Version: 0.1.1.

The logictools Package

Miles Min Yin Cheang

May 20, 2025

Contents

1 Purpose of this package

The star of the show here is the formallogic environment. Prior to the development of this environment, spending way too much time fiddling around with spacing commands was a familiar experience for every logician. Most of the spacing you need in a logical statement is context sensitive, so only so much can be done through basic macros. Furthermore, using too many macros destroys the readability of the code, and slows down writing to a crawl.

In an effort to change this, I wrote an environment that both *speeds up* writing formal logic (by offering shorter syntax) and improves the output considerably. The details of how this works will be presented in the upcoming sections. The default settings were made with LaTeX's default math font in mind, with the intention that the user come up with a preset that matches their preferences. The options can be changed on the fly, so more than one preset can be used in different parts of the document.

Other than this, the option 'oxford' will load a few neat macros that might be of particular interest to those studying logic at the University of Oxford; they provide shortcuts to notations that are commonly used in the first-year courses. It is likely that this section of the package will be updated with more content as I go through my degree.

2 The formallogic environment

2.1 Introduction

This interface, accessed through the environment named `formallogic`, or the command `\fmllgc{<content>}`, helps to type formal logic in LaTeX. Here is a demo[1]:

<table>
<tr><th>Code:</th><th>Output:</th><th>Default LaTeX:</th></tr>
<tr><td><code>|forall, x ; exists, y| (Ryx)</code></td><td>$\forall x \; \exists y \; (Ryx)$</td><td>$\forall x \exists y (Ryx)$</td></tr>
<tr><td><code>|f,x;e,y|(Ryx)</code></td><td>$\forall x \; \exists y \; (Ryx)$</td><td>$\forall x \exists y (Ryx)$</td></tr>
<tr><td><code>((P \land Q) \liff R)</code></td><td>$\llparenthesis \llparenthesis\, P \land Q \,\rrparenthesis \leftrightarrow R \,\rrparenthesis$</td><td>$\llparenthesis \llparenthesis P \land Q \rrparenthesis \leftrightarrow R \rrparenthesis$</td></tr>
<tr><td><code>((P \land Q) \liff R)</code></td><td>$((P \land Q) \leftrightarrow R)$</td><td>$((P \land Q) \leftrightarrow R)$</td></tr>
<tr><td><code>((P \land Q) \liff R)</code></td><td>$(\,(\,P \land Q\,) \leftrightarrow R\,)$</td><td>$((P \land Q) \leftrightarrow R)$</td></tr>
</table>

The first and foremost difference that can be observed is the readability of the code. Certainly where we wish to use $\llparenthesis \cdots \rrparenthesis$, the code for the default LaTeX output looks like:

```
$\llparenthesis\llparenthesis P\land Q\rrparenthesis\liff R\rrparenthesis$
```

Which can be shortened with shortcut macros but is still essentially unreadable. The case is arguably even worse for quantifiers, where we need to achieve nice spacing:

`$\forall\:\! x\ \exists\, y \ \, (Ryx)$` (What a mess!)

$\rightarrow \forall x \; \exists y \;\; (Ryx)$

vs. $\forall x \; \exists y \;\; (Ryx)$ (from formallogic)

In the table, `forall` and `exists` reference direct copies of `\forall` and `\exists`. Logictools declares these two quantifiers by default. They will inherit the ugly kerning present for these symbols in LaTeX's default math font (see table row 1). Good math fonts will not have this bug, and so fixing it will not be necessary in either default LaTeX or in formallogic. For the above we declared quantifiers[2] `f` and `e` with better spacing than the LaTeX default (as seen in row 2). The resulting syntax is clearly superior to what can be achieved without this environment.

Note that there are various user-defined parameters controlling the typesetting (e.g. spacing, kerning, parenthesis style); this is how the same code produces markedly different outputs. Furthermore, the user can customise some of the syntax (e.g. the names of quantifiers). This means that just a few lines of code suffice to set the style and syntax for an entire document. The benefits of this approach over manual typesetting should be obvious.

---

[1] Zooming in is recommended.

[2] The next section covers quantifier declaration in detail

More useful syntactic shortcuts may be found in the OTHER SYNTAX section. The next section covers quantifiers in more detail (primarily, quantifier syntax and declaration).

2.2 Quantifier stacks

Quantifier stacks are a concept introduced for typesetting logical quantifiers:

$$\forall x\ \exists y\ \forall x_1\ \exists z \in \mathbb{R}$$

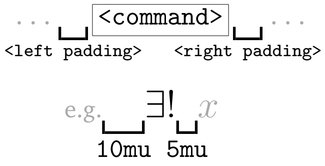
```
| forall, x ; exists, y ; forall, x_1 ; exists, z\in\mathbb{R} |
```

Quantifier stacks are used by the formallogic environment. They are delimited by '`|`'. A quantifier stack is made up of quantifiers, written in the form `<label>,<argument>`. The label consists of some text that indicates which quantifier will be used, while the argument can be any math mode code. These quantifiers are separated by ';'. The formallogic environment processes these stacks, turning them into a fully typeset sequence of quantifiers.

Spacing on either side of the label and argument is trimmed, but spacing inside the label is not. This means 'for all' is a distinct label from 'forall'. Officially, only Latin alphabetical characters are supported for labels; although, other characters will *probably* work as long as they can be used in expl3 csnames.

2.2.1 Declaring quantifiers

A declared quantifier has the following form:



Syntax for quantifier declaration:

```
\DeclareQuantifier{<label>}{<command>}[<left pad>][<right pad>]
```

This command globally declares (or redeclares) a quantifier with the associated properties. The label consists of some text that refers to this quantifier; the command should be a LaTeX command providing the quantifier symbol; left and right padding are optional padding values on either side of the command. For instance: `\DeclareQuantifier{ex!}{\exists !}[5mu][1.5mu]` provides a quantifier that can be used like: `|ex!,x;forall,y|` → $\exists! x\ \forall y \ldots$

```
\LDeclareQuantifier{<label>}{<command>}[<left pad>][<right pad>]
```

This command does the same thing, but locally. With this, one can quickly redefine a quantifier in the middle of an environment, and not worry about the changes carrying over to the rest of the document.

[!] Local definitions overwrite any global definitions until their group is closed. Thus, if one writes:

```
\begingroup
\LDeclareQuantifier{ex}{\exists}[1mu][1mu]
\begin{formallogic}
    \DeclareQuantifier{ex}{\forall}
    |ex,x|(Px)
\end{formallogic}
\endgroup |
\fmllgc{|ex,x|(Px)}
```

The output is: $\exists x\ (Px)$ | $\forall x\ (Px)$, since the global `\DeclareQuantifier{ex}{\forall}` does not come into effect until after the group is closed. As a word of warning, this means that using `\LDeclareQuantifier` at the top level will make the declared quantifier immune to change by `\DeclareQuantifier` for the rest of the document, as the group never closes.

2.3 Typesetting features

2.3.1 Customisation

The formallogic environment offers many customisation options through user-adjustable keys. They may be changed with the command `\logictoolsoptions{<key>=<value>, ... }` (in either the document or the preamble). A list of key-value pairs may also be given in an optional argument to the formallogic environment or command. The following keys are available:

Key	Description	Accepts	Default
`partype`	Determines the type of parenthesis used, single ‘(...)’ or double ‘⦅ ... ⦆’.	`single`, `double`	`single`
`parinnerpad`	Extra space inserted between parentheses and their content.	mu	`0.9mu`
`parstackkern`	Kern applied to stacked parentheses.	mu	`-0.9mu`
`italiccorrection`	Extra kern between closing parentheses and their content, to offset italic math font.	mu	`1.12mu`
`parvoffset`	Amount to raise parentheses by; helps center them on text in some fonts.	ex	`0.2ex`
`quantskip`	Default skip inserted between quantifiers.	mu	`4.32mu`
`lastquantskip`	Default skip inserted after last quantifier.	mu	`4.32mu`
`scriptspace`	Determines space after sub/superscript, same as the LaTeX primitive.	em	`-0.025em`

2.3.2 Other syntax

- Parentheses written consecutively (without spaces) will become parenthesis stacks, and use `parstackkern` instead of `parinnerpad` as spacing.
- `[<arg 1>/<arg 2>]` → $[{}^{<1>}/_{<2>}]$, allowing one to write easy variable substitutions inline[3]. This uses the package `xfrac`, so fraction appearance is changed through that interface[4].
- `.=` → $\doteq$, providing quick access to `\doteq`.

One can use `"<content>"` within the environment to escape `<content>`, preventing it from being parsed by the environment; this is useful when one wishes to use a character that is active in the syntax of the environment. The delimiter used here is the double quote, `"` (U+0022).

---

[3] Note that this means ‘`[`’ is by default active in the syntax, and so requires escaping if one wishes to use it without following it with ‘`/`’ and then ‘`]`’.

[4] `xfrac` likes to generate warnings about font size substitutions with some fonts; loading a package like `anyfontsize` should fix this.

For example, this can be used to write a function with single parentheses in a double parenthesis environment, or a list using commas inside of a quantifier stack[5]:

```
\logictoolsoptions{partype=double, parinnerpad=3.5mu}
\begin{formallogic}
|forall,"x_1,x_2,x_3,\ldots";exists,y|(f"(y)"=x_1+x_2+x_3+\ldots\land Py)
\end{formallogic}
```

Produces: $\forall x_1, x_2, x_3, \ldots\ \exists y\ \llparenthesis\, f(y) = x_1 + x_2 + x_3 + \ldots \land Py \,\rrparenthesis$

---

[5] To be exact, ‘,’ and ‘;’ only need escaping when inside of a quantifier stack, delimited by ‘|’.

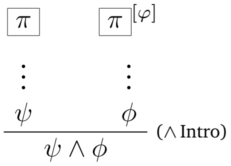
3 The ‘oxford’ package option

This package option adds a few macros for common notations at University of Oxford.

3.1 Good looking ‘proof from φ to ψ’

‘A proof π from φ to ψ’ (perhaps with some discharged assumptions) might be notated like this:

```
\begin{prooftree}
    \alwaysNoLine
    \AxiomC{\fbox{$\pi$}}
    \UnaryInfC{$\vdots$}
    \UnaryInfC{$\psi$}
    \AxiomC{\fbox{$\pi$}$^{[\varphi]}$}
    \UnaryInfC{$\vdots$}
    \UnaryInfC{$\phi$}
    \alwaysSingleLine
    \andlabel{Intro}[6]
    \BinaryInfC{$\psi \land \phi$}
\end{prooftree}
```



The output is not ideal; introducing discharged assumptions puts the box off center in an annoying way, and the `\vdots` are not aligned correctly. The following command achieves better output[7] with nicer syntax:

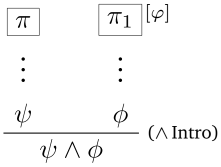
`\prooffrom{<1>}<2>{<3>}`

<2> = Either ‘`^`’ (for superscript), ‘`_`’ (for subscript), or nothing (for centered script).

followed by...

Some `content` delimited by `[···]` (for square-bracketed content) or `<···>` (for non-square-bracketed content).

E.g. `\prooffrom{$\pi$}{$\psi$}`, `\prooffrom{$\pi_1$}^[$\varphi$]{$\phi$}` give:



---

[6] The macro `\andlabel{#1}` gives `\RightLabel{\scriptsize($\land$\hspace{1px}#1)}`.

[7] Note, if one loads something that redefines `$\vdots$`, its vertical spacing might get ugly (as of logictools v0.1.1).

3.2 Bits and Bobs

[Math mode only.] `\difmost{<variable>}`

Gives the variable assignment notation: $\alpha \overset{v}{\sim} \beta$, meaning ‘$\beta$ differs from α in at most v’.

`\lcma`

Gives ɔ, the ‘logical comma’ that Professor Beau Mount uses in the PTLP lecture notes.

`\semval{<sent.>}{<structure>}[<var.assign.>]`

Gives $|\texttt{<sent.>}|^{\alpha}_{\mathcal{A}}$, the semantic value of some sentence over model $\mathcal{A}$ with variable assignment α. The input `<structure>` is converted to `\mathcal{...}`. If the input `<var.assign.>` is a single latin letter (e.g. ‘a’, ‘b’, ‘d’ ‘g’), it is converted into an appropriate greek one[8].

`\lsym{<language>}[<signature>]`[9]

Gives $\mathcal{L}_\circ$, where $\circ$ can be $1,2,=$ or something else. Also optionally allows the addition of a superscript, for a signature. The ‘=’ uses `\@ltoolsshorteq`, ‘=’, which is prettier in most fonts.

---

[8] This respects capitalisation, so one gets γ from ‘g’, and Γ from ‘G’. The command used is `\latinletterstogreek`.

[9] This command loads even without the package option ‘oxford’. Why? Because I couldn’t get it to work otherwise.

4 Implementation

4.1 formallogic

First, we save the original meaning of each symbol used in the syntax to its own csname. We do this on startup instead of at the beginning of every environment because it saves on performance, and 99% of the time it won't matter. This does mean that if some other package messes with the definition of these symbols, logictools will not notice; the user can fix this if they so wish by updating the definition of `\__formal_original_X:` manually.

```
% Thanks to everyone at TeX-exchange for teaching me how to use expl3!
\ExplSyntaxOn
\exp_args:Nc \mathchardef { __formal_original_): }=\char_value_mathcode:n {`)}
\exp_args:Nc \mathchardef { __formal_original_(: }=\char_value_mathcode:n {`(}
\exp_args:Nc \mathchardef { __formal_original_|: }=\char_value_mathcode:n {`|}
\exp_args:Nc \mathchardef { __formal_original_;: }=\char_value_mathcode:n {`;}
\exp_args:Nc \mathchardef { __formal_original_.: }=\char_value_mathcode:n {`.}
\exp_args:Nc \mathchardef { __formal_original_[: }=\char_value_mathcode:n {`[}
```

Then we initialise some error message variables, and token lists for our parentheses:

```
\msg_new:nnnn { logictools } { quantnoglobaldeferror }{}{}
% msg if global def for a quantifier went missing
\msg_new:nnnn { logictools } { quantneverdeferror }{}{}
% msg if quant is never defined at all
\tl_new:N \l__formal_rparchar_tl % the right parenthesis used in formallogic
\tl_new:N \l__formal_lparchar_tl % the left parenthesis used in formallogic
```

Now we define all the keys for customisation, and initialise them:

```
\keys_define:nn {formal/options}
 {
   ...
 }
\keys_set:nn {formal/options}
    {
    parstackkern,
    parinnerpad,
    italiccorrection,
    parvoffset,
    quantskip,
    lastquantskip,
    partype,
    scriptspace,
    }
```

Now we set up some booleans:

```
\bool_new:N \l__formal_rparpadreq_bool
\bool_set_true:N \l__formal_rparpadreq_bool
\bool_new:N \l__formal_escaped_bool
\bool_set_false:N \l__formal_escaped_bool
```

`\l__formal_rparpadreq_bool` will tell us when a right parenthesis ')' needs extra padding (i.e. when it surrounds content); meanwhile, `\l__formal_escaped_bool` can be queried to tell us if we are between two quotation marks.

Next, we define commands that will be used for the left and right parentheses:

```
\box_new:N \l__formal_lpar_box % the box for parentheses
\box_new:N \l__formal_rpar_box

\cs_new_protected:Nn \__formal_llpar_char:
{
    \bool_if:NTF \l__formal_escaped_bool
    {\use:c {__formal_original_(:}}
    {
        \box_use:N \l__formal_lpar_box
        \peek_charcode:NF ( {\mskip \l__formal_parinnerpad_muskip}
    }
}

\cs_new_protected:Nn \__formal_rrpar_char:
{
    \bool_if:NTF \l__formal_escaped_bool
    {\use:c {__formal_original_):}}
    {
        \bool_if:NTF \l__formal_rparpadreq_bool
        {\mskip \l__formal_parinnerpad_muskip
        \mkern \l__formal_italiccorrection_muskip}
        {}

        \box_use:N \l__formal_rpar_box
        \peek_charcode:NTF )
        {\bool_set_false:N \l__formal_rparpadreq_bool}
        {\bool_set_true:N \l__formal_rparpadreq_bool}
    }
}
```

First, we use boxes here to take arbitrary typeset content. Eventually, we will be setting these to the correct parenthesis commands, with any extra spacing commands. Using boxes lets us skip executing these commands over and over, which saves some performance.

First, we check if we are currently escaped: if we are, we return the original parenthesis; if we aren't, we use the formallogic definition.

For the left parenthesis, we peek at the next character to see if it is another left parenthesis. If it isn't, we will put in parinnerpad. For the right parenthesis, the logic is slightly more complex. We check the current value of rparpadreq, and if it is true, we insert parinnerpad and italiccorrection. Then, we look at the next character; if this is another right parenthesis, we set rparpadreq to false; if it is something else then we set it to true. The effect is that extra padding will not be inserted if the prior character was also a right parenthesis. To round off this section on parenthesis padding, note that the formallogic will check if the first character in its input is a right parenthesis, and set rparpadreq to false in this case. This means that `\fmllgc{(}Px\fmllgc{)}` gives ⦅Px⦆, while `\fmllgc{(}Px\fmllgc{ )}` gives ⦅Px ⦆[10].

Here we define the command for ":

```
\cs_new_protected:Nn \__formal_escapetoggle:
{
    \bool_set_inverse:N \l__formal_escaped_bool
}
```

---

[10] This makes spacing behaviour consistent: spacing is inserted if and only if a character other than ')' precedes.

This next section of code implements quantifier stacks. It is quite complicated and involves a lot of trickery with math-active characters and weird arguments. The reason for this approach is that it is *very* performant compared to alternatives (e.g. l3regex, sequence splitting):

```
\cs_new_protected:Nn \__formal_quantstackenter:
{
    \bool_if:NTF \l__formal_escaped_bool
        {\use:c {__formal_original_|:}}
        {
            \begingroup
            \char_set_active_eq:nN { `| } \__formal_quantstackesc:
            \char_set_active_eq:nN { `; } \__formal_quantdivider:
            \char_set_mathcode:nn { `; } { "8000 }
            \__formal_headquant:w
        }
}
```

This will be assigned to '|'. As always, if escaped nothing happens. If not escaped, it:

1. Begins a new group.
2. Sets the next occurrence of '|' to escape the quantifier stack instead of enter it.
3. Sets ';' to its active role as the divider between quantifiers.
4. Calls the command that parses the leading quantifier (headquant).

Headquant does the following:

```
\cs_new_protected:Npn \__formal_headquant:w #1,
{
 \tl_trim_spaces_apply:nN {#1} \__formal_headbox:n
}
```

Here we are using the 'weird' argument specification to eat the comma as part of the argument. As such, this command takes this section of the input: `|<=#1=>, ...|` and discards the comma from the input stream. It then trims any spaces on either side of this input and passes the result to the 'headbox' command, defined as follows:

```
\cs_new_protected:Npn \__formal_headbox:n #1
{
```

```
\box_if_exist:cTF {l__formal_#1head_box}
  {
    \box_if_empty:cTF {l__formal_#1head_box}
      {
        \box_if_exist:cTF {g__formal_#1head_box}
          {\box_use:c{g__formal_#1head_box}}
          {
            \msg_set:nnnn { logictools } { quantnoglobaldeferror }
            {Quantifier~'#1'~not~defined~\msg_line_context:.}
            {You~are~attempting~to~use~a~locally~declared~quantifier
            ~outside~of~its~group.~Either~define~it~globally~with
            ~DeclareQuantifier,~or~define~it~locally~here~too.
            \\ \msg_see_documentation_text:n {logictools}}
            \msg_error:nn {logictools} {quantnoglobaldeferror}
          }
      }
      {
        \box_use:c{l__formal_#1head_box}
      }
  }
  {
    \msg_set:nnnn { logictools } { quantneverdeferror }
    {Quantifier~'#1'~used~but~never~defined.}
    {You~must~declare~quantifiers~with~the~command(s)
    ~(L)DeclareQuantifier~before~using~them.
    \\ \msg_see_documentation_text:n {logictools}}
    \msg_error:nn {logictools} {quantneverdeferror}
  }
}
```

This code takes the output from headquant and tries to find a matching declared quantifier. When quantifiers are declared, they are given associated local and global ‘headboxes’ and ‘bodyboxes’ (more on this later). Here we try to find headboxes associated with `<label>`, which will be called `\(g,l)__formal_<label>head_box`. A local box should always exist if the quantifier was declared somewhere (as both LDeclare and Declare commands instantiate this), so we first check if this one does. Then, we check if it has content. If it is empty, we revert back to the global definition of the quantifier (which hopefully exists); however, if it is non-empty, then we take the local definition over the global one. Finally, rather than throwing cryptic errors, we send out custom error messages if our existence checks fail.

All user input after the comma is processed as normal, forming the variable for the quantifier. We simply wait for ‘;’, which tells us that a new quantifier is coming.

On receiving this token, we do the same thing, but with bodyboxes:

```
\cs_new_protected:Npn \__formal_quantdivider:
{
    \bool_if:NTF \l__formal_escaped_bool
        {\use:c {__formal_original_;:}}
        {\__formal_bodyquant:w}
}

\cs_new_protected:Npn \__formal_bodyquant:w #1,
{
    \mskip \use:c{l__formal_quantskip_muskip} \tl_trim_spaces_apply:nN {#1} \__formal_bodybox:
}

\cs_new_protected:Npn \__formal_bodybox:n #1
{
```

Same as above but with every occurrence of ‘head’ replaced with ‘body’.

```
}
```

The only significant change here is that we now insert quantskip before each quantifier. Also, bodyboxes include left padding values from the declared quantifier, which headboxes do not. In retrospect, there is probably a nicer way to accomplish the same behaviour. Too bad!

Finally, the next occurrence of ‘|’ ends the group and inserts lastquantskip:

```
\cs_new_protected:Nn \__formal_quantstackesc:
{
    \endgroup \mskip \use:c{l__formal_lastquantskip_muskip}
}
```

By ending the group, we chuck out the active definition of ‘;’ and the definition of ‘|’ as quantstackesc. Thus, the next occurrence of ‘|’ enters a new quantstack.

We follow up with some more active definitions:

```
\cs_new_protected:Nn \__formal_dot:
{
    \bool_if:NTF \l__formal_escaped_bool
        {\use:c {__formal_original_.:}}
        {\peek_charcode_remove:NTF =
            {\doteq}
            {\use:c {__formal_original_.:}}}
}

\cs_new_protected:Nn \__formal_lbrack:
{
    \bool_if:NTF \l__formal_escaped_bool
        {\use:c {__formal_original_[:}}
        {\__formal_varsub:w}
}

\cs_new_protected:Npn \__formal_varsub:w #1/#2]
{
    \left["\sfrac{"#1"}{"#2"}"\right]
}
```

The definition for ‘.’ just checks if there is an equals sign afterwards, and prints $\doteq$ if there is. The definition for ‘[’ calls the command `\__formal_varsub:w`. This command uses the weird argument type to grab any input of the form `[<stuff>/<more>]`. It then converts this into a nice looking fraction using the xfrac package. Unfortunately, for some reason the definition of `\sfrac` makes syntactic use of the parentheses characters ‘(’ and ‘)’ (i.e. it cares that they have been redefined), and so the command must be escaped. However, the inputs need not be escaped, so we unescape them. This issue is something that should be looked out for when coding in this environment, though it is quite rare.

Now we get on to the formallogic environment itself:

```
\NewDocumentEnvironment{formallogic}{O{}}
{
    \setlength{\parindent}{0pt}
    \cs_set:Npn \par {$\newline$}
```

This makes `\par` work in math mode, so we can get a new line by hitting return twice.

```
    \keys_set:nn {formal/options}
        {
            #1,
        }
    \setlength\scriptspace{\l__formal_scriptspace_dim}
```

Set any key-value pairs given as an optional argument, and then set scriptspace to the value specified by the user. We do this in the environment so the assignment is local.

```
    \hbox_set:Nn \l__formal_lpar_box
        {
            \raisebox{\l__formal_parvoffset_dim}
            {$\l__formal_lparchar_tl \mkern \l__formal_parstackkern_muskip$}
        }

    \hbox_set:Nn \l__formal_rpar_box
        {
            \raisebox{\l__formal_parvoffset_dim}
            {$\mkern \l__formal_parstackkern_muskip \l__formal_rparchar_tl$}
        }
```

Sets the parenthesis boxes: We take the associated character token (declared by the partype key), raise it by parvoffset, and insert parstackkern on the correct side.

```
    \char_set_active_eq:nN { ‘( } \__formal_llpar_char:
    \char_set_mathcode:nn { ‘( } { "8000 }
    \char_set_active_eq:nN { ‘) } \__formal_rrpar_char:
    \char_set_mathcode:nn { ‘) } { "8000 }
    \char_set_active_eq:nN { ‘" } \__formal_escapetoggle:
    \char_set_mathcode:nn { ‘" } { "8000 }
    \char_set_active_eq:nN { ‘| } \__formal_quantstackenter:
    \char_set_mathcode:nn { ‘| } { "8000 }
    \char_set_active_eq:nN { ‘. } \__formal_dot:
    \char_set_mathcode:nn { ‘. } { "8000 }
    \char_set_active_eq:nN { ‘[ } \__formal_lbrack:
    \char_set_mathcode:nn { ‘[ } { "8000 }
```

Makes all the characters used in the syntax math-active and sets their definitions.

```
    \(
    \peek_charcode:NTF )
    {\bool_set_false:N \l__formal_rparpadreq_bool}
    {\bool_set_true:N \l__formal_rparpadreq_bool}
}
{\)}
```

Begins math-mode and sets rparpadreq based on if the next character is a ‘)’. At the end, we come out of math-mode.

The rest of the code is quite self-explanatory. The only thing worth noting is that DeclareQuantifier sets global variants of head and body boxes, while LDeclareQuantifier only sets local ones. This means that it is possible to locally declare a quantifier and have its definition fall out of scope by ending the group it was declared in. This will result in an empty local box, and a non-existent global box (it is also the only way for this configuration to happen); a custom error message explains this to the user if it occurs.

```
\NewDocumentCommand{\DeclareQuantifier}{m m O{0mu} O{0mu}}
{
    \box_if_exist:cF {g__formal_#1head_box}
        {
            \box_new:c {g__formal_#1head_box}
            \box_new:c {g__formal_#1body_box}
        }
    \box_if_exist:cF {l__formal_#1head_box}
        {
            \box_new:c {l__formal_#1head_box}
            \box_new:c {l__formal_#1body_box}
        }
    \hbox_gset:cn {g__formal_#1head_box}
        {
            $#2 \mkern#4$
        }
    \hbox_gset:cn {g__formal_#1body_box}
        {
            $\mskip#3 #2 \mkern#4$
        }
}
\NewDocumentCommand{\LDeclareQuantifier}{m m O{0mu} O{0mu}}
{
    \box_if_exist:cF {l__formal_#1head_box}
        {
            \box_new:c {l__formal_#1head_box}
            \box_new:c {l__formal_#1body_box}
        }
    \hbox_set:cn {l__formal_#1head_box}
        {
            $#2 \mkern#4$
        }
    \hbox_set:cn {l__formal_#1body_box}
        {
            $\mskip#3 #2 \mkern#4$
        }
}

% Use this command to declare settings that
will stay for the rest of the document!
\NewDocumentCommand \logictoolsoptions { m }
{
\keys_set:nn {formal/options}
    {
    #1
    }
```

```
}

% Command that puts things inside the formallogic environment,
can be used inline.
\NewDocumentCommand \fmllgc {o m}
{
\text{\begin{formallogic}[#1]#2
\end{formallogic}}
}
```

4.2 Everything else

NOTES FOR OTHER FUNCTIONALITY PENDING

It is worth pointing out these two things that happen right before the package finishes loading:

```
\DeclareQuantifier{exists}{\exists}
\DeclareQuantifier{forall}{\forall}

% Fix for amsmath messing with math-active codes.
\edef\originalbmathcode{%
    \noexpand\mathchardef\noexpand\@tempa\the\mathcode`\(\relax}
\def\resetMathstrut@{%
  \setbox\z@\hbox{%
    \originalbmathcode
    \def\@tempb##1"##2##3{\the\textfont"##3\char"}%
    \expandafter\@tempb\meaning\@tempa \relax
  }%
  \ht\Mathstrutbox@\ht\z@ \dp\Mathstrutbox@\dp\z@
}
```

The first is a declaration of quantifiers ‘exists’ and ‘forall’ as defaults. The final piece of code prevents a slew of errors caused by amsmath interacting with active mathcodes. I do not understand it in the slightest; I think it comes from David Carlisle.