The phfparen package[1]

Philippe Faist *philippe.faist@bluewin.ch*

August 15, 2016

---

[1] *This document corresponds to phfparen v1.0, dated 2016/08/15. It is part of the phfqitltx package suite, see https://github.com/phfaist/phfqitltx.*

---

phfparen—Parenthetic math expressions made simpler and less redundant.

■ 1 Introduction

The phfparen package provides an easier way to handle parenthetic expressions in math formulae, by reducing the amount of redundancy in the LaTeX code.

The basic command is `\paren` ⟨*optional size arg*⟩ ⟨*opening delimiter*⟩ . . . ⟨*closing delimiter*⟩. It puts the given contents (“. . . ”) within the given delimiters (such as “[...]”, “(...)” or “{...}”), and where the delimiters are sized according to the default size (no argument), the explicitly given size argument (“`\big`”), or the automatic sizing (“`*`”). As a shorthand, you may use a backtick character (“`‘`”) instead of `\paren`.

For example, you can use `‘\Big(1+\hat{f}‘\big(x))` or `\paren\Big(1+\hat{f}\paren\big(x))` instead of `\Bigl(1+\hat{f}\bigl(x\bigr)\Bigr)`, to display $\Big(1+\hat{f}\big(x\big)\Big)$. In this way, the writing is more condensed. Moreover, you don't have to repeat the size for the second delimiter, making it easier to change the delimiter sizes if you later change your mind. See further examples in subsection 2.1.

■ 2 The `\paren` command and ‘ backtick shorthand

`\paren` The basic macro of this package is `\paren`. It handles a parenthetic expression including parsing matching delimiters and adjusting their sizing.

`‘...` The backtick char provides a shorthand for specifying `\paren`. It is useful to de-clutter long formulas.

The syntax for `\paren` and `‘` is:

`\paren`⟨*optional size argument*⟩⟨*open-delimiter*⟩ ... ⟨*close-delimiter*⟩

or

`‘`⟨*optional size argument*⟩⟨*open-delimiter*⟩ ... ⟨*close-delimiter*⟩

By default, the following pairs of delimiters are recognized:
`\paren(expression in parenthesis)`,
`\paren[expression in brackets]`,
`\paren<expression in angle brackets>`,
`\paren\{expression in curly braces\}`, and
`\paren{expression in curly braces}` (alternative syntax for curly braces).
Of course, the same are available with the backtick shorthand: `‘(expression)`, `‘[expression]`, `‘<expression>`, and so on.

Delimiters are balanced, meaning that `\paren[g[x] + f[x]]` is parsed as you would expect (the expression is "`g[x] + f[x]`") and not as TeX/LaTeX usually parses default arguments (in which case the argument would be "`g[x`").

The optional size argument must be a single token. It may be one of `*`, `\big`, `\Big`, `\bigg`, `\Bigg`, or another standard sizing command. If the optional size argument is `*`, then the delimiters are sized with `\left( ... \right)`, that is, they are sized automatically.

2.1 Examples

```
\log\paren\big( a + \sin(x) -
                y)
```

$\log\big(a+\sin(x)-y\big)$

`\log‘\Big( a + \sin(x) - y)`	$\log\Big(a + \sin(x) - y\Big)$
`\log‘*( \sum_k‘\big[ a_k^\dagger - a_k ] )`	$\log\left(\sum_k \big[a_k^\dagger - a_k\big]\right)$
`F = ‘\big< f(x) >_x = ‘*< \sum g(‘<p>) >`	$F = \big\langle f(x) \big\rangle_x = \left\langle \sum g(\langle p\rangle) \right\rangle$
`‘\Big[\sum_j ‘[x]^2 - \sum [y]^2 + \sum f‘*(x_3^\dagger)^2]`	$\Big[\sum_j [x]^2 - \sum [y]^2 + \sum f\left(x_3^\dagger\right)^2\Big]$
`‘(z + \backtick x)`	$(z + `x)$
`‘\{\vec z\}`	$\{\vec z\}$
`‘{\vec z}`	$\{\vec z\}$
`‘*{ \vec z :  \sum_i z_i = 1 }`	$\left\{\vec z : \sum_i z_i = 1\right\}$
`‘\big{ \vec z :  \sum_i z_i = 1 }`	$\big\{\vec z : \sum_i z_i = 1\big\}$

2.2 Controlling the behavior of the ‘ backtick shorthand

The behavior of the backtick character is only altered in math mode. The character works as normal in text mode. (Its catcode is in fact not changed, only its mathcode.)

`\backtick` If you need a literal backtick, you may in any case use the macro `\backtick`.

You may use the `backtick` package option to enable or disable the ‘ backtick shorthand.

`backtick=true`

Enable the ‘ backtick shorthand. This is the default.

`backtick=false`

Disable the ‘ backtick shorthand. A backtick character in math mode will now simply display a backtick character.

`\parenMakeBacktickActiveParen`
`\parenMakeNormalBacktick`

You may use the command `\parenMakeBacktickActiveParen` to enable the backtick shorthand at any time in the document. Similarly, the command `\parenMakeNormalBacktick` disables the backtick shorthand. These changes are local to LaTeX groups, which means that you can temporarily disable the backtick shorthand with

```
{\parenMakeNormalBacktick Normal backtick in math mode: $‘$.}
```

■ 3 Declaring matching delimiters and corresponding representations

`\parenRegister` Register a new delimiter type with `\parenRegister`. This macro should only be called in the preamble.

Syntax is: `\parenRegister{`⟨*internal name*⟩`}{`⟨*open parse delimiter*⟩`}{`⟨*close parse delimiter*⟩`}{`⟨*open display delimiter*⟩`}{`⟨*close display delimiter*⟩`}`.

The ⟨*internal name*⟩ is just an internal identifier: use plain ascii chars please (e.g. “`angbrackets`”).

The ⟨*open parse delimiter*⟩ and ⟨*close parse delimiter*⟩ are those characters which will be used to parse the delimited expression. They may differ from the actual delimiters used to display the expression, given by `{`⟨*open display delimiter*⟩`}` and `{`⟨*close display delimiter*⟩`}`.

For example, you may use

```
\parenRegister{angbrackets}{<}{>}{\langle}{\rangle}
```

to instruct `\paren` (and `‘`) to do the following: whenever the “`<`” open delimiter is encountered, then the expression is to be parsed within the delimiters “`< ... >`,” and the final expression is to be delimited as “`\langle ... \rangle`,” where the `\langle` and `\rangle` are possibly sized according to an optional size argument.

You may repeat calls to `\parenRegister`, with different internal identifiers, to register different kinds of delimiters. It is always the opening delimiter which determines which registered info to use.

NOTE

With older versions of xparse (older than 2015/11/04), only single-character tokens or single-character escape sequences (such as `\{`) could be used as delimiters for parsing (second and third arguments to `\parenRegister`).

Otherwise, with versions of xparse newer than 2015/11/04, you may use full macros to delimit content, for example `\parenRegister{testang}{\start}{\stop}{\langle}{\rangle}` allows you to write e.g. `‘\start content...\stop`, and the content will be displayed in angle brackets.

`\parenRegsiterSimpleBraces` The case where the delimiters are a normal LaTeX group, that is “`‘{...}`,” is treated specially. In this case, you need to specify which internal identifier to fall back to. For example, if you defined `\parenRegister{braces}{\{}{\}}{\{}{\}}`, then you may use `\parenRegisterSimpleBraces{braces}` to instruct `\paren` (and `‘`) to use curly braces whenever both the forms `‘\{...\}` and `‘{...}` are used.

`\parenRegisterDefaults` Register the default set of delimiters: `‘(...)`, `‘[...]`, `‘<...>`, `‘\{...\}`, and with `‘{...}` the same as `‘\{...\}`.

There is a package option `registerdefaults` to instruct to load or not to load the default set of delimiters:

`registerdefaults=true`

Set up the default set of delimiters provided by `\parenRegisterDefaults` when loading the package. This is the default.

`registerdefaults=false`

Do not set up any delimiters. It is your job to call `\parenRegister` as necessary to register all your delimiters.

4 Implementation

Load some utility packages.

```
\RequirePackage{etoolbox}
\RequirePackage{kvoptions}
\RequirePackage{xparse}
\RequirePackage{amsmath}
\RequirePackage{mathtools}
```

4.1 Internal `\paren` API for different delimiters

The idea is to combine the power of the delimiter-parsing mechanism of xparse with the display power of paired delimiters provided by mathtools. With xparse, you can define commands which parse an argument given by arbitrary (but fixed) delimiters. The mathtools package allows you to define commands such as `\abs` with `\DeclarePairedDelimiter`, such that you can use them as `\abs{x}` for default delimiter sizes, `\abs*{x}` for automatic delimiter sizing, and `\abs[\big]{x}` to use an explicitly given size.

The idea for `\paren` is the following: detect what the open delimiter is, and relay the call to a macro which knows how to parse a balanced argument with those delimiters. That macro then relays the call to the paired-delimiter-display macro with appropriate star, optional argument and mandatory argument.

When parsing, a delimiter set is recognized by its opening delimiter token. Furthermore, to each delimiter set corresponds an internal identifier name.

A delimiter set is declared by `\parenRegister` via three macros. First, the macro `\paren@registered@delims@`⟨*open delimiter token (as* `\string`*)*⟩ is defined to expand to the internal identifier name for this delimiter pair.

Then, the macro `\paren@impl@`⟨*internal identifier name*⟩ is defined to accept one argument, the possible modifiers passed to `\paren` before the opening delimiter.

Finally, the macro `\paren@impl@`⟨*internal identifier name*⟩`@go` is a `\DeclarePairedDelimiter`-declared macro which can display the delimited expression.

4.2 Implementation of `\paren`

`\paren` Define the main `\paren` macro. The syntax is: `\paren`⟨*modifier-token*⟩⟨*delimited-argument*⟩, where modifier-token is optional and can be `*` or a sizing command such as `\big`.

Because of the way LaTeX parses arguments, we need to treat the case `\paren{...}` separately form, say, `\paren\{...\}`. Recall that `\@ifnextchar\bgroup` checks for an opening brace. Then, delegate call to appropriate macro for further processing.

```
\def\paren{%
  \@ifnextchar\bgroup\paren@impl@bgroup\paren@impl@nobgroup%
}
```

Treat the special case in which we have a single LaTeX group as argument, no modifiers. In this case, directly call the display macro corresponding to the delimiter set we should use in this case. (We know that the main argument follows in the input token chain.)

```
\def\paren@impl@bgroup{%
  \csname paren@impl@\paren@registered@default @go\endcsname%
}
```

Parse the following token. See if `#1` is an opening delimiter. Do that by seeing if `#1` is a known and registered delimiter by checking if `\paren@registered@delims@`⟨*open delimiter token as* `\string`⟩ is defined. If it is not a recognized open delimiter, assume that it is a modifier token.

```
\def\paren@impl@nobgroup#1{%
  \ifcsdef{paren@registered@delims@\string#1}{%
    \paren@impl@nomod{#1}%
  }{%
    \paren@impl@mod{#1}%
  }%
}
```

We have just parsed and recognized an open delimiter token, repeated here as argument `#1`. So get the internal delimiter identifier via `\paren@registered@delims@\string`⟨*open delim*⟩, and relay the call to `\paren@impl@`⟨*internal delim identifier*⟩. Don't forget an empty argument as we saw no modifier tokens.

```
\def\paren@impl@nomod#1{%
  \letcs\paren@tmp@delimname{paren@registered@delims@\string#1}%
  \csname paren@impl@\paren@tmp@delimname\endcsname{}#1%
  %
}
```

We have just seen a token which is not recognized as an open delimiter, so we've assumed it's a modifier token (repeated here as `#1`). Store it in `\paren@tmp@modtoken`, and parse further, keeping in mind that again we may have a LaTeX braced group instead of a single token.

```
\def\paren@impl@mod#1{%
  \def\paren@tmp@modtoken{#1}%
  \@ifnextchar\bgroup\paren@impl@modBgroup\paren@impl@modNobgroup%
}
```

We have seen a modifier token, then a LaTeX braced group. First, use `\paren@util@parseModifs` to parse the modifier token and to get the actual argument to the `\DeclarePairedDelimiter`-defined macro. Then, see which delimiter set to fall back to, and relay the call to that including the modifier flags.

```
\def\paren@impl@modBgroup{%
  \paren@util@parseModifs\paren@tmp@modifsForMathtools\paren@tmp@modtoken%
  \letcs\paren@tmp@gocmd{paren@impl@\paren@registered@default @go}%
  \expandafter\paren@tmp@gocmd\paren@tmp@modifsForMathtools%
}
```

We have seen a modifier token, stored in `\paren@tmp@modtoken`, and now there is a next token coming in (`#1`) which has to be the open delimiter (there cannot be several modifier tokens). First, check that the given delimiter is known and registered. `\paren@registered@delims@`⟨*delimiter token as* `\string`⟩ should be defined to be the internal name of the registered delimiter. Then, call the implementation for that delimiter with the modifier-token as argument, and let that do the actual parsing.

```
\def\paren@impl@modNobgroup#1{%
  \ifcsdef{paren@registered@delims@\string#1}{%
    \letcs\paren@tmp@delimname{paren@registered@delims@\string#1}%
    \letcs\paren@tmp@delimcmd{paren@impl@\paren@tmp@delimname}%
    \edef\paren@tmp@modtokenarg{{\expandonce\paren@tmp@modtoken}}%
    \expandafter\paren@tmp@delimcmd\paren@tmp@modtokenarg#1%
  }{%
```

In the event the open delimiter is not recognized, display an error message, and attempt to recover by keeping all arguments as text tokens.

```
    \PackageError{phfqit}{Unknown delimiter: (or can’t parse args?)
      {\expandafter\string\paren@tmp@modtoken}{\string#1}}{Your call
      to \string\paren couldn’t be parsed, presumably because I didn’t
      recognize your delimiter, or because there’s a bug in this
      package.}%
    \paren@tmp@modtoken#1%
  }%
}
```

4.3 Some Utilities

`\paren@util@parseModifs` Parse a modifier token, and generates the appropriate size argument tokens (star or optional argument) for a `\DeclarePairedDelimiter`-declared command.

`#1` = new macro name to be defined.

`#2` = macro containing the modifier token. (Will be expanded once.)

A call to `\paren@util@parseModifs` defines `#1` to contain mathtools-compatible size args. `#2` is expanded once, so it is expected to be a macro which contains the relevant modifiers.

```
\def\paren@util@parseModifs#1#2{%
```

If the modifier token is a star `*`, then the argument should be a simple star.

```
  \expandafter\ifstrequal\expandafter{#2}{*}{%
```

```
    \def#1{*}%
  }%
```

Otherwise, if there is something and #2 is not blank, put it in an optional argument (square braces).

```
  {% (else:)
    \expandafter\ifblank\expandafter{#2}{%
      \def#1{}%
    }{%
      \edef#1{[\expandonce{#2}]}%
    }%
  }%
}
```

`\paren@xparsefix@storeparenarg`

The second utility is a fix for a bug (or feature?) in xparse. It seems that when specifying a delimited argument with `r`⟨*open*⟩⟨*close*⟩, if the ⟨*open*⟩ delimiter is not a single character, then the argument is not properly prepared: for example, with some versions of xparse, for delimiters `r\{\}`, we get for the macro call `\mymacro\{hi there\}` as parsed argument `{hi there`. This bug was not present in older versions of xparse, where it was impossible to use full macros as delimiters, but it seemed to have been introduced in more recent versions (tracked to starting from version 2015/11/04).

So we wrap the fix into a macro `\paren@xparsefix@storeparenarg{`⟨*main argument which may need fix*⟩`}`. This defines the macro `\paren@tmp@contents` which is the actual argument, possibly fixed.

The caller must ensure that the macro `\paren@impl@tmp@delimtoken` is defined to expand to the opening delimiter token.

The default implementation assumes that xparse doesn't have the bug.

```
\def\paren@xparsefix@storeparenarg#1{%
  \def\paren@tmp@contents{#1}%
}
```

Now, redefine this macro to the correct fix depending on the xparse package version. If the version is more recent than 2015/11/04, then we need to strip macro name length (excluding backslash char) from beginning of argument.

```
\@ifpackagelater{xparse}{2015/11/04}{
  \RequirePackage{xstring}%
  \def\paren@xparsefix@storeparenarg#1{%
    \fullexpandarg\StrLen{\expandafter\string\paren@impl@tmp@delimtoken}%
        [\paren@tmp@delimstrlen]%
    \ifnum\paren@tmp@delimstrlen=1\relax%
      \def\paren@tmp@contents{#1}%
    \else%
      \def\paren@tmp@delimstrlen@{\numexpr\paren@tmp@delimstrlen-1\relax}%
      \noexpandarg\StrGobbleLeft{#1}{\the\paren@tmp@delimstrlen@}%
          [\paren@tmp@contents]%
    \fi%
  }%
}{%
}
```

4.4 Registering Delimiters

Helper to register a delimiter set for use with `\paren`.

`\parenRegister`

The macro `\parenRegister` defines the necessary macros documented in subsection 4.1. Namely, `\paren@registered@delims@`⟨*open delimiter token,* `\string`*ified*⟩ expands to the internal registered ⟨*identifier*⟩ for this delimiter. Then, defines `\paren@impl@<name>` which is a parser for this delimiter and calls the mathtools goodies etc. `\paren@impl@<name>` takes one mandatory argument, which is all the stuff to insert before the open delimiter and to pass on to mathtools (just a `*` or a delimiter size), and then reads the delimited (balanced) content.

The syntax is `\parenRegister{`⟨*identifier*⟩`}{`⟨*open delimiter token*⟩`}{`⟨*close delimiter token*⟩`}{`⟨*display open delimiter*⟩`}{`⟨*display close delimiter*⟩`}`.

This macro should only be called in the preamble.

```
\newcommand\parenRegister[5]{%
```

First, define the actual `\DeclarePairedDelimiter`-type command `\paren@impl@#1@go` which displays the final delimited expression. Recall that `#1` = the internal identifier, while `#4` and `#5` are the delimiters we should use to display the expression.

Use `\DeclarePairedDelimiterX` so that we put the main argument inside a LaTeX group, just to be sure. (For example, we want to make sure that in arguments such as `-1`, the minus sign is a unary minus and not seen as a binary operator.)

```
  \expandafter\DeclarePairedDelimiterX\csname paren@impl@#1@go\endcsname[1]{#4}{#5}{{##1}}
```

The second macro is the one which is capable of parsing the balanced, delimited expression, while first taking a mandatory argument containing the possible modifier token.

Here, we just read the modifier token and store them in a temporary macro, while relaying the delimiter parsing to a helper macro (not sure why this is necessary).

```
  \csdef{paren@impl@#1}##1{%
    \gdef\paren@impl@tmp@modifs{##1}%
    \csname paren@impl@#1@parsedelim\endcsname%
  }
  \expandafter\DeclareDocumentCommand\csname paren@impl@#1@parsedelim\endcsname{r#2#3}{%
    \letcs\paren@impl@tmp@gocmd{paren@impl@#1@go}%
%%\message{********MESSAGE*********}%
%%\message{**********\detokenize\expandafter{\paren@impl@tmp@modifs}**********}%
%%\message{**********\detokenize{##1}***********}%
    \paren@util@parseModifs\paren@impl@tmp@modifsForMathtools\paren@impl@tmp@modifs%
%%\show\paren@impl@tmp@modifsForMathtools%
    \def\paren@impl@tmp@delimtoken{#2}%
    \paren@xparsefix@storeparenarg{##1}%
%%\show\paren@impl@tmp@modifsForMathtools
%%\show\paren@tmp@contents
  \expandafter\paren@impl@tmp@gocmd\paren@impl@tmp@modifsForMathtools{\paren@tmp@contents}%
}
```

Finally, register this open delimiter to be detected by the `\paren` macro. With this, `\paren` also knows the internal identifier associated to this open delimiter token.

```
  \csdef{paren@registered@delims@\string#2}{#1}
}
```

`\parenRegisterSimpleBraces` Specify the internal name for the delimiter which should be used in the syntax `` `{...} `` or `\paren{...}` (with or without modifier token).

If this macro is called multiple times, the last specified name is used.

```
\newcommand\parenRegisterSimpleBraces[1]{%
  \def\paren@registered@default{#1}%
}
```

`\parenRegisterDefaults` A macro which registers the default set of delimiters.

```
\newcommand\parenRegisterDefaults{
```

The default set of delimiters:

```
  \parenRegister{parens}{(}{)}{(}{)}
  \parenRegister{brackets}{[}{]}{[}{]}
  \parenRegister{angbrackets}{<}{>}{\langle}{\rangle}
  \parenRegister{braces}{\{}{\}}{\{}{\}}
```

And the registered paren type to use when given the syntax `` `{...} ``: curly braces.

```
  \parenRegisterSimpleBraces{braces}
}
```

4.5 Handle backtick shorthand

`\parenMakeBacktickActiveParen` Change the behavior of the backtick character in math mode to (`\parenMakeBacktickActiveParen`) be an alias for `\paren`.

```
\def\parenMakeBacktickActiveParen{%
```

Set the backtick character to be active, but only in math mode. It still has its normal `\catcode`, only its `\mathcode` is altered.

```
\mathcode`\`="8000\relax%
```

Use the usual `\lccode`-trick to do “`` \def`{...} ``” but for an active `` ` `` char

```
  \begingroup%
  \lccode`\~=`\`%
  \lowercase{\endgroup\def~}{\paren}%
}
```

`\parenMakeNormalBacktick` The command `\parenMakeNormalBacktick` changes the behavior of the backtick character in math mode to be a normal backtick char (“‘”).

```
\def\parenMakeNormalBacktick{\mathcode`\`="0060\relax}
```

`\backtick` A literal backtick. Simply done by temporarily disabling our backtick shorthand within a LaTeX group.

```
\def\backtick{\begingroup\parenMakeNormalBacktick`\endgroup}
```

4.6 Parse package options

Set up kvoptions option parsing.

```
\SetupKeyvalOptions{
  family=phfparen,
  prefix=phfparen@opt@
}
```

Option `backtick`, and complementary option `nobacktick`.

```
\DeclareBoolOption[true]{backtick}
\DeclareComplementaryOption{nobacktick}{backtick}
```

For backwards compatibility with my previous versions of phfparen: provide the `backtickon` package option.

```
\DeclareVoidOption{backtickon}{%
  \phfparen@opt@backticktrue
  \PackageWarning{phfparen}{Option 'backtickon' is deprecated.  It still works,
      but please consider using 'backtick=true|false' instead.}%
}
```

Option `registerdefaults` to specify whether to register the default set of delimiters or not.

```
\DeclareBoolOption[true]{registerdefaults}
\DeclareComplementaryOption{noregisterdefaults}{registerdefaults}
```

Finally, process the options.

```
\DeclareDefaultOption{%
  \@unknownoptionerror
}
\ProcessKeyvalOptions*
```

Now, execute the settings dictated by the user options.

```
\ifphfparen@opt@backtick
  \parenMakeBacktickActiveParen
\fi
\ifphfparen@opt@registerdefaults
  \parenRegisterDefaults
\fi
```

Change History

v1.0
  General: Initial version 1

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.