨*test-icomma⟩
\usepackage{lualatex-math}
\usepackage{icomma}
\begin{document}
$1,234 \; (x, y)$
\AssertNoSpace{$1,234$}
\AssertMuSpace{$(x, y)$}{\thinmuskip}
\AssertIntEqual{\mathcomma}{"1C0003B}
\end{document}
⟨/test-icomma⟩
```

5.7 icomma with unicode-math

This test file loads both icomma and unicode-math to test whether they interact well.

```
⟨*test-icomma-unicode⟩
\usepackage{unicode-math}[2011/05/05]
\setmathfont{XITS Math}
\usepackage{lualatex-math}
\usepackage{icomma}
\begin{document}
$1,234 \; (x, y)$
\AssertNoSpace{$1,234$}
\AssertMuSpace{$(x, y)$}{\thinmuskip}
\AssertIntEqual{\mathcomma}{"0C0002C}
\end{document}
⟨/test-icomma-unicode⟩
```

Change History

Index

Numbers written in italic refer to the page where the corresponding entry is described; numbers underlined refer to the code line of the definition; numbers in roman refer to the code lines where the entry is used.

Symbols

A

B

C

D

E

F

G

H

I

L

M

N

O

P

R

S

T

U

V

W

X

Z