# The fvextra package

Geoffrey M. Poore
gpoore@gmail.com
github.com/gpoore/fvextra

v1.1 from 2016/07/14

**Abstract**

fvextra provides several extensions to fancyvrb, including automatic line breaking and improved math mode. It also patches some fancyvrb internals.

# Contents


**1 Introduction** **4**

**2 Usage** **4**

**3 General options** **5**

**4 General commands** **10**
4.1 Line and text formatting . . . . . 10

**5 Line breaking** **10**
5.1 Line breaking options . . . . . 11
5.2 Line breaking and tab expansion . . . . . 15
5.3 Advanced line breaking . . . . . 16
5.3.1 A few notes on algorithms . . . . . 16
5.3.2 Breaks within macro arguments . . . . . 17
5.3.3 Customizing break behavior . . . . . 18

**6 Patches** **19**
6.1 Visible spaces . . . . . 19
6.2 `obeytabs` with visible tabs and with tabs inside macro arguments . 19
6.3 Math mode . . . . . 20
6.3.1 Spaces . . . . . 20
6.3.2 Symbols and fonts . . . . . 20
6.4 Orphaned labels . . . . . 21
6.5 `rulecolor` and `fillcolor` . . . . . 21

**7 Additional modifications to fancyvrb** **21**
7.1 Backtick and single quotation mark . . . . . 21
7.2 Line numbering . . . . . 21

**8 Undocumented features of fancyvrb** **22**
8.1 Undocumented options . . . . . 22
8.2 Undocumented macros . . . . . 22

**Version History** **22**

**9 Implementation** **23**
9.1 Required packages . . . . . 23
9.2 Utility macros . . . . . 24
9.3 Hooks . . . . . 24
9.4 Escaped characters . . . . . 24
9.5 Patches . . . . . 25
9.5.1 Visible spaces . . . . . 25
9.5.2 `obeytabs` with visible tabs and with tabs inside macro arguments . . . . . 25

9.5.3 Spacing in math mode . . . . . 29
9.5.4 Fonts and symbols in math mode . . . . . 29
9.5.5 Ophaned label . . . . . 30
9.5.6 `rulecolor` and `fillcolor` . . . . . 30
9.6 Extensions . . . . . 31
9.6.1 New options requiring minimal implementation . . . . . 31
9.6.2 Formatting with `\FancyVerbFormatLine`, `\FancyVerbFormatText`, and `\FancyVerbHighlightLine` . . . . . 32
9.6.3 Line numbering . . . . . 34
9.6.4 Line highlighting or emphasis . . . . . 38
9.7 Line breaking . . . . . 40
9.7.1 Options and associated macros . . . . . 40
9.7.2 Line breaking implementation . . . . . 47

# 1 Introduction

The fancyvrb package had its first public release in January 1998. In July of the same year, a few additional features were added. Since then, the package has remained almost unchanged except for a few bug fixes. fancyvrb has become one of the primary LaTeX packages for working with verbatim text.

Additional verbatim features would be nice, but since fancyvrb has remained almost unchanged for so long, a major upgrade could be problematic. There are likely many existing documents that tweak or patch fancyvrb internals in a way that relies on the existing implementation. At the same time, creating a completely new verbatim package would require a major time investment and duplicate much of fancyvrb that remains perfectly functional. Perhaps someday there will be an amazing new verbatim package. Until then, we have fvextra.

fvextra is an add-on package that gives fancyvrb several additional features, including automatic line breaking. Because fvextra patches and overwrites some of the fancyvrb internals, it may not be suitable for documents that rely on the details of the original fancyvrb implementation. fvextra tries to maintain the default fancyvrb behavior in most cases. All patches (section 6) and modifications to fancyvrb defaults (section 7) are documented.

Some features of fvextra were originally created as part of the pythontex and minted packages. fancyvrb-related patches and extensions that currently exist in those packages will gradually be migrated into fvextra, and both packages will require fvextra in the future.

# 2 Usage

fvextra may be used as a drop-in replacement for fancyvrb. It will load fancyvrb if it has not yet been loaded, and then proceeds to patch fancyvrb and define additional features.

The upquote package is loaded to give correct backticks (`) and typewriter single quotation marks (').  When this is not desirable within a given environment, use the option `curlyquotes`. fvextra modifies the behavior of these and other symbols in typeset math within verbatim, so that they will behave as expected (section 6.3). fvextra uses the lineno package for working with automatic line breaks. lineno gives a warning when the csquotes package is loaded before it, so fvextra should be loaded before csquotes. The ifthen and etoolbox packages are required. color or xcolor should be loaded manually to use color-dependent features.

While fvextra attempts to minimize changes to the fancyvrb internals, in some cases it completely overwrites fancyvrb macros with new definitions. New definitions typically follow the original definitions as much as possible, but code that depends on the details of the original fancyvrb implementation may be incompatible with fvextra.

## 3 General options

fvextra adds several general options to fancyvrb. All options related to automatic line breaking are described separately in section 5.

`curlyquotes` (boolean) (default: `false`)

Unlike fancyvrb, fvextra requires the upquote package, so the backtick (`) and typewriter single quotation mark (') always appear literally by default, instead of becoming the left and right curly single quotation marks (‘ ’). This option allows these characters to be replaced by the curly quotation marks when that is desirable.

| | |
|---|---|
| `\begin{Verbatim}`<br>`` `quoted text' ``<br>`\end{Verbatim}` | `` `quoted text' `` |

| | |
|---|---|
| `\begin{Verbatim}[curlyquotes]`<br>`` `quoted text' ``<br>`\end{Verbatim}` | ‘quoted text’ |

`highlightcolor` (string) (default: `LightCyan`)

Set the color used for `highlightlines`, using a predefined color name from color or xcolor, or a color defined via `\definecolor`.

`highlightlines` (string) (default: ⟨*none*⟩)

This highlights a single line or a range of lines based on line numbers. The line numbers refer to the line numbers that fancyvrb would show if `numbers=left`, etc. They do not refer to original or actual line numbers before adjustment by `firstnumber`.

The highlighting color can be customized with `highlightcolor`.

```
\begin{Verbatim}[numbers=left, highlightlines={1, 3-4}]
First line
Second line
Third line
Fourth line
Fifth line
\end{Verbatim}
```

```
1 First line
2 Second line
3 Third line
4 Fourth line
5 Fifth line
```

The actual highlighting is performed by a set of commands. These may be customized for additional fine-tuning of highlighting. See the default definition of `\FancyVerbHighlightLineFirst` as a starting point.

- `\FancyVerbHighlightLineFirst`: First line in a range.
- `\FancyVerbHighlightLineMiddle`: Inner lines in a range.
- `\FancyVerbHighlightLineLast`: Last line in a range.
- `\FancyVerbHighlightLineSingle`: Single highlighted lines.
- `\FancyVerbHighlightLineNormal`: Normal lines without highlighting.

If these are customized in such a way that indentation or inter-line spacing is changed, then `\FancyVerbHighlightLineNormal` may be modified as well to make all lines uniform. When working with the `First`, `Last`, and `Single` commands, keep in mind that fvextra merges all numbers ranges, so that `{1, 2-3, 3-5}` is treated the same as `{1-5}`.

Highlighting is applied after `\FancyVerbFormatText`, so any text formatting defined via that command will work with highlighting. Highlighting is applied before `\FancyVerbFormatLine`, so if `\FancyVerbFormatLine` puts a line in a box, the box will be behind whatever is created by highlighting. This prevents highlighting from vanishing due to user-defined customization.

`linenos` (boolean) (default: `false`)
fancyvrb allows line numbers via the options `numbers=`⟨*position*⟩. This is essentially an alias for `numbers=left`. It primarily exists for better compatibility with the minted package.

`mathescape` (boolean) (default: `false`)
This causes everything between dollar signs `$...$` to be typeset as math. The caret `^` and underscore `_` have their normal math meanings.

This is equivalent to `codes={\catcode`$=3\catcode`^=7\catcode`_=8}`. `mathescape` is always applied *before* `codes`, so that `codes` can be used to override

some of these definitions.

Note that fvextra provides several patches that make math mode within verbatim as close to normal math mode as possible (section 6.3).

**numberfirstline** (boolean) (default: `false`)

When line numbering is used with `stepnumber` $\neq 1$, the first line may not always be numbered, depending on the line number of the first line. This causes the first line always to be numbered.

```
\begin{Verbatim}[numbers=left, stepnumber=2,
                 numberfirstline]
First line
Second line
Third line
Fourth line
\end{Verbatim}
```

```
1 First line
2 Second line
  Third line
4 Fourth line
```

**numbers** (none | left | right | both) (default: `none`)

fvextra adds the `both` option for line numbering.

```
\begin{Verbatim}[numbers=both]
First line
Second line
Third line
Fourth line
\end{Verbatim}
```

```
1 First line      1
2 Second line     2
3 Third line      3
4 Fourth line     4
```

**space** (macro) (default: `\textvisiblespace`, ␣)

Redefine the visible space character. Note that this is only used if `showspaces=true`. The color of the character may be set with `spacecolor`.

**spacecolor** (string) (default: `none`)

Set the color of visible spaces. By default (`none`), they take the color of their surroundings.

```
\color{gray}
\begin{Verbatim}[showspaces, spacecolor=red]
One  two  three
\end{Verbatim}
```

---

```
One␣␣two␣␣three
```

`stepnumberfromfirst` (boolean) (default: `false`)

By default, when line numbering is used with `stepnumber` $\neq 1$, only line numbers that are a multiple of `stepnumber` are included. This offsets the line numbering from the first line, so that the first line, and all lines separated from it by a multiple of `stepnumber`, are numbered.

```
\begin{Verbatim}[numbers=left, stepnumber=2,
                 stepnumberfromfirst]
First line
Second line
Third line
Fourth line
\end{Verbatim}
```

---

```
1 First line
  Second line
3 Third line
  Fourth line
```

`stepnumberoffsetvalues` (boolean) (default: `false`)

By default, when line numbering is used with `stepnumber` $\neq 1$, only line numbers that are a multiple of `stepnumber` are included. Using `firstnumber` to offset the numbering will change which lines are numbered and which line gets which number, but will not change which *numbers* appear. This option causes `firstnumber` to be ignored in determining which line numbers are a multiple of `stepnumber`. `firstnumber` is still used in calculating the actual numbers that appear. As a result, the line numbers that appear will be a multiple of `stepnumber`, plus `firstnumber` minus 1.

This option gives the original behavior of fancyvrb when `firstnumber` is used with `stepnumber` $\neq 1$ (section 7.2).

```
\begin{Verbatim}[numbers=left, stepnumber=2,
                 firstnumber=4, stepnumberoffsetvalues]
First line
Second line
Third line
Fourth line
\end{Verbatim}
```

---

```
   First line
5  Second line
   Third line
7  Fourth line
```

**tab** (macro) (default: fancyvrb's `\FancyVerbTab`, ⇥)

Redefine the visible tab character. Note that this is only used if `showtabs=true`. The color of the character may be set with `tabcolor`.

When redefining the tab, you should include the font family, font shape, and text color in the definition. Otherwise these may be inherited from the surrounding text. This is particularly important when using the tab with syntax highlighting, such as with the minted or pythontex packages.

fvextra patches fancyvrb tab expansion so that variable-width symbols such as `\rightarrowfill` may be used as tabs. For example,

```
\begin{Verbatim}[obeytabs, showtabs, breaklines,
                 tab=\rightarrowfill, tabcolor=orange]
       ⇥First  ⇥Second ⇥Third  ⇥And more text that goes on for a
         ↪  while until wrapping is needed
       ⇥First  ⇥Second ⇥Third  ⇥Forth
\end{Verbatim}
```

```
————→First——→Second→Third——→And more text that goes on for a
         ↪  while until wrapping is needed
————→First——→Second→Third——→Forth
```

**tabcolor** (string) (default: `none`)

Set the color of visible tabs. By default (`none`), they take the color of their surroundings.

# 4 General commands

## 4.1 Line and text formatting

\FancyVerbFormatLine
\FancyVerbFormatText

fancyvrb defines \FancyVerbFormatLine, which can be used to apply custom formatting to each individual line of text. By default, it takes a line as an argument and inserts it with no modification. This is equivalent to \newcommand{\FancyVerbFormatLine}[1]{#1}.[1]

fvextra introduces line breaking, which complicates line formatting. We might want to apply formatting to the entire line, including line breaks, line continuation symbols, and all indentation, including any extra indentation provided by line breaking. Or we might want to apply formatting only to the actual text of the line. fvextra leaves \FancyVerbFormatLine as applying to the entire line, and introduces a new command \FancyVerbFormatText that only applies to the text part of the line.[2] By default, \FancyVerbFormatText inserts the text unmodified. When it is customized, it should not use boxes that do not allow line breaks to avoid conflicts with line breaking code.

```
\renewcommand{\FancyVerbFormatLine}[1]{%
  \fcolorbox{DarkBlue}{LightGray}{#1}}
\renewcommand{\FancyVerbFormatText}[1]{\textcolor{Green}{#1}}

\begin{Verbatim}[breaklines]
Some text that proceeds for a while and finally wraps onto another line
Some more text
\end{Verbatim}
```

```
Some text that proceeds for a while and finally wraps onto
↪  another line
Some more text
```

# 5 Line breaking

Automatic line breaking may be turned on with breaklines=true. By default, breaks only occur at spaces. Breaks may be allowed anywhere with breakanywhere,

[1] The actual definition in fancyvrb is \def\FancyVerbFormatLine#1{\FV@ObeyTabs{#1}}. This is problematic because redefining the macro could easily eliminate \FV@ObeyTabs, which governs tab expansion. fvextra redefines the macro to \def\FancyVerbFormatLine#1{#1} and patches all parts of fancyvrb that use \FancyVerbFormatLine so that \FV@ObeyTabs is explicitly inserted at the appropriate points.

[2] When breaklines=true, each line is wrapped in a \parbox. \FancyVerbFormatLine is outside the \parbox, and \FancyVerbFormatText is inside.

or only before or after specified characters with breakbefore and breakafter. Many options are provided for customizing breaks. A good place to start is the description of breaklines.

## 5.1 Line breaking options

Options are provided for customizing typical line breaking features. See section 5.3 for details about low-level customization of break behavior.

**breakafter** (string) (default: ⟨*none*⟩)

Break lines after specified characters, not just at spaces, when breaklines=true. For example, breakafter=-/ would allow breaks after any hyphens or slashes. Special characters given to breakafter should be backslash-escaped (usually #, {, }, %, [, ]; the backslash \ may be obtained via \\ and the space via \space).[3]

For an alternative, see breakbefore. When breakbefore and breakafter are used for the same character, breakbeforegroup and breakaftergroup must both have the same setting.

Note that when commandchars or codes are used to include macros within verbatim content, breaks will not occur within mandatory macro arguments by default. Depending on settings, macros that take optional arguments may not work unless the entire macro including arguments is wrapped in a group (curly braces {}, or other characters specified with commandchars). See section 5.3 for details.

```
\begin{Verbatim}[breaklines, breakafter=d]
some_string = 'SomeTextThatGoesOnAndOnForSoLongThatItCouldNeverFitOnOneLine'
\end{Verbatim}
```

---

```
some_string = 'SomeTextThatGoesOnAndOnForSoLongThatItCould⌋
↪  NeverFitOnOneLine'
```

**breakaftergroup** (boolean) (default: true)

When breakafter is used, group all adjacent identical characters together, and only allow a break after the last character. When breakbefore and breakafter are used for the same character, breakbeforegroup and breakaftergroup must both have the same setting.

**breakaftersymbolpre** (string) (default: \,\footnotesize\ensuremath{_\rfloor}, ⌋)

The symbol inserted pre-break for breaks inserted by breakafter.

**breakaftersymbolpost** (string) (default: ⟨*none*⟩)

[3] breakafter expands each token it is given once, so when it is given a macro like \%, the macro should expand to a literal character that will appear in the text to be typeset. fvextra defines special character escapes that are activated for breakafter so that this will work with common escapes. The only exception to token expansion is non-ASCII characters under pdfTeX; these should appear literally. breakafter is not catcode-sensitive.

The symbol inserted post-break for breaks inserted by `breakafter`.

**breakanywhere** (boolean) (default: `false`)
Break lines anywhere, not just at spaces, when `breaklines=true`.

Note that when `commandchars` or `codes` are used to include macros within verbatim content, breaks will not occur within mandatory macro arguments by default. Depending on settings, macros that take optional arguments may not work unless the entire macro including arguments is wrapped in a group (curly braces `{}`, or other characters specified with `commandchars`). See section 5.3 for details.

```
\begin{Verbatim}[breaklines, breakanywhere]
some_string = 'SomeTextThatGoesOnAndOnForSoLongThatItCouldNeverFitOnOneLine'
\end{Verbatim}
```

---

```
some_string = 'SomeTextThatGoesOnAndOnForSoLongThatItCouldNeve⌋
↪  rFitOnOneLine'
```

**breakanywheresymbolpre** (string) (default: `\,\footnotesize\ensuremath{_\rfloor}, ⌋`)
The symbol inserted pre-break for breaks inserted by `breakanywhere`.

**breakanywheresymbolpost** (string) (default: ⟨*none*⟩)
The symbol inserted post-break for breaks inserted by `breakanywhere`.

**breakautoindent** (boolean) (default: `true`)
When a line is broken, automatically indent the continuation lines to the indentation level of the first line. When `breakautoindent` and `breakindent` are used together, the indentations add. This indentation is combined with `breaksymbolindentleft` to give the total actual left indentation.

**breakbefore** (string) (default: ⟨*none*⟩)
Break lines before specified characters, not just at spaces, when `breaklines=true`. For example, `breakbefore=A` would allow breaks before capital A's. Special characters given to `breakbefore` should be backslash-escaped (usually `#`, `{`, `}`, `%`, `[`, `]`; the backslash `\` may be obtained via `\\` and the space via `\space`).[4]

For an alternative, see `breakafter`. When `breakbefore` and `breakafter` are used for the same character, `breakbeforegroup` and `breakaftergroup` must both have the same setting.

Note that when `commandchars` or `codes` are used to include macros within verbatim content, breaks will not occur within mandatory macro arguments by default. Depending on settings, macros that take optional arguments may not work

[4] `breakbefore` expands each token it is given once, so when it is given a macro like `\%`, the macro should expand to a literal character that will appear in the text to be typeset. fvextra defines special character escapes that are activated for `breakbefore` so that this will work with common escapes. The only exception to token expansion is non-ASCII characters under pdfTeX; these should appear literally. `breakbefore` is not catcode-sensitive.

unless the entire macro including arguments is wrapped in a group (curly braces `{}`, or other characters specified with `commandchars`). See section 5.3 for details.

```
\begin{Verbatim}[breaklines, breakbefore=A]
some_string = 'SomeTextThatGoesOnAndOnForSoLongThatItCouldNeverFitOnOneLine'
\end{Verbatim}
```

---

```
some_string = 'SomeTextThatGoesOn⌟
↪  AndOnForSoLongThatItCouldNeverFitOnOneLine'
```

**breakbeforegroup** (boolean) (default: `true`)
When `breakbefore` is used, group all adjacent identical characters together, and only allow a break before the first character. When `breakbefore` and `breakafter` are used for the same character, `breakbeforegroup` and `breakaftergroup` must both have the same setting.

**breakbeforesymbolpre** (string) (default: `\,\footnotesize\ensuremath{_\rfloor}, ⌟`)
The symbol inserted pre-break for breaks inserted by `breakbefore`.

**breakbeforesymbolpost** (string) (default: ⟨*none*⟩)
The symbol inserted post-break for breaks inserted by `breakbefore`.

**breakindent** (dimension) (default: `0pt`)
When a line is broken, indent the continuation lines by this amount. When `breakautoindent` and `breakindent` are used together, the indentations add. This indentation is combined with `breaksymbolindentleft` to give the total actual left indentation.

**breaklines** (boolean) (default: `false`)
Automatically break long lines.

By default, automatic breaks occur at spaces. Use `breakanywhere` to enable breaking anywhere; use `breakbefore` and `breakafter` for more fine-tuned breaking.

```
...text.
\begin{Verbatim}[breaklines]
def f(x):
    return 'Some text ' + str(x)
\end{Verbatim}
```

...text.

```
def f(x):
    return 'Some text ' +
    ↪  str(x)
```

To customize the indentation of broken lines, see `breakindent` and `breakautoindent`. To customize the line continuation symbols, use `breaksymbolleft` and `breaksymbolright`. To customize the separation between the continuation symbols and the text, use `breaksymbolsepleft` and `breaksymbolsepright`. To customize the extra indentation that is supplied to make room for the break symbols, use

breaksymbolindentleft and breaksymbolindentright. Since only the left-hand symbol is used by default, it may also be modified using the alias options breaksymbol, breaksymbolsep, and breaksymbolindent.

An example using these options to customize the Verbatim environment is shown below. This uses the \carriagereturn symbol from the dingbat package.

```
\begin{Verbatim}[breaklines,
                 breakautoindent=false,
                 breaksymbolleft=\raisebox{0.8ex}{
                   \small\reflectbox{\carriagereturn}},
                 breaksymbolindentleft=0pt,
                 breaksymbolsepleft=0pt,
                 breaksymbolright=\small\carriagereturn,
                 breaksymbolindentright=0pt,
                 breaksymbolsepright=0pt]
def f(x):
    return 'Some text ' + str(x) + ' some more text ' +
     ↪  str(x) + ' even more text that goes on for a while'
\end{Verbatim}
```

```
def f(x):
    return 'Some text ' + str(x) + ' some more text ' +
↪ str(x) + ' even more text that goes on for a while'
```

Automatic line breaks will not work with showspaces=true unless you use breakanywhere, or use breakbefore or breakafter with \space. For example,

```
\begin{Verbatim}[breaklines, showspaces, breakafter=\space]
some_string = 'Some Text That Goes On And On For So Long That It Could Never Fit'
\end{Verbatim}
```

```
some_string␣=␣'Some␣Text␣That␣Goes␣On␣And␣On␣For␣So␣Long␣That␣
 ↪  It␣Could␣Never␣Fit'
```

**breaksymbol** (string) (default: breaksymbolleft)
Alias for breaksymbolleft.

**breaksymbolleft** (string) (default: \tiny\ensuremath{\hookrightarrow}, ↪)
The symbol used at the beginning (left) of continuation lines when breaklines=true. To have no symbol, simply set breaksymbolleft to an empty string ("=," or "={}"). The symbol is wrapped within curly braces {} when used, so there is no danger of formatting commands such as \tiny "escaping."

The \hookrightarrow and \hookleftarrow may be further customized by the use of the \rotatebox command provided by graphicx. Additional arrow-type symbols that may be useful are available in the dingbat (\carriagereturn) and mnsymbol (hook and curve arrows) packages, among others.

breaksymbolright (string) (default: ⟨*none*⟩)
The symbol used at breaks (right) when breaklines=true. Does not appear at the end of the very last segment of a broken line.

breaksymbolindent (dimension) (default: breaksymbolindentleft)
Alias for breaksymbolindentleft.

breaksymbolindentleft (dimension) (default: ⟨*width of 4 characters in teletype font at default point size*⟩)
The extra left indentation that is provided to make room for breaksymbolleft. This indentation is only applied when there is a breaksymbolleft.

This may be set to the width of a specific number of (fixed-width) characters by using an approach such as

```
\newdimen\temporarydimen
\settowidth{\temporarydimen}{\ttfamily aaaa}
```

and then using breaksymbolindentleft=\temporarydimen.

breaksymbolindentright (dimension) (default: ⟨*width of 4 characters in teletype font at default point size*⟩)
The extra right indentation that is provided to make room for breaksymbolright. This indentation is only applied when there is a breaksymbolright.

breaksymbolsep (dimension) (default: breaksymbolsepleft)
Alias for breaksymbolsepleft.

breaksymbolsepleft (dimension) (default: 1em)
The separation between the breaksymbolleft and the adjacent text.

breaksymbolsepright (dimension) (default: 1em)
The separation between the breaksymbolright and the adjacent text.

## 5.2 Line breaking and tab expansion

fancyvrb provides an obeytabs option that expands tabs based on tab stops rather than replacing them with a fixed number of spaces (see fancyvrb's tabsize). The fancyvrb implementation of tab expansion is not directly compatible with fvextra's line-breaking algorithm, but fvextra builds on the fancyvrb approach to obtain identical results.

Tab expansion in the context of line breaking does bring some additional considerations that should be kept in mind. In each line, all tabs are expanded exactly as they would have been had the line not been broken. This means that after a line break, any tabs will not align with tab stops unless the total left

indentation of continuation lines is a multiple of the tab stop width. The total indentation of continuation lines is the sum of `breakindent`, `breakautoindent`, and `breaksymbolindentleft` (alias `breaksymbolindent`).

A sample `Verbatim` environment that uses `obeytabs` with `breaklines` is shown below, with numbers beneath the environment indicating tab stops (`tabsize=8` by default). The tab stops in the wrapped and unwrapped lines are identical. However, the continuation line does not match up with the tab stops because by default the width of `breaksymbolindentleft` is equal to four monospace characters. (By default, `breakautoindent=true`, so the continuation line gets a tab plus `breaksymbolindentleft`.)

```
\begin{Verbatim}[obeytabs, showtabs, breaklines]
        ⇥First   ⇥Second ⇥Third   ⇥And more text that goes on for a
          ↪  while until wrapping is needed
        ⇥First   ⇥Second ⇥Third   ⇥Forth
\end{Verbatim}
12345678123456781234567812345678123456781234567812345678123456781234567812345678
```

We can set the symbol indentation to eight characters by creating a dimen,

```
\newdimen\temporarydimen
```

setting its width to eight characters,

```
\settowidth{\temporarydimen}{\ttfamily AaAaAaAa}
```

and finally adding the option `breaksymbolindentleft=\temporarydimen` to the `Verbatim` environment to obtain the following:

```
        ⇥First   ⇥Second ⇥Third   ⇥And more text that goes on for a
              ↪  while until wrapping is needed
        ⇥First   ⇥Second ⇥Third   ⇥Forth
12345678123456781234567812345678123456781234567812345678123456781234567812345678
```

## 5.3 Advanced line breaking

### 5.3.1 A few notes on algorithms

`breakanywhere`, `breakbefore`, and `breakafter` work by scanning through the tokens in each line and inserting line breaking commands wherever a break should be allowed. By default, they skip over all groups (`{...}`) and all math (`$...$`). Note that this refers to curly braces and dollar signs with their normal LaTeX meaning (catcodes), not verbatim curly braces and dollar signs; such non-verbatim content may be enabled with `commandchars` or `codes`. This means that math and macros

that only take mandatory arguments (`{...}`) will function normally within otherwise verbatim text. However, macros that take optional arguments may not work because `[...]` is not treated specially, and thus break commands may be inserted within `[...]` depending on settings. Wrapping an entire macro, including its arguments, in a group will protect the optional argument: `{\`⟨*macro*⟩`[`⟨*oarg*⟩`]{`⟨*marg*⟩`}}`.

`breakbefore` and `breakafter` insert line breaking commands around specified characters. This process is catcode-independent; tokens are `\detokenized` before they are checked against characters specified via `breakbefore` and `breakafter`.

### 5.3.2 Breaks within macro arguments

`\FancyVerbBreakStart`
`\FancyVerbBreakStop`

When `commandchars` or `codes` are used to include macros within verbatim content, the options `breakanywhere`, `breakbefore`, and `breakafter` will not generate breaks within mandatory macro arguments. Macros with optional arguments may not work, depending on settings, unless they are wrapped in a group (curly braces `{}`, or other characters specified via `commandchars`).

If you want to allow breaks within macro arguments (optional or mandatory), then you should (re)define your macros so that the relevant arguments are wrapped in the commands

```
\FancyVerbBreakStart ... \FancyVerbBreakStop
```

For example, suppose you have the macro

```
\newcommand{\mycmd}[1]{\_before:#1:after\_}
```

Then you would discover that line breaking does not occur:

```
\begin{Verbatim}[commandchars=\\\{\}, breaklines, breakafter=a]
\mycmd{1}\mycmd{2}\mycmd{3}\mycmd{4}\mycmd{5}
\end{Verbatim}
```

---

```
_before:1:after__before:2:after__before:3:after__before:4:after__before:5:after_
```

Now redefine the macro:

```
\renewcommand{\mycmd}[1]{\FancyVerbBreakStart\_before:#1:after\_\FancyVerbBreakStop}
```

This is the result:

```
\begin{Verbatim}[commandchars=\\\{\}, breaklines, breakafter=a]
\mycmd{1}\mycmd{2}\mycmd{3}\mycmd{4}\mycmd{5}
\end{Verbatim}
```

---

```
_before:1:after__before:2:after__before:3:after__before:4:a
↪  fter__before:5:after_
```

Instead of completely redefining macros, it may be more convenient to use `\let`. For example,

```
\let\originalmycmd\mycmd
\renewcommand{\mycmd}[1]{%
  \expandafter\FancyVerbBreakStart\originalmycmd{#1}\FancyVerbBreakStop}
```

Notice that in this case `\expandafter` is required, because `\FancyVerbBreakStart` does not perform any expansion and thus will skip over `\originalmycmd{#1}` unless it is already expanded. The etoolbox package provides commands that may be useful for patching macros to insert line breaks.

When working with `\FancyVerbBreakStart ... \FancyVerbBreakStop`, keep in mind that any groups `{...}` or math `$...$` between the two commands will be skipped as far as line breaks are concerned, and breaks may be inserted within any optional arguments `[...]` depending on settings. Inserting breaks within groups requires another level of `\FancyVerbBreakStart` and `\FancyVerbBreakStop`, and protecting optional arguments requires wrapping the entire macro in a group `{...}`. Also, keep in mind that `\FancyVerbBreakStart` cannot introduce line breaks in a context in which they are never allowed, such as in an `\hbox`.

### 5.3.3 Customizing break behavior

`\FancyVerbBreakAnywhereBreak`

`\FancyVerbBreakBeforeBreak`

`\FancyVerbBreakAfterBreak`

These macros govern the behavior of breaks introduced by `breakanywhere`, `breakbefore`, and `breakafter`. Breaks introduced by the default `breaklines` when `showspaces=false` are standard breaks following spaces. No special commands are provided for working with them; the normal LaTeX commands for breaking should suffice.

By default, these macros use `\discretionary`. `\discretionary` takes three arguments: commands to insert before the break, commands to insert after the break, and commands to insert if there is no break. For example, the default definition of `\FancyVerbBreakAnywhereBreak`:

```
\newcommand{\FancyVerbBreakAnywhereBreak}{%
  \discretionary{\FancyVerbBreakAnywhereSymbolPre}%
   {\FancyVerbBreakAnywhereSymbolPost}{}}
```

The other macros are equivalent, except that "`Anywhere`" is swapped for "`Before`" or "`After`".

`\discretionary` will generally only insert breaks when breaking at spaces simply cannot make lines short enough (this may be tweaked to some extent with hyphenation settings). This can produce a somewhat ragged appearance in some cases. If you want breaks exactly at the margin (or as close as possible) regardless of whether a break at a space is an option, you may want to use `\allowbreak` instead. Another option is `\linebreak[`$\langle n \rangle$`]`, where $\langle n \rangle$ is between 0 to 4, with 0 allowing a break and 4 forcing a break.

# 6 Patches

fvextra modifies some fancyvrb behavior that is the result of bugs or omissions.

## 6.1 Visible spaces

The command `\FancyVerbSpace` defines the visible space when `showspaces=true`. The default fancyvrb definition allows a font command to escape under some circumstances, so that all following text is forced to be teletype font. The command is redefined to use `\textvisiblespace`.

## 6.2 `obeytabs` with visible tabs and with tabs inside macro arguments

The original fancyvrb treatment of visible tabs when `showtabs=true` and `obeytabs=true` did not allow variable-width tab symbols such as `\rightarrowfill` to function correctly. This is fixed through a redefinition of `\FV@TrueTab`.

Various macros associated with `obeytabs=true` are also redefined so that tabs may be expanded regardless of whether they are within a group (within `{...}` with the normal LaTeX meaning due to `commandchars`, etc.). In the fancyvrb implementation, using `obeytabs=true` when a tab is inside a group typically causes the entire line to vanish. fvextra patches this so that the tab is expanded and will be visible if `showtabs=true`. Note, though, that the tab expansion in these cases is only guaranteed to be correct for leading whitespace that is inside a group. The start of each run of whitespace that is inside a group is treated as a tab stop, whether or not it actually is, due to limitations of the tab expansion algorithm. A more detailed discussion is provided in the implementation.

The example below shows correct tab expansion of leading whitespace within a macro argument. With fancyvrb, the line of text would simply vanish in this case.

```
\begin{Verbatim}[obeytabs, showtabs, showspaces, tabsize=4,
   commandchars=\\\{\}, tab=\textcolor{orange}{\rightarrowfill}]
\textcolor{blue}{        ⤔       ⤔Text after 1 space + 2 tabs}
\end{Verbatim}
```

```
␣⟶⟶Text␣after␣1␣space␣+␣2␣tabs
```

The next example shows that tab expansion inside macros in the midst of text typically does not match up with the correct tab stops, since in such circumstances the beginning of the run of whitespace must be treated as a tab stop.

```
\begin{Verbatim}[obeytabs, showtabs, commandchars=\\\{\},
                 tab=\textcolor{orange}{\rightarrowfill}]
\textcolor{blue}{      ⇥       ⇥2 leading tabs}
\textcolor{blue}{Text  ⇥       ⇥then 2 tabs}
\end{Verbatim}
```

```
————————→————————→2 leading tabs
Text————————→————————→then 2 tabs
```

## 6.3 Math mode

### 6.3.1 Spaces

When typeset math is included within verbatim material, fancyvrb makes spaces within the math appear literally.

```
\begin{Verbatim}[commandchars=\\\{\}, mathescape]
Verbatim $\displaystyle\frac{1}{  x^2    +    y^2  }$ verbatim
\end{Verbatim}
```

---

Verbatim $\displaystyle\frac{1}{\quad x^2 \quad + \quad y^2 \quad}$ verbatim

fvextra patches this by redefining fancyvrb's space character within math mode so that it behaves as expected:

Verbatim $\displaystyle\frac{1}{x^2 + y^2}$ verbatim

### 6.3.2 Symbols and fonts

With fancyvrb, using a single quotation mark (`'`) in typeset math within verbatim material results in an error rather than a prime symbol ($'$).[5] fvextra redefines the behavior of the single quotation mark within math mode to fix this, so that it will become a proper prime.

The amsmath package provides a `\text` command for including normal text within math. With fancyvrb, `\text` does not behave normally when used in typeset math within verbatim material. fvextra redefines the backtick (`` ` ``) and the single quotation mark so that they function normally within `\text`, becoming left and

[5] The single quotation mark is made active within verbatim material to prevent ligatures, via `\@noligs`. The default definition is incompatible with math mode.

right quotation marks. It redefines the greater-than sign, less-than sign, comma, and hyphen so that they function normally as well. fvextra also switches back to the default document font within `\text`, rather than using the verbatim font, which is typically a monospace or typewriter font.

The result of these modifications is a math mode that very closely mimics the behavior of normal math mode outside of verbatim material.

```
\begin{Verbatim}[commandchars=\\\{\}, mathescape]
Verbatim $\displaystyle f'''(x) = \text{``Some quoted text---''}$
\end{Verbatim}
```

---

`Verbatim` $f'''(x) =$ “Some quoted text—”

### 6.4 Orphaned labels

When `frame=lines` is used with a `label`, fancyvrb does not prevent the label from being orphaned under some circumstances. `\FV@BeginListFrame@Lines` is patched to prevent this.

### 6.5 `rulecolor` and `fillcolor`

The `rulecolor` and `fillcolor` options are redefined so that they accept color names directly, rather than requiring `\color{`⟨*color_name*⟩`}`. The definitions still allow the old usage.

## 7 Additional modifications to fancyvrb

fvextra modifies some fancyvrb behavior with the intention of improving logical consistency or providing better defaults.

### 7.1 Backtick and single quotation mark

With fancyvrb, the backtick `` ` `` and typewriter single quotation mark `'` are typeset as the left and right curly single quotation marks `‘’`. fvextra loads the upquote package so that these characters will appear literally by default. The original fancyvrb behavior can be restored with the fvextra option `curlyquotes` (section 3).

### 7.2 Line numbering

With fancyvrb, using `firstnumber` to offset line numbering in conjunction with `stepnumber` changes which line numbers appear. Lines are numbered if their original line numbers, without the `firstnumber` offset, are a multiple of `stepnumber`. But the actual numbers that appear are the offset values that include `firstnumber`.

Thus, using `firstnumber=2` with `stepnumber=5` would cause the original lines $5, 10, 15, \ldots$ to be numbered, but with the values $6, 11, 16, \ldots$.

fvextra changes line numbering so that when `stepnumber` is used, the actual line numbers that appear are always multiples of `stepnumber` by default, regardless of any `firstnumber` offset. The original fancyvrb behavior may be turned on by setting `stepnumberoffsetvalues=true` (section 3).

# 8 Undocumented features of fancyvrb

fancyvrb defines some potentially useful but undocumented features.

## 8.1 Undocumented options

`codes*` (macro) (default: ⟨*empty*⟩)
fancyvrb's `codes` is used to specify catcode changes. It overwrites any existing `codes`. `codes*` appends changes to existing settings.

`defineactive*` (macro) (default: ⟨*empty*⟩)
fancyvrb's `defineactive` is used to define the effect of active characters. It overwrites any existing `defineactive`. `defineactive*` appends changes to existing settings.

`formatcom*` (macro) (default: ⟨*empty*⟩)
fancyvrb's `formatcom` is used to execute commands before verbatim text. It overwrites any existing `formatcom`. `formatcom*` appends changes to existing settings.

## 8.2 Undocumented macros

`\FancyVerbTab`

This defines the visible tab character (⇥) that is used when `showtabs=true`. The default definition is

```
\def\FancyVerbTab{%
  \valign{%
    \vfil##\vfil\cr
    \hbox{$\scriptscriptstyle-$}\cr
    \hbox to 0pt{\hss$\scriptscriptstyle\rangle\mskip -.8mu$}\cr
    \hbox{$\scriptstyle\mskip -3mu\mid\mskip -1.4mu$}\cr}}
```

While this may be redefined directly, fvextra also defines a new option `tab`

`\FancyVerbSpace`

This defines the visible space character (␣) that is used when `showspaces=true`. The default definition (as patched by fvextra, section 6.1) is `\textvisiblespace`. While this may be redefined directly, fvextra also defines a new option `space`.

## Version History

**v1.1** (2016/07/14)

- The options `rulecolor` and `fillcolor` now accept color names directly; using `\color{<color_name>}` is no longer necessary, though it still works.
- Added `tabcolor` and `spacecolor` options for use with `showtabs` and `showspaces`.
- Added `highlightlines` option that takes a line number or range of line numbers and highlights the corresponding lines. Added `highlightcolor` option that controls hightlighting color.
- `obeytabs` no longer causes lines to vanish when tabs are inside macro arguments. Tabs and spaces inside a macro argument but otherwise at the beginning of a line are expanded correctly. Tabs inside a macro argument that are preceded by non-whitespace characters (not spaces or tabs) are expanded based on the starting position of the run of whitespace in which they occur.
- The line breaking options `breakanywhere`, `breakbefore`, and `breakafter` now work with multi-byte UTF-8 code points under pdfTeX with `inputenc`. They were already fully functional under XeTeX and LuaTeX.
- Added `curlyquotes` option, which essentially disables the `uquote` package.

**v1.0** (2016/06/28)

- Initial release.

# 9 Implementation

## 9.1 Required packages

The upquote package performs some font checks when it is loaded to determine whether textcomp is needed, but errors can result if the font is changed later in the preamble, so duplicate the package's font check at the end of the preamble. Also check for a package order issue with lineno and csquotes.

```
\RequirePackage{ifthen}
\RequirePackage{etoolbox}
\RequirePackage{fancyvrb}
\RequirePackage{upquote}
\AtEndPreamble{%
  \ifx\encodingdefault\upquote@OTone
    \ifx\ttdefault\upquote@cmtt\else\RequirePackage{textcomp}\fi
  \else
    \RequirePackage{textcomp}
```

```
  \fi}
\RequirePackage{lineno}
\@ifpackageloaded{csquotes}%
 {\PackageWarning{fvextra}{csquotes should be loaded after fvextra, %
  to avoid a warning from the lineno package}}{}
```

## 9.2 Utility macros

\FV@Space@ifx Macro for testing if a `\let` token is `\FV@Space` with `\ifx`. The space will be active and defined as `\FV@Space`.

```
\def\FV@Space@ifx{\FV@Space}
```

\FV@Tab@ifx Macro for testing if a `\let` token is `\FV@Tab` with `\ifx`. The tab will be active and defined as `\FV@Tab`.

```
\def\FV@Tab@ifx{\FV@Tab}
```

## 9.3 Hooks

\FV@FormattingPrepHook This is a hook for extending `\FV@FormattingPrep`. `\FV@FormattingPrep` is inside a group, before the beginning of processing, so it is a good place to add extension code. This hook is used for such things as tweaking math mode behavior and preparing for `breakbefore` and `breakafter`.

```
\let\FV@FormattingPrepHook\@empty
\expandafter\def\expandafter\FV@FormattingPrep\expandafter{%
  \expandafter\FV@FormattingPrepHook\FV@FormattingPrep}
```

## 9.4 Escaped characters

\FV@EscChars Define versions of common escaped characters that reduce to raw characters. This is useful, for example, when working with text that is almost verbatim, but was captured in such a way that some escapes were unavoidable.

```
\edef\FV@hashchar{\string#}
\edef\FV@dollarchar{\string$}
\edef\FV@ampchar{\string&}
\edef\FV@underscorechar{\string_}
\edef\FV@tildechar{\string~}
\edef\FV@leftsquarebracket{\string[}
\edef\FV@rightsquarebracket{\string]}
\newcommand{\FV@EscChars}{%
  \let\#\FV@hashchar
  \let\%\@percentchar
  \let\{\@charlb
  \let\}\@charrb
  \let\$\FV@dollarchar
  \let\&\FV@ampchar
  \let\_\FV@underscorechar
  \let\\\@backslashchar
  \let~\FV@tildechar
```

```
  \let\~\FV@tildechar
  \let\[\FV@leftsquarebracket
  \let\]\FV@rightsquarebracket
} %$ <- highlighting
```

## 9.5 Patches

### 9.5.1 Visible spaces

\FancyVerbSpace The default definition of visible spaces (`showspaces=true`) could allow font commands to escape under some circumstances, depending on how it is used:

```
{\catcode`\ =12 \gdef\FancyVerbSpace{\tt }}
```

The command is redefined in more robust and standard LaTeX form.

```
\def\FancyVerbSpace{\textvisiblespace}
```

### 9.5.2 `obeytabs` with visible tabs and with tabs inside macro arguments

`\FV@TrueTab` governs tab appearance when `obeytabs=true` and `showtabs=true`. It is redefined so that symbols with flexible width, such as `\rightarrowfill`, will work as expected. In the original fancyvrb definition, `\kern\@tempdima\hbox to\z@{...}`. The `\kern` is removed and instead the `\hbox` is given the width `\@tempdima`.

`\FV@TrueTab` and related macros are also modified so that they function for tabs inside macro arguments when `obeytabs=true` (inside curly braces `{}` with their normal meaning, when using `commandchars`, etc.). The fancyvrb implementation of tab expansion assumes that tabs are never inside a group; when a group that contains a tab is present, the entire line typically vanishes. The new implementation keeps the fancyvrb behavior exactly for tabs outside groups; they are perfectly expanded to tab stops. Tabs inside groups cannot be perfectly expanded to tab stops, at least not using the fancyvrb approach. Instead, when fvextra encounters a run of whitespace characters (tabs and possibly spaces), it makes the assumption that the nearest tab stop was at the beginning of the run. This gives the correct behavior if the whitespace characters are leading indentation that happens to be within a macro. Otherwise, it will typically not give correct tab expansion—but at least the entire line will not be discarded, and the run of whitespace will be represented, even if imperfectly.

A general solution to tab expansion may be possible, but will almost certainly require multiple compiles, perhaps even one compile (or more) per tab. The zref package provides a `\zsaveposx` macro that stores the current $x$ position on the page for subsequent compiles. This macro, or a similar macro from another package, could be used to establish a reference point at the beginning of each line. Then each run of whitespace that contains a tab could have a reference point established at its start, and tabs could be expanded based on the distance between the start of the run and the start of the line. Such an approach would allow the first run of whitespace to measure its distance from the start of the line on the 2nd compile (once both reference points were established), so it would be able expand the first run of whitespace correctly on the 3rd compile. That would allow a second run of

whitespace to definitely establish its starting point on the 3rd compile, which would allow it to expand correctly on the 4th compile. And so on. Thus, while it should be possible to perform completely correct tab expansion with such an approach, it will in general require at least 4 compiles to do better than the current approach. Furthermore, the sketch of the algorithm provided so far does not include any complications introduced by line breaking. In the current approach, it is necessary to determine how each tab would be expanded in the absence of line breaking, save all tab widths, and then expand using saved widths during the actual typesetting with line breaking.

`FV@TrueTabGroupLevel` Counter for keeping track of the group level (`\currentgrouplevel`) at the very beginning of a line, inside `\FancyVerbFormatLine` but outside `\FancyVerbFormatText`, which is where the tab expansion macro is invoked. This allows us to determine whether we are in a group, and expand tabs accordingly.

```
\newcounter{FV@TrueTabGroupLevel}
```

`\FV@@ObeyTabs` The fancyvrb macro responsible for tab expansion is modified so that it can handle tabs inside groups, even if imperfectly. We need to use a special version of the space, `\FV@Space@ObeyTabs`, that within a group will capture all following spaces or tabs and then insert them with tab expansion based on the beginning of the run of whitespace. We need to record the current group level, but then increment it by 1 because all comparisons will be performed within the `\hbox{...}`.

```
\def\FV@@ObeyTabs#1{%
  \let\FV@Space@Orig\FV@Space
  \let\FV@Space\FV@Space@ObeyTabs
  \setcounter{FV@TrueTabGroupLevel}{\the\currentgrouplevel}%
  \addtocounter{FV@TrueTabGroupLevel}{1}%
  \setbox\FV@TabBox=\hbox{#1}\box\FV@TabBox
  \let\FV@Space\FV@Space@Orig}
```

`\FV@TrueTab` Version that follows fancyvrb if not in a group and takes another approach otherwise.

```
\def\FV@TrueTab{%
  \ifnum\value{FV@TrueTabGroupLevel}=\the\currentgrouplevel\relax
    \expandafter\FV@TrueTab@NoGroup
  \else
    \expandafter\FV@TrueTab@Group
  \fi}
```

`\FV@TrueTabSaveWidth` When linebreaking is in use, the fancyvrb tab expansion algorithm cannot be used directly, since it involves `\hbox`, which doesn't allow for line breaks. In those cases, tab widths will be calculated for the case without breaks and saved, and then saved widths will be used in the actual typesetting. This macro is `\let` to width-saving code in those cases.

```
\let\FV@TrueTabSaveWidth\relax
```

`FV@TrueTabCounter` Counter for tracking saved tabs.

```
\newcounter{FV@TrueTabCounter}
```

\FV@TrueTabSaveWidth@Save

Save the current tab width, then increment the tab counter. \@tempdima will hold the current tab width.

```
\def\FV@TrueTabSaveWidth@Save{%
  \expandafter\xdef\csname FV@TrueTab:Width\arabic{FV@TrueTabCounter}\endcsname{%
    \number\@tempdima}%
  \stepcounter{FV@TrueTabCounter}}
```

\FV@TrueTab@NoGroup

This follows the fancyvrb approach exactly, except for the \hbox to\@tempdima adjustment and the addition of \FV@TrueTabSaveWidth.

```
\def\FV@TrueTab@NoGroup{%
  \egroup
  \@tempdima=\FV@ObeyTabSize sp\relax
  \@tempcnta=\wd\FV@TabBox
  \advance\@tempcnta\FV@@ObeyTabSize\relax
  \divide\@tempcnta\@tempdima
  \multiply\@tempdima\@tempcnta
  \advance\@tempdima-\wd\FV@TabBox
  \FV@TrueTabSaveWidth
  \setbox\FV@TabBox=\hbox\bgroup
    \unhbox\FV@TabBox\hbox to\@tempdima{\hss\FV@TabChar}}
```

FV@ObeyTabs@Whitespace@Tab

In a group where runs of whitespace characters are collected, we need to keep track of whether a tab has been found, so we can avoid expansion and the associated \hbox for spaces without tabs.

```
\newboolean{FV@ObeyTabs@Whitespace@Tab}
```

\FV@TrueTab@Group

If in a group, a tab should start collecting whitespace characters for later tab expansion, beginning with itself. The collected whitespace will use \FV@Tab@ifx and \FV@Space@ifx so that any \ifx comparisons performed later will behave as expected. This shouldn't be strictly necessary, because \FancyVerbBreakStart operates with saved tab widths rather than using the tab expansion code directly. But it is safer in case any other unanticipated scanning is going on.

```
\def\FV@TrueTab@Group{%
  \booltrue{FV@ObeyTabs@Whitespace@Tab}%
  \gdef\FV@TmpWhitespace{\FV@Tab@ifx}%
  \FV@ObeyTabs@ScanWhitespace}
```

\FV@Space@ObeyTabs

Space treatment, like tab treatment, now depends on whether we are in a group, because in a group we want to collect all runs of whitespace and then expand any tabs.

```
\def\FV@Space@ObeyTabs{%
  \ifnum\value{FV@TrueTabGroupLevel}=\the\currentgrouplevel\relax
    \expandafter\FV@Space@ObeyTabs@NoGroup
  \else
    \expandafter\FV@Space@ObeyTabs@Group
  \fi}
```

\FV@Space@ObeyTabs@NoGroup

Fall back to normal space.

```
\def\FV@Space@ObeyTabs@NoGroup{\FV@Space@Orig}
```

`\FV@Space@ObeyTabs@Group` Make a note that no tabs have yet been encountered, store the current space, then scan for following whitespace.

```
\def\FV@Space@ObeyTabs@Group{%
  \boolfalse{FV@ObeyTabs@Whitespace@Tab}%
  \gdef\FV@TmpWhitespace{\FV@Space@ifx}%
  \FV@ObeyTabs@ScanWhitespace}
```

`\FV@ObeyTabs@ScanWhitespace` Collect whitespace until the end of the run, then process it. Proper lookahead comparison requires `\FV@Space@ifx` and `\FV@Tab@ifx`.

```
\def\FV@ObeyTabs@ScanWhitespace{%
  \@ifnextchar\FV@Space@ifx%
   {\FV@TrueTab@CaptureWhitespace@Space}%
   {\ifx\@let@token\FV@Tab@ifx
      \expandafter\FV@TrueTab@CaptureWhitespace@Tab
    \else
      \expandafter\FV@ObeyTabs@ResolveWhitespace
    \fi}}
\def\FV@TrueTab@CaptureWhitespace@Space#1{%
  \g@addto@macro\FV@TmpWhitespace{\FV@Space@ifx}%
  \FV@ObeyTabs@ScanWhitespace}
\def\FV@TrueTab@CaptureWhitespace@Tab#1{%
  \booltrue{FV@ObeyTabs@Whitespace@Tab}%
  \g@addto@macro\FV@TmpWhitespace{\FV@Tab@ifx}%
  \FV@ObeyTabs@ScanWhitespace}
```

`\FV@TrueTab@Group@Expand` Yet another tab definition, this one for use in the actual expansion of tabs in whitespace. This uses the fancyvrb algorithm, but only over a restricted region known to contain no groups.

```
\newbox\FV@TabBox@Group
\def\FV@TrueTab@Group@Expand{%
  \egroup
  \@tempdima=\FV@ObeyTabSize sp\relax
  \@tempcnta=\wd\FV@TabBox@Group
  \advance\@tempcnta\FV@@ObeyTabSize\relax
  \divide\@tempcnta\@tempdima
  \multiply\@tempdima\@tempcnta
  \advance\@tempdima-\wd\FV@TabBox@Group
  \FV@TrueTabSaveWidth
  \setbox\FV@TabBox@Group=\hbox\bgroup
    \unhbox\FV@TabBox@Group\hbox to\@tempdima{\hss\FV@TabChar}}
```

`\FV@ObeyTabs@ResolveWhitespace` Need to make sure the right definitions of the space and tab are in play here. Only do tab expansion, with the associated `\hbox`, if a tab is indeed present.

```
\def\FV@ObeyTabs@ResolveWhitespace{%
  \let\FV@Space\FV@Space@Orig
  \let\FV@Tab\FV@TrueTab@Group@Expand
  \expandafter\FV@ObeyTabs@ResolveWhitespace@i\expandafter{\FV@TmpWhitespace}%
  \let\FV@Space\FV@Space@ObeyTabs
  \let\FV@Tab\FV@TrueTab}
```

```
\def\FV@ObeyTabs@ResolveWhitespace@i#1{%
  \ifbool{FV@ObeyTabs@Whitespace@Tab}%
   {\setbox\FV@TabBox@Group=\hbox{#1}\box\FV@TabBox@Group}%
   {#1}}
```

### 9.5.3 Spacing in math mode

\FancyVerbMathSpace

\FV@Space is defined as either a non-breaking space or a visible representation of a space, depending on the option showspaces. Neither option is desirable when typeset math is included within verbatim content, because spaces will not be discarded as in normal math mode. Define a space for math mode.

```
\def\FancyVerbMathSpace{ }
```

\FV@SetupMathSpace

Define a macro that will activate math spaces, then add it to an fvextra hook.

```
\def\FV@SetupMathSpace{%
  \everymath\expandafter{\the\everymath\let\FV@Space\FancyVerbMathSpace}}
\g@addto@macro\FV@FormattingPrepHook{\FV@SetupMathSpace}
```

### 9.5.4 Fonts and symbols in math mode

The single quote (') does not become ^\prime when typeset math is included within verbatim content, due to the definition of the character in \@noligs. This patch adds a new definition of the character in math mode, inspired by http://tex.stackexchange.com/q/223876/10742. It also redefines other characters in \@noligs to behave normally within math mode and switches the default font within math mode, so that amsmath's \text will work as expected.

\FV@pr@m@s

Define a version of \pr@m@s from latex.ltx that works with active '. In verbatim contexts, ' is made active by \@noligs.

```
\begingroup
\catcode`\'=\active
\catcode`\^=7
\gdef\FV@pr@m@s{%
  \ifx'\@let@token
    \expandafter\pr@@@s
  \else
    \ifx^\@let@token
      \expandafter\expandafter\expandafter\pr@@@t
    \else
      \egroup
    \fi
  \fi}
\endgroup
```

\FV@SetupMathFont

Set the font back to default from the verbatim font.

```
\def\FV@SetupMathFont{%
  \everymath\expandafter{\the\everymath\fontfamily{\familydefault}\selectfont}}
\g@addto@macro\FV@FormattingPrepHook{\FV@SetupMathFont}
```

\FV@SetupMathLigs Make all characters in \@noligs behave normally, and switch to \FV@pr@m@s. The relevant definition from latex.ltx:

```
\def\verbatim@nolig@list{\do\`\do\<\do\>\do\,\do\'\do\-}
```

```
\def\FV@SetupMathLigs{%
  \everymath\expandafter{%
    \the\everymath
    \let\pr@m@s\FV@pr@m@s
    \begingroup\lccode`\~=`\'\lowercase{\endgroup\def~}{%
      \ifmmode\expandafter\active@math@prime\else'\fi}%
    \begingroup\lccode`\~=`\`\lowercase{\endgroup\def~}{`}%
    \begingroup\lccode`\~=`\<\lowercase{\endgroup\def~}{<}%
    \begingroup\lccode`\~=`\>\lowercase{\endgroup\def~}{>}%
    \begingroup\lccode`\~=`\,\lowercase{\endgroup\def~}{,}%
    \begingroup\lccode`\~=`\-\lowercase{\endgroup\def~}{-}%
  }%
}
\g@addto@macro\FV@FormattingPrepHook{\FV@SetupMathLigs}
```

### 9.5.5 Ophaned label

\FV@BeginListFrame@Lines When frame=lines is used with a label, the label can be orphaned. This overwrites the default definition to add \penalty\@M. The fix is attributed to http://tex.stackexchange.com/a/168021/10742.

```
\def\FV@BeginListFrame@Lines{%
  \begingroup
  \lineskip\z@skip
  \FV@SingleFrameLine{\z@}%
  \kern-0.5\baselineskip\relax
  \baselineskip\z@skip
  \kern\FV@FrameSep\relax
  \penalty\@M
  \endgroup}
```

### 9.5.6 rulecolor and fillcolor

The rulecolor and fillcolor options are redefined so that they accept color names directly, rather than requiring \color{⟨*color_name*⟩}. The definitions still allow the old usage.

rulecolor

```
\define@key{FV}{rulecolor}{%
  \ifstrempty{#1}%
   {\let\FancyVerbRuleColor\relax}%
   {\ifstrequal{#1}{none}%
     {\let\FancyVerbRuleColor\relax}%
     {\def\@tempa{#1}%
      \FV@KVProcess@RuleColor#1\FV@Undefined}}}
\def\FV@KVProcess@RuleColor#1#2\FV@Undefined{%
```

```
  \ifx#1\color
  \else
    \expandafter\def\expandafter\@tempa\expandafter{%
      \expandafter\color\expandafter{\@tempa}}%
  \fi
  \let\FancyVerbRuleColor\@tempa}
\fvset{rulecolor=none}
```

fillcolor

```
\define@key{FV}{fillcolor}{%
  \ifstrempty{#1}%
   {\let\FancyVerbFillColor\relax}%
   {\ifstrequal{#1}{none}%
     {\let\FancyVerbFillColor\relax}%
     {\def\@tempa{#1}%
      \FV@KVProcess@FillColor#1\FV@Undefined}}}
\def\FV@KVProcess@FillColor#1#2\FV@Undefined{%
  \ifx#1\color
  \else
    \expandafter\def\expandafter\@tempa\expandafter{%
      \expandafter\color\expandafter{\@tempa}}%
  \fi
  \let\FancyVerbFillColor\@tempa}
\fvset{fillcolor=none}
```

## 9.6 Extensions

### 9.6.1 New options requiring minimal implementation

linenos

fancyvrb allows line numbers via the options `numbers=left` and `numbers=right`. This creates a `linenos` key that is essentially an alias for `numbers=left`.

```
\define@booleankey{FV}{linenos}%
  {\@nameuse{FV@Numbers@left}}{\@nameuse{FV@Numbers@none}}
```

tab

Redefine `\FancyVerbTab`.

```
\define@key{FV}{tab}{\def\FancyVerbTab{#1}}
```

tabcolor

Set tab color, or allow it to adjust to surroundings (the default fancyvrb behavior). This involves re-creating the `showtabs` option to add `\FV@TabColor`.

```
\define@key{FV}{tabcolor}%
 {\ifstrempty{#1}%
   {\let\FV@TabColor\relax}%
   {\ifstrequal{#1}{none}%
     {\let\FV@TabColor\relax}%
     {\def\FV@TabColor{\textcolor{#1}}}}}
\define@booleankey{FV}{showtabs}%
 {\def\FV@TabChar{\FV@TabColor{\FancyVerbTab}}}%
 {\let\FV@TabChar\relax}
\fvset{tabcolor=none, showtabs=false}
```

space Redefine \FancyVerbSpace.

```
\define@key{FV}{space}{\def\FancyVerbSpace{#1}}
```

spacecolor Set space color, or allow it to adjust to surroundings (the default fancyvrb behavior). This involves re-creating the showspaces option to add \FV@SpaceColor.

```
\define@key{FV}{spacecolor}%
 {\ifstrempty{#1}%
   {\let\FV@SpaceColor\relax}%
   {\ifstrequal{#1}{none}%
     {\let\FV@SpaceColor\relax}%
     {\def\FV@SpaceColor{\textcolor{#1}}}}}
\define@booleankey{FV}{showspaces}%
 {\def\FV@Space{\FV@SpaceColor{\FancyVerbSpace}}}%
 {\def\FV@Space{\ }}
\fvset{spacecolor=none, showspaces=false}
```

mathescape Give $, ^, and _ their normal catcodes to allow normal typeset math.

```
\define@booleankey{FV}{mathescape}%
 {\let\FancyVerbMathEscape\FV@MathEscape}%
 {\let\FancyVerbMathEscape\relax}
\def\FV@MathEscape{\catcode`\$=3\catcode`\^=7\catcode`\_=8\relax}
\FV@AddToHook\FV@CatCodesHook\FancyVerbMathEscape
\fvset{mathescape=false}
```

curlyquotes Let ` and ' produce curly quotation marks ‘ and ’ rather than the backtick and typewriter single quotation mark produced by default via upquote.

```
\newbool{FV@CurlyQuotes}
\define@booleankey{FV}{curlyquotes}%
 {\booltrue{FV@CurlyQuotes}}%
 {\boolfalse{FV@CurlyQuotes}}
\def\FancyVerbCurlyQuotes{%
  \ifbool{FV@CurlyQuotes}%
   {\expandafter\def\expandafter\@noligs\expandafter{\@noligs
      \begingroup\lccode`\~=`\`\lowercase{\endgroup\def~}{`}%
      \begingroup\lccode`\~=`\'\lowercase{\endgroup\def~}{'}}}%
   {}}
\g@addto@macro\FV@FormattingPrepHook{\FancyVerbCurlyQuotes}
\fvset{curlyquotes=false}
```

### 9.6.2 Formatting with \FancyVerbFormatLine, \FancyVerbFormatText, and \FancyVerbHighlightLine

fancyvrb defines \FancyVerbFormatLine, which defines the formatting for each line. The introduction of line breaks introduces an issue for \FancyVerbFormatLine. Does it format the entire line, including any whitespace in the margins or behind line break symbols (that is, is it outside the \parbox in which the entire line is wrapped when breaking is active)? Or does it only format the text part of the line, only affecting the actual characters (inside the \parbox)? Since both might be

desirable, \FancyVerbFormatLine is assigned to the entire line, and a new macro \FancyVerbFormatText is assigned to the text, within the \parbox.

An additional complication is that the fancyvrb documentation says that the default value is \def\FancyVerbFormatLine#1{#1}. But the actual default is \def\FancyVerbFormatLine#1{\FV@ObeyTabs{#1}}. That is, \FV@ObeyTabs needs to operate directly on the line to handle tabs. As a result, *all* fancyvrb commands that involve \FancyVerbFormatLine are patched, so that \def\FancyVerbFormatLine#1{#1}.

An additional macro \FancyVerbHighlightLine is added between \FancyVerbFormatLine and \FancyVerbFormatText. This is used to highlight selected lines (section 9.6.4). It is inside \FancyVerbHighlightLine so that if \FancyVerbHighlightLine is used to provide a background color, \FancyVerbHighlightLine can override it.

\FancyVerbFormatLine Format the entire line, following the definition given in the fancyvrb documentation. Because this is formatting the entire line, using boxes works with line breaking.

```
\def\FancyVerbFormatLine#1{#1}
```

\FancyVerbFormatText Format only the text part of the line. Because this is inside all of the line breaking commands, using boxes here can conflict with line breaking.

```
\def\FancyVerbFormatText#1{#1}
```

\FV@ListProcessLine@NoBreak Redefined \FV@ListProcessLine in which \FancyVerbFormatText is added and tab handling is explicit. The @NoBreak suffix is added because \FV@ListProcessLine will be \let to either this macro or to \FV@ListProcessLine@Break depending on whether line breaking is enabled.

```
\def\FV@ListProcessLine@NoBreak#1{%
  \hbox to \hsize{%
    \kern\leftmargin
    \hbox to \linewidth{%
      \FV@LeftListNumber
      \FV@LeftListFrame
      \FancyVerbFormatLine{%
        \FancyVerbHighlightLine{%
          \FV@ObeyTabs{\FancyVerbFormatText{#1}}}}\hss
      \FV@RightListFrame
      \FV@RightListNumber}%
    \hss}}
```

\FV@BProcessLine Redefined \FV@BProcessLine in which \FancyVerbFormatText is added and tab handling is explicit.

```
\def\FV@BProcessLine#1{%
  \hbox{\FancyVerbFormatLine{%
    \FancyVerbHighlightLine{%
      \FV@ObeyTabs{\FancyVerbFormatText{#1}}}}}}
```

### 9.6.3 Line numbering

Add several new line numbering options. `numberfirstline` always numbers the first line, regardless of `stepnumber`. `stepnumberfromfirst` numbers the first line, and then every line that differs from its number by a multiple of `stepnumber`. `stepnumberoffsetvalues` determines whether line number are always an exact multiple of `stepnumber` (the new default behavior) or whether there is an offset when `firstnumber` $\neq 1$ (the old default behavior). A new option `numbers=both` is created to allow line numbers on both left and right simultaneously.

FV@NumberFirstLine

```
\newbool{FV@NumberFirstLine}
```

numberfirstline

```
\define@booleankey{FV}{numberfirstline}%
 {\booltrue{FV@NumberFirstLine}}%
 {\boolfalse{FV@NumberFirstLine}}
\fvset{numberfirstline=false}
```

FV@StepNumberFromFirst

```
\newbool{FV@StepNumberFromFirst}
```

stepnumberfromfirst

```
\define@booleankey{FV}{stepnumberfromfirst}%
 {\booltrue{FV@StepNumberFromFirst}}%
 {\boolfalse{FV@StepNumberFromFirst}}
\fvset{stepnumberfromfirst=false}
```

FV@StepNumberOffsetValues

```
\newbool{FV@StepNumberOffsetValues}
```

stepnumberoffsetvalues

```
\define@booleankey{FV}{stepnumberoffsetvalues}%
 {\booltrue{FV@StepNumberOffsetValues}}%
 {\boolfalse{FV@StepNumberOffsetValues}}
\fvset{stepnumberoffsetvalues=false}
```

\FV@Numbers@left

Redefine fancyvrb macro to account for `numberfirstline`, `stepnumberfromfirst`, and `stepnumberoffsetvalues`. The `\let\FancyVerbStartNum\@ne` is needed to account for the case where `firstline` is never set, and defaults to zero (`\z@`).

```
\def\FV@Numbers@left{%
  \let\FV@RightListNumber\relax
  \def\FV@LeftListNumber{%
    \ifx\FancyVerbStartNum\z@
      \let\FancyVerbStartNum\@ne
    \fi
    \ifbool{FV@StepNumberFromFirst}%
     {\@tempcnta=\FV@CodeLineNo
```

```
      \@tempcntb=\FancyVerbStartNum
      \advance\@tempcntb\FV@StepNumber
      \divide\@tempcntb\FV@StepNumber
      \multiply\@tempcntb\FV@StepNumber
      \advance\@tempcnta\@tempcntb
      \advance\@tempcnta-\FancyVerbStartNum
      \@tempcntb=\@tempcnta}%
     {\ifbool{FV@StepNumberOffsetValues}%
       {\@tempcnta=\FV@CodeLineNo
        \@tempcntb=\FV@CodeLineNo}%
       {\@tempcnta=\c@FancyVerbLine
        \@tempcntb=\c@FancyVerbLine}}%
    \divide\@tempcntb\FV@StepNumber
    \multiply\@tempcntb\FV@StepNumber
    \ifnum\@tempcnta=\@tempcntb
      \if@FV@NumberBlankLines
        \hbox to\z@{\hss\theFancyVerbLine\kern\FV@NumberSep}%
      \else
        \ifx\FV@Line\empty
        \else
          \hbox to\z@{\hss\theFancyVerbLine\kern\FV@NumberSep}%
        \fi
      \fi
    \else
      \ifbool{FV@NumberFirstLine}{%
        \ifnum\FV@CodeLineNo=\FancyVerbStartNum
          \hbox to\z@{\hss\theFancyVerbLine\kern\FV@NumberSep}%
        \fi}{}%
    \fi}%
}
```

`\FV@Numbers@right` Redefine fancyvrb macro to account for `numberfirstline`, `stepnumberfromfirst`, and `stepnumberoffsetvalues`.

```
\def\FV@Numbers@right{%
  \let\FV@LeftListNumber\relax
  \def\FV@RightListNumber{%
    \ifx\FancyVerbStartNum\z@
      \let\FancyVerbStartNum\@ne
    \fi
    \ifbool{FV@StepNumberFromFirst}%
     {\@tempcnta=\FV@CodeLineNo
      \@tempcntb=\FancyVerbStartNum
      \advance\@tempcntb\FV@StepNumber
      \divide\@tempcntb\FV@StepNumber
      \multiply\@tempcntb\FV@StepNumber
      \advance\@tempcnta\@tempcntb
      \advance\@tempcnta-\FancyVerbStartNum
      \@tempcntb=\@tempcnta}%
     {\ifbool{FV@StepNumberOffsetValues}%
       {\@tempcnta=\FV@CodeLineNo
```

```
        \@tempcntb=\FV@CodeLineNo}%
       {\@tempcnta=\c@FancyVerbLine
        \@tempcntb=\c@FancyVerbLine}}%
    \divide\@tempcntb\FV@StepNumber
    \multiply\@tempcntb\FV@StepNumber
    \ifnum\@tempcnta=\@tempcntb
      \if@FV@NumberBlankLines
        \hbox to\z@{\kern\FV@NumberSep\theFancyVerbLine\hss}%
      \else
        \ifx\FV@Line\empty
        \else
          \hbox to\z@{\kern\FV@NumberSep\theFancyVerbLine\hss}%
        \fi
      \fi
    \else
      \ifbool{FV@NumberFirstLine}{%
        \ifnum\FV@CodeLineNo=\FancyVerbStartNum
          \hbox to\z@{\hss\theFancyVerbLine\kern\FV@NumberSep}%
        \fi}{}%
    \fi}%
}
```

`\FV@Numbers@both` Define a new macro to allow `numbers=both`. This copies the definitions of `\FV@LeftListNumber` and `\FV@RightListNumber` from `\FV@Numbers@left` and `\FV@Numbers@right`, without the `\relax`'s.

```
\def\FV@Numbers@both{%
  \def\FV@LeftListNumber{%
    \ifx\FancyVerbStartNum\z@
      \let\FancyVerbStartNum\@ne
    \fi
    \ifbool{FV@StepNumberFromFirst}%
     {\@tempcnta=\FV@CodeLineNo
      \@tempcntb=\FancyVerbStartNum
      \advance\@tempcntb\FV@StepNumber
      \divide\@tempcntb\FV@StepNumber
      \multiply\@tempcntb\FV@StepNumber
      \advance\@tempcnta\@tempcntb
      \advance\@tempcnta-\FancyVerbStartNum
      \@tempcntb=\@tempcnta}%
     {\ifbool{FV@StepNumberOffsetValues}%
       {\@tempcnta=\FV@CodeLineNo
        \@tempcntb=\FV@CodeLineNo}%
       {\@tempcnta=\c@FancyVerbLine
        \@tempcntb=\c@FancyVerbLine}}%
    \divide\@tempcntb\FV@StepNumber
    \multiply\@tempcntb\FV@StepNumber
    \ifnum\@tempcnta=\@tempcntb
      \if@FV@NumberBlankLines
        \hbox to\z@{\hss\theFancyVerbLine\kern\FV@NumberSep}%
      \else
```

```
        \ifx\FV@Line\empty
        \else
          \hbox to\z@{\hss\theFancyVerbLine\kern\FV@NumberSep}%
        \fi
      \fi
    \else
      \ifbool{FV@NumberFirstLine}{%
        \ifnum\FV@CodeLineNo=\FancyVerbStartNum
          \hbox to\z@{\hss\theFancyVerbLine\kern\FV@NumberSep}%
        \fi}{}%
    \fi}%
  \def\FV@RightListNumber{%
    \ifx\FancyVerbStartNum\z@
      \let\FancyVerbStartNum\@ne
    \fi
    \ifbool{FV@StepNumberFromFirst}%
     {\@tempcnta=\FV@CodeLineNo
      \@tempcntb=\FancyVerbStartNum
      \advance\@tempcntb\FV@StepNumber
      \divide\@tempcntb\FV@StepNumber
      \multiply\@tempcntb\FV@StepNumber
      \advance\@tempcnta\@tempcntb
      \advance\@tempcnta-\FancyVerbStartNum
      \@tempcntb=\@tempcnta}%
     {\ifbool{FV@StepNumberOffsetValues}%
       {\@tempcnta=\FV@CodeLineNo
        \@tempcntb=\FV@CodeLineNo}%
       {\@tempcnta=\c@FancyVerbLine
        \@tempcntb=\c@FancyVerbLine}}%
    \divide\@tempcntb\FV@StepNumber
    \multiply\@tempcntb\FV@StepNumber
    \ifnum\@tempcnta=\@tempcntb
      \if@FV@NumberBlankLines
        \hbox to\z@{\kern\FV@NumberSep\theFancyVerbLine\hss}%
      \else
        \ifx\FV@Line\empty
        \else
          \hbox to\z@{\kern\FV@NumberSep\theFancyVerbLine\hss}%
        \fi
      \fi
    \else
      \ifbool{FV@NumberFirstLine}{%
        \ifnum\FV@CodeLineNo=\FancyVerbStartNum
          \hbox to\z@{\hss\theFancyVerbLine\kern\FV@NumberSep}%
        \fi}{}%
    \fi}%
}
```

### 9.6.4 Line highlighting or emphasis

This adds an option `highlightlines` that allows specific lines, or lines within a range, to be highlighted or otherwise emphasized.

highlightlines
\FV@HighlightLinesList

```
\define@key{FV}{highlightlines}{\def\FV@HighlightLinesList{#1}}%
\fvset{highlightlines=}
```

highlightcolor
\FV@HighlightColor

Define color for highlighting. The default is LightCyan. A good alternative for a brighter color would be LemonChiffon.

```
\define@key{FV}{highlightcolor}{\def\FancyVerbHighlightColor{#1}}%
\let\FancyVerbHighlightColor\@empty
\ifcsname definecolor\endcsname
\ifx\definecolor\relax
\else
  \definecolor{FancyVerbHighlightColor}{HTML}{E0FFFF}
  \fvset{highlightcolor=FancyVerbHighlightColor}
\fi\fi
\AtBeginDocument{%
  \ifx\FancyVerbHighlightColor\@empty
    \ifcsname definecolor\endcsname
    \ifx\definecolor\relax
    \else
    \definecolor{FancyVerbHighlightColor}{rgb}{0,1,1}
    \fvset{highlightcolor=FancyVerbHighlightColor}
    \fi\fi
  \fi}
```

\FancyVerbHighlightLine

This is the entry macro into line highlighting. By default it should do nothing. It is always invoked between `\FancyVerbFormatLine` and `\FancyVerbFormatText`, so that it can provide a background color (won't interfere with line breaking) and can override any formatting provided by `\FancyVerbFormatLine`. It is `\let` to `\FV@HighlightLine` when highlighting is active.

```
\def\FancyVerbHighlightLine#1{#1}
```

\FV@HighlightLine

This determines whether highlighting should be performed, and if so, which macro should be invoked.

```
\def\FV@HighlightLine#1{%
  \@tempcnta=\c@FancyVerbLine
  \@tempcntb=\c@FancyVerbLine
  \ifcsname FV@HighlightLine:\number\@tempcnta\endcsname
    \advance\@tempcntb\m@ne
    \ifcsname FV@HighlightLine:\number\@tempcntb\endcsname
      \advance\@tempcntb\tw@
      \ifcsname FV@HighlightLine:\number\@tempcntb\endcsname
        \let\FV@HighlightLine@Next\FancyVerbHighlightLineMiddle
      \else
        \let\FV@HighlightLine@Next\FancyVerbHighlightLineLast
```

```
      \fi
    \else
      \advance\@tempcntb\tw@
      \ifcsname FV@HighlightLine:\number\@tempcntb\endcsname
        \let\FV@HighlightLine@Next\FancyVerbHighlightLineFirst
      \else
        \let\FV@HighlightLine@Next\FancyVerbHighlightLineSingle
      \fi
    \fi
  \else
    \let\FV@HighlightLine@Next\FancyVerbHighlightLineNormal
  \fi
  \FV@HighlightLine@Next{#1}%
}
```

`\FancyVerbHighlightLineNormal` A normal line that is not highlighted or otherwise emphasized. This could be redefined to de-emphasize the line.

```
\def\FancyVerbHighlightLineNormal#1{#1}
```

`\FV@TmpLength`

```
\newlength{\FV@TmpLength}
```

`\FancyVerbHighlightLineFirst` The first line in a multi-line range.

`\fboxsep` is set to zero so as to avoid indenting the line or changing inter-line spacing. It is restored to its original value inside to prevent any undesired effects. The `\strut` is needed to get the highlighting to be the appropriate height. The `\rlap` and `\hspace` make the `\colorbox` expand to the full `\linewidth`. Note that if `\fboxsep` $\neq 0$, then we would want to use `\dimexpr\linewidth-2\fboxsep` or add `\hspace{-2\fboxsep}` at the end.

If this macro is customized so that the text cannot take up the full `\linewidth`, then adjustments may need to be made here or in the line breaking code to make sure that line breaking takes place at the appropriate location.

```
\def\FancyVerbHighlightLineFirst#1{%
  \setlength{\FV@TmpLength}{\fboxsep}%
  \setlength{\fboxsep}{0pt}%
  \colorbox{\FancyVerbHighlightColor}{%
    \setlength{\fboxsep}{\FV@TmpLength}%
    \rlap{\strut#1}%
    \hspace{\linewidth}}}
```

`\FancyVerbHighlightLineMiddle` A middle line in a multi-line range.

```
\let\FancyVerbHighlightLineMiddle\FancyVerbHighlightLineFirst
```

`\FancyVerbHighlightLineLast` The last line in a multi-line range.

```
\let\FancyVerbHighlightLineLast\FancyVerbHighlightLineFirst
```

`\FancyVerbHighlightLineSingle` A single line not in a multi-line range.

```
\let\FancyVerbHighlightLineSingle\FancyVerbHighlightLineFirst
```

\FV@HighlightLinesPrep Process the list of lines to highlight (if any). A macro is created for each line to be highlighted. During highlighting, a line is highlighted if the corresponding macro exists. All of the macro creating is ultimately within the current environment group so it stays local. `\FancyVerbHighlightLine` is `\let` to a version that will invoke the necessary logic.

```
\def\FV@HighlightLinesPrep{%
  \ifx\FV@HighlightLinesList\@empty
  \else
    \let\FancyVerbHighlightLine\FV@HighlightLine
    \expandafter\FV@HighlightLinesPrep@i
  \fi}
\def\FV@HighlightLinesPrep@i{%
  \renewcommand{\do}[1]{%
    \ifstrempty{##1}{}{\FV@HighlightLinesParse##1-\FV@Undefined}}%
  \expandafter\docsvlist\expandafter{\FV@HighlightLinesList}}
\def\FV@HighlightLinesParse#1-#2\FV@Undefined{%
  \ifstrempty{#2}%
   {\FV@HighlightLinesParse@Single{#1}}%
   {\FV@HighlightLinesParse@Range{#1}#2\relax}}
\def\FV@HighlightLinesParse@Single#1{%
  \expandafter\let\csname FV@HighlightLine:\detokenize{#1}\endcsname\relax}
\newcounter{FV@HighlightLinesStart}
\newcounter{FV@HighlightLinesStop}
\def\FV@HighlightLinesParse@Range#1#2-{%
  \setcounter{FV@HighlightLinesStart}{#1}%
  \setcounter{FV@HighlightLinesStop}{#2}%
  \stepcounter{FV@HighlightLinesStop}%
  \FV@HighlightLinesParse@Range@Loop}
\def\FV@HighlightLinesParse@Range@Loop{%
  \ifnum\value{FV@HighlightLinesStart}<\value{FV@HighlightLinesStop}\relax
    \expandafter\let\csname FV@HighlightLine:\arabic{FV@HighlightLinesStart}\endcsname\relax
    \stepcounter{FV@HighlightLinesStart}%
    \expandafter\FV@HighlightLinesParse@Range@Loop
  \fi}
\g@addto@macro\FV@FormattingPrepHook{\FV@HighlightLinesPrep}
```

## 9.7 Line breaking

The following code adds automatic line breaking functionality to fancyvrb's `Verbatim` environment. Automatic breaks may be inserted after spaces, or before or after specified characters. Breaking before or after specified characters involves scanning each line token by token to insert `\discretionary` at all potential break locations.

### 9.7.1 Options and associated macros

Begin by defining keys, with associated macros, bools, and dimens.

FV@BreakLines Turn line breaking on or off. The \FV@ListProcessLine from fancyvrb is \let to a (patched) version of the original or a version that supports line breaks.

```
\newboolean{FV@BreakLines}
\define@booleankey{FV}{breaklines}%
  {\FV@BreakLinestrue
    \let\FV@ListProcessLine\FV@ListProcessLine@Break}%
  {\FV@BreakLinesfalse
    \let\FV@ListProcessLine\FV@ListProcessLine@NoBreak}
\AtEndOfPackage{\fvset{breaklines=false}}
```

\FV@BreakIndent Indentation of continuation lines.

```
\newdimen\FV@BreakIndent
\define@key{FV}{breakindent}{\FV@BreakIndent=#1\relax}
\fvset{breakindent=0pt}
```

FV@BreakAutoIndent Auto indentation of continuation lines to indentation of original line. Adds to \FV@BreakIndent.

```
\newboolean{FV@BreakAutoIndent}
\define@booleankey{FV}{breakautoindent}%
  {\FV@BreakAutoIndenttrue}{\FV@BreakAutoIndentfalse}
\fvset{breakautoindent=true}
```

\FancyVerbBreakSymbolLeft The left-hand symbol indicating a break. Since breaking is done in such a way that a left-hand symbol will often be desired while a right-hand symbol may not be, a shorthand option breaksymbol is supplied. This shorthand convention is continued with other options applying to the left-hand symbol.

```
\define@key{FV}{breaksymbolleft}{\def\FancyVerbBreakSymbolLeft{#1}}
\define@key{FV}{breaksymbol}{\fvset{breaksymbolleft=#1}}
\fvset{breaksymbolleft=\tiny\ensuremath{\hookrightarrow}}
```

\FancyVerbBreakSymbolRight The right-hand symbol indicating a break.

```
\define@key{FV}{breaksymbolright}{\def\FancyVerbBreakSymbolRight{#1}}
\fvset{breaksymbolright={}}
```

\FV@BreakSymbolSepLeft Separation of left break symbol from the text.

```
\newdimen\FV@BreakSymbolSepLeft
\define@key{FV}{breaksymbolsepleft}{\FV@BreakSymbolSepLeft=#1\relax}
\define@key{FV}{breaksymbolsep}{\fvset{breaksymbolsepleft=#1}}
\fvset{breaksymbolsepleft=1em}
```

\FV@BreakSymbolSepRight Separation of right break symbol from the text.

```
\newdimen\FV@BreakSymbolSepRight
\define@key{FV}{breaksymbolsepright}{\FV@BreakSymbolSepRight=#1\relax}
\fvset{breaksymbolsepright=1em}
```

\FV@BreakSymbolIndentLeft Additional left indentation to make room for the left break symbol.

```
\newdimen\FV@BreakSymbolIndentLeft
\settowidth{\FV@BreakSymbolIndentLeft}{\ttfamily xxxx}
\define@key{FV}{breaksymbolindentleft}{\FV@BreakSymbolIndentLeft=#1\relax}
\define@key{FV}{breaksymbolindent}{\fvset{breaksymbolindentleft=#1}}
```

\FV@BreakSymbolIndentRight Additional right indentation to make room for the right break symbol.

```
\newdimen\FV@BreakSymbolIndentRight
\settowidth{\FV@BreakSymbolIndentRight}{\ttfamily xxxx}
\define@key{FV}{breaksymbolindentright}{\FV@BreakSymbolIndentRight=#1\relax}
```

We need macros that contain the logic for typesetting the break symbols. By default, the symbol macros contain everything regarding the symbol and its typesetting, while these macros contain pure logic. The symbols should be wrapped in braces so that formatting commands (for example, `\tiny`) don't escape.

\FancyVerbBreakSymbolLeftLogic The left break symbol should only appear with continuation lines. Note that `linenumber` here refers to local line numbering for the broken line, *not* line numbering for all lines in the environment being typeset.

```
\newcommand{\FancyVerbBreakSymbolLeftLogic}[1]{%
  \ifnum\value{linenumber}=1\relax\else{#1}\fi}
```

FancyVerbLineBreakLast We need a counter for keeping track of the local line number for the last segment of a broken line, so that we can avoid putting a right continuation symbol there. A line that is broken will ultimately be processed twice when there is a right continuation symbol, once to determine the local line numbering, and then again for actual insertion into the document.

```
\newcounter{FancyVerbLineBreakLast}
```

\FV@SetLineBreakLast Store the local line number for the last continuation line.

```
\newcommand{\FV@SetLineBreakLast}{%
  \setcounter{FancyVerbLineBreakLast}{\value{linenumber}}}
```

\FancyVerbBreakSymbolRightLogic Only insert a right break symbol if not on the last continuation line.

```
\newcommand{\FancyVerbBreakSymbolRightLogic}[1]{%
  \ifnum\value{linenumber}=\value{FancyVerbLineBreakLast}\relax\else{#1}\fi}
```

\FancyVerbBreakStart Macro that starts fine-tuned breaking (`breakanywhere`, `breakbefore`, `breakafter`) by examining a line token-by-token. Initially `\let` to `\relax`; later `\let` to `\FV@Break` as appropriate.

```
\let\FancyVerbBreakStart\relax
```

\FancyVerbBreakStop Macro that stops the fine-tuned breaking region started by `\FancyVerbBreakStart`. Initially `\let` to `\relax`; later `\let` to `\FV@EndBreak` as appropriate.

```
\let\FancyVerbBreakStop\relax
```

\FV@Break@Token Macro that controls token handling between `\FancyVerbBreakStart` and `\FancyVerbBreakStop`. Initially `\let` to `\relax`; later `\let` to `\FV@Break@AnyToken` or `\FV@Break@BeforeAfterToken` as appropriate. There is no need to `\let\FV@Break@Token\relax` when `breakanywhere`, `breakbefore`, and `breakafter` are not in use. In that case, `\FancyVerbBreakStart` and `\FancyVerbBreakStop` are `\let` to `\relax`, and `\FV@Break@Token` is never invoked.

```
\let\FV@Break@Token\relax
```

FV@BreakAnywhere Allow line breaking (almost) anywhere. Set \FV@Break and \FV@EndBreak to be used, and \let \FV@Break@Token to the appropriate macro.

```
\newboolean{FV@BreakAnywhere}
\define@booleankey{FV}{breakanywhere}%
  {\FV@BreakAnywheretrue
    \let\FancyVerbBreakStart\FV@Break
    \let\FancyVerbBreakStop\FV@EndBreak
    \let\FV@Break@Token\FV@Break@AnyToken}%
  {\FV@BreakAnywherefalse
    \let\FancyVerbBreakStart\relax
    \let\FancyVerbBreakStop\relax}
\fvset{breakanywhere=false}
```

\FV@BreakBefore Allow line breaking (almost) anywhere, but only before specified characters.

```
\define@key{FV}{breakbefore}{%
  \ifstrempty{#1}%
   {\let\FV@BreakBefore\@empty
    \let\FancyVerbBreakStart\relax
    \let\FancyVerbBreakStop\relax}%
   {\def\FV@BreakBefore{#1}%
    \let\FancyVerbBreakStart\FV@Break
    \let\FancyVerbBreakStop\FV@EndBreak
    \let\FV@Break@Token\FV@Break@BeforeAfterToken}%
}
\fvset{breakbefore={}}
```

FV@BreakBeforeGroup Determine whether breaking before specified characters is always allowed before each individual character, or is only allowed before the first in a group of identical characters.

```
\newboolean{FV@BreakBeforeGroup}
\define@booleankey{FV}{breakbeforegroup}%
 {\FV@BreakBeforeGrouptrue}%
 {\FV@BreakBeforeGroupfalse}%
\fvset{breakbeforegroup=true}
```

\FV@BreakBeforePrep We need a way to break before characters if and only if they have been specified as breaking characters. It would be possible to do that via a nested conditional, but that would be messy. It is much simpler to create an empty macro whose name contains the character, and test for the existence of this macro. This needs to be done inside a \begingroup...\endgroup so that the macros do not have to be cleaned up manually. A good place to do this is in \FV@FormattingPrep, which is inside a group and before processing starts. The macro is added to \FV@FormattingPrepHook, which contains fvextra exntensions to \FV@FormattingPrep, after \FV@BreakAfterPrep is defined below.

The procedure here is a bit roundabout. We need to use \FV@EscChars to handle character escapes, but the character redefinitions need to be kept local, requiring that we work within a \begingroup...\endgroup. So we loop through the breaking tokens and assemble a macro that will itself define character macros.

Only this defining macro is declared global, and it contains *expanded* characters so that there is no longer any dependence on `\FV@EscChars`.

A pdfTeX-compatible version for working with UTF-8 is defined later, and `\FV@BreakBeforePrep` is `\let` to it under pdfTeX as necessary.

```
\def\FV@BreakBeforePrep{%
  \ifx\FV@BreakBefore\@empty\relax
  \else
    \gdef\FV@BreakBefore@Def{}%
    \begingroup
    \def\FV@BreakBefore@Process##1##2\FV@Undefined{%
      \expandafter\FV@BreakBefore@Process@i\expandafter{##1}%
      \expandafter\ifx\expandafter\relax\detokenize{##2}\relax
      \else
        \FV@BreakBefore@Process##2\FV@Undefined
      \fi
    }%
    \def\FV@BreakBefore@Process@i##1{%
      \g@addto@macro\FV@BreakBefore@Def{%
        \@namedef{FV@BreakBefore@Token\detokenize{##1}}{}}%
    }%
    \FV@EscChars
    \expandafter\FV@BreakBefore@Process\FV@BreakBefore\FV@Undefined
    \endgroup
    \FV@BreakBefore@Def
  \fi
}
```

`\FV@BreakAfter` Allow line breaking (almost) anywhere, but only after specified characters.

```
\define@key{FV}{breakafter}{%
  \ifstrempty{#1}%
   {\let\FV@BreakAfter\@empty
    \let\FancyVerbBreakStart\relax
    \let\FancyVerbBreakStop\relax}%
   {\def\FV@BreakAfter{#1}%
    \let\FancyVerbBreakStart\FV@Break
    \let\FancyVerbBreakStop\FV@EndBreak
    \let\FV@Break@Token\FV@Break@BeforeAfterToken}%
}
\fvset{breakafter={}}
```

`FV@BreakAfterGroup` Determine whether breaking after specified characters is always allowed after each individual character, or is only allowed after groups of identical characters.

```
\newboolean{FV@BreakAfterGroup}
\define@booleankey{FV}{breakaftergroup}%
 {\FV@BreakAfterGrouptrue}%
 {\FV@BreakAfterGroupfalse}%
\fvset{breakaftergroup=true}
```

`\FV@BreakAfterPrep` This is the `breakafter` equivalent of `\FV@BreakBeforePrep`. It is also used within `\FV@FormattingPrep`. The order of `\FV@BreakBeforePrep` and `\FV@BreakAfterPrep`

is important; \FV@BreakAfterPrep must always be second, because it checks for conflicts with breakbefore.

A pdfTeX-compatible version for working with UTF-8 is defined later, and \FV@BreakAfterPrep is \let to it under pdfTeX as necessary.

```
\def\FV@BreakAfterPrep{%
  \ifx\FV@BreakAfter\@empty\relax
  \else
    \gdef\FV@BreakAfter@Def{}%
    \begingroup
    \def\FV@BreakAfter@Process##1##2\FV@Undefined{%
      \expandafter\FV@BreakAfter@Process@i\expandafter{##1}%
      \expandafter\ifx\expandafter\relax\detokenize{##2}\relax
      \else
        \FV@BreakAfter@Process##2\FV@Undefined
      \fi
    }%
    \def\FV@BreakAfter@Process@i##1{%
      \ifcsname FV@BreakBefore@Token\detokenize{##1}\endcsname
        \ifthenelse{\boolean{FV@BreakBeforeGroup}}%
         {\ifthenelse{\boolean{FV@BreakAfterGroup}}%
           {}%
           {\PackageError{fvextra}%
            {Conflicting breakbeforegroup and breakaftergroup for "\detokenize{##1}"}%
            {Conflicting breakbeforegroup and breakaftergroup for "\detokenize{##1}"}}}%
         {\ifthenelse{\boolean{FV@BreakAfterGroup}}%
           {\PackageError{fvextra}%
             {Conflicting breakbeforegroup and breakaftergroup for "\detokenize{##1}"}%
             {Conflicting breakbeforegroup and breakaftergroup for "\detokenize{##1}"}}%
           {}}%
      \fi
      \g@addto@macro\FV@BreakAfter@Def{%
        \@namedef{FV@BreakAfter@Token\detokenize{##1}}{}}%
    }%
    \FV@EscChars
    \expandafter\FV@BreakAfter@Process\FV@BreakAfter\FV@Undefined
    \endgroup
    \FV@BreakAfter@Def
  \fi
}
```

Now that \FV@BreakBeforePrep and \FV@BreakAfterPrep are defined, add them to \FV@FormattingPrepHook, which is the fvextra extension to \FV@FormattingPrep. The ordering here is important, since \FV@BreakAfterPrep contains compatibility checks with \FV@BreakBeforePrep, and thus must be used after it. Also, we have to check for the pdfTeX engine with inputenc using UTF-8, and use the UTF macros instead when that is the case.

```
\g@addto@macro\FV@FormattingPrepHook{%
  \ifcsname pdfmatch\endcsname
  \ifx\pdfmatch\relax
```

```
  \else
    \ifcsname inputencodingname\endcsname
    \ifx\inputencodingname\relax
    \else
      \ifdefstring{\inputencodingname}{utf8}%
       {\let\FV@BreakBeforePrep\FV@BreakBeforePrep@UTF
        \let\FV@BreakAfterPrep\FV@BreakAfterPrep@UTF}%
       {}%
    \fi\fi
  \fi\fi
  \FV@BreakBeforePrep\FV@BreakAfterPrep}
```

\FancyVerbBreakAnywhereSymbolPre

The pre-break symbol for breaks introduced by `breakanywhere`. That is, the symbol before breaks that occur between characters, rather than at spaces.

```
\define@key{FV}{breakanywheresymbolpre}{%
  \ifstrempty{#1}%
    {\def\FancyVerbBreakAnywhereSymbolPre{}}%
    {\def\FancyVerbBreakAnywhereSymbolPre{\hbox{#1}}}}
\fvset{breakanywheresymbolpre={\,\footnotesize\ensuremath{_\rfloor}}}
```

\FancyVerbBreakAnywhereSymbolPost

The post-break symbol for breaks introduced by `breakanywhere`.

```
\define@key{FV}{breakanywheresymbolpost}{%
  \ifstrempty{#1}%
    {\def\FancyVerbBreakAnywhereSymbolPost{}}%
    {\def\FancyVerbBreakAnywhereSymbolPost{\hbox{#1}}}}
\fvset{breakanywheresymbolpost={}}
```

\FancyVerbBreakBeforeSymbolPre

The pre-break symbol for breaks introduced by `breakbefore`.

```
\define@key{FV}{breakbeforesymbolpre}{%
  \ifstrempty{#1}%
    {\def\FancyVerbBreakBeforeSymbolPre{}}%
    {\def\FancyVerbBreakBeforeSymbolPre{\hbox{#1}}}}
\fvset{breakbeforesymbolpre={\,\footnotesize\ensuremath{_\rfloor}}}
```

\FancyVerbBreakBeforeSymbolPost

The post-break symbol for breaks introduced by `breakbefore`.

```
\define@key{FV}{breakbeforesymbolpost}{%
  \ifstrempty{#1}%
    {\def\FancyVerbBreakBeforeSymbolPost{}}%
    {\def\FancyVerbBreakBeforeSymbolPost{\hbox{#1}}}}
\fvset{breakbeforesymbolpost={}}
```

\FancyVerbBreakAfterSymbolPre

The pre-break symbol for breaks introduced by `breakafter`.

```
\define@key{FV}{breakaftersymbolpre}{%
  \ifstrempty{#1}%
    {\def\FancyVerbBreakAfterSymbolPre{}}%
    {\def\FancyVerbBreakAfterSymbolPre{\hbox{#1}}}}
\fvset{breakaftersymbolpre={\,\footnotesize\ensuremath{_\rfloor}}}
```

\FancyVerbBreakAfterSymbolPost The post-break symbol for breaks introduced by `breakafter`.

```
\define@key{FV}{breakaftersymbolpost}{%
  \ifstrempty{#1}%
    {\def\FancyVerbBreakAfterSymbolPost{}}%
    {\def\FancyVerbBreakAfterSymbolPost{\hbox{#1}}}}
\fvset{breakaftersymbolpost={}}
```

\FancyVerbBreakAnywhereBreak The macro governing breaking for `breakanywhere=true`.

```
\newcommand{\FancyVerbBreakAnywhereBreak}{%
  \discretionary{\FancyVerbBreakAnywhereSymbolPre}%
   {\FancyVerbBreakAnywhereSymbolPost}{}}
```

\FancyVerbBreakBeforeBreak The macro governing breaking for `breakbefore=true`.

```
\newcommand{\FancyVerbBreakBeforeBreak}{%
  \discretionary{\FancyVerbBreakBeforeSymbolPre}%
   {\FancyVerbBreakBeforeSymbolPost}{}}
```

\FancyVerbBreakAfterBreak The macro governing breaking for `breakafter=true`.

```
\newcommand{\FancyVerbBreakAfterBreak}{%
  \discretionary{\FancyVerbBreakAfterSymbolPre}%
   {\FancyVerbBreakAfterSymbolPost}{}}
```

### 9.7.2 Line breaking implementation

#### Helper macros

\FV@LineBox A box for saving a line of text, so that its dimensions may be determined and thus we may figure out if it needs line breaking.

```
\newsavebox{\FV@LineBox}
```

\FV@LineIndentBox A box for saving the indentation of code, so that its dimensions may be determined for use in auto-indentation of continuation lines.

```
\newsavebox{\FV@LineIndentBox}
```

\FV@LineIndentChars A macro for storing the indentation characters, if any, of a given line. For use in auto-indentation of continuation lines

```
\let\FV@LineIndentChars\@empty
```

\FV@GetLineIndent A macro that takes a line and determines the indentation, storing the indentation chars in `\FV@LineIndentChars`.

```
\def\FV@CleanRemainingChars#1\FV@Undefined{}
\def\FV@GetLineIndent{\afterassignment\FV@CheckIndentChar\let\FV@NextChar=}
\def\FV@CheckIndentChar{%
  \ifx\FV@NextChar\FV@Undefined\relax
    \let\FV@Next=\relax
  \else
    \ifx\FV@NextChar\FV@Space@ifx\relax
      \g@addto@macro{\FV@LineIndentChars}{\FV@Space@ifx}%
```

```
      \let\FV@Next=\FV@GetLineIndent
    \else
      \ifx\FV@NextChar\FV@Tab@ifx\relax
        \g@addto@macro{\FV@LineIndentChars}{\FV@Tab@ifx}%
        \let\FV@Next=\FV@GetLineIndent
      \else
        \let\FV@Next=\FV@CleanRemainingChars
      \fi
    \fi
  \fi
  \FV@Next
}
```

### Tab expansion

The fancyvrb option `obeytabs` uses a clever algorithm involving boxing and unboxing to expand tabs based on tab stops rather than a fixed number of equivalent space characters. (See the definitions of `\FV@@ObeyTabs` and `\FV@TrueTab` in section 9.5.2.) Unfortunately, since this involves `\hbox`, it interferes with the line breaking algorithm, and an alternative is required.

There are probably many ways tab expansion could be performed while still allowing line breaks. The current approach has been chosen because it is relatively straightforward and yields identical results to the case without line breaks. Line breaking involves saving a line in a box, and determining whether the box is too wide. During this process, if `obeytabs=true`, `\FV@TrueTabSaveWidth`, which is inside `\FV@TrueTab`, is `\let` to a version that saves the width of every tab in a macro. When a line is broken, all tabs within it will then use a variant of `\FV@TrueTab` that sequentially retrieves the saved widths. This maintains the exact behavior of the case without line breaks.

Note that the special version of `\FV@TrueTab` is based on the fvextra patched version of `\FV@TrueTab`, not on the original `\FV@TrueTab` defined in fancyvrb.

`\FV@TrueTab@UseWidth` Version of `\FV@TrueTab` that uses pre-computed tab widths.

```
\def\FV@TrueTab@UseWidth{%
  \@tempdima=\csname FV@TrueTab:Width\arabic{FV@TrueTabCounter}\endcsname sp\relax
  \stepcounter{FV@TrueTabCounter}%
  \hbox to\@tempdima{\hss\FV@TabChar}}
```

### Line scanning and break insertion macros

The strategy here is to scan a line token-by-token, and insert breaks at appropriate points. An alternative would be to make characters active, and have them expand to literal versions of themselves plus appropriate breaks. Both approaches have advantages and drawbacks. A catcode-based approach could work, but in general would require redefining some existing active characters to insert both appropriate breaks and their original definitions. The current approach works regardless of catcodes. It is also convenient for working with macros that expand to single

characters, such as those created in highlighting code with Pygments (which is used by minted and pythontex). In that case, working with active characters would not be enough, and scanning for macros (or redefining them) is necessary. With the current approach, working with more complex macros is also straightforward. Adding support for line breaks within a macro simply requires wrapping macro contents with `\FancyVerbBreakStart...\FancyVerbBreakStop`. A catcode-based approach could require `\scantokens` or a similar retokenization in some cases, but would have the advantage that in other cases no macro redefinition would be needed.

`\FV@Break` The entry macro for breaking lines, either anywhere or before/after specified characters. The current line (or argument) will be scanned token by token/group by group, and accumulated (with added potential breaks) in `\FV@TmpLine`. After scanning is complete, `\FV@TmpLine` will be inserted. It would be possible to insert each token/group into the document immediately after it is scanned, instead of accumulating them in a "buffer." But that would interfere with macros. Even in the current approach, macros that take optional arguments are problematic.[6] The last token is tracked with `\FV@LastToken`, to allow lookbehind when breaking by groups of identical characters. `\FV@LastToken` is `\let` to `\FV@Undefined` any time the last token was something that shouldn't be compared against (for example, a non-empty group), and it is not reset whenever the last token may be ignored (for example, `{}`). When setting `\FV@LastToken`, it is vital always to use `\let\FV@LastToken=...` so that `\let\FV@LastToken==` will work (so that the equals sign `=` won't break things).

The current definition of `\FV@Break@Token` is swapped for a UTF-8 compatible one under pdfTeX when necessary. The standard macros are defined next, since they make the algorithms simpler to understand. The more complex `UTF` variants are defined later. When swapping for the `UTF` macros, it is important to make sure that pdfTeX is indeed in use, that inputenc is indeed in use, and that the current encoding is UTF-8. The checks take into account the possibility of an errant `\ifx` test creating a previously non-existent macro and then `\let`ting it to `\relax`.

```
\def\FV@Break{%
  \def\FV@TmpLine{}%
  \let\FV@LastToken=\FV@Undefined
  \ifcsname pdfmatch\endcsname
  \ifx\pdfmatch\relax
  \else
    \ifcsname inputencodingname\endcsname
    \ifx\inputencodingname\relax
    \else
      \ifdefstring{\inputencodingname}{utf8}%
       {\ifx\FV@Break@Token\FV@Break@AnyToken
          \let\FV@Break@Token\FV@Break@AnyToken@UTF
        \else
```

[6] Through a suitable definition that tracks the current state and looks for square brackets, this might be circumvented. Then again, in verbatim contexts, macro use should be minimal, so the restriction to macros without optional arguments should generally not be an issue.

```
          \ifx\FV@Break@Token\FV@Break@BeforeAfterToken
            \let\FV@Break@Token\FV@Break@BeforeAfterToken@UTF
          \fi
        \fi}%
       {}%
    \fi\fi
  \fi\fi
  \FV@Break@Scan
}
```

\FV@EndBreak

```
\def\FV@EndBreak{\FV@TmpLine}
```

\FV@Break@Scan Look ahead via `\@ifnextchar`. Don't do anything if we're at the end of the region to be scanned. Otherwise, invoke a macro to deal with what's next based on whether it is math, or a group, or something else.

This and some following macros are defined inside of groups, to ensure proper catcodes.

```
\begingroup
\catcode`\$=3%
\gdef\FV@Break@Scan{%
  \@ifnextchar\FV@EndBreak%
   {}%
   {\ifx\@let@token$\relax
      \let\FV@Break@Next\FV@Break@Math
    \else
      \ifx\@let@token\bgroup\relax
        \let\FV@Break@Next\FV@Break@Group
      \else
        \let\FV@Break@Next\FV@Break@Token
      \fi
    \fi
    \FV@Break@Next}%
}
\endgroup
```

\FV@Break@Math Grab an entire math span, and insert it into `\FV@TmpLine`. Due to grouping, this works even when math contains things like `\text{$x$}`. After dealing with the math span, continue scanning.

```
\begingroup
\catcode`\$=3%
\gdef\FV@Break@Math$#1${%
  \g@addto@macro{\FV@TmpLine}{$#1$}%
  \let\FV@LastToken=\FV@Undefined
  \FV@Break@Scan}
\endgroup
```

\FV@Break@Group Grab the group, and insert it into `\FV@TmpLine` (as a group) before continuing scanning.

```
\def\FV@Break@Group#1{%
  \g@addto@macro{\FV@TmpLine}{{#1}}%
  \ifstrempty{#1}{}{\let\FV@LastToken=\FV@Undefined}%
  \FV@Break@Scan}
```

\FV@Break@AnyToken

Deal with breaking around any token. This doesn't break macros with *mandatory* arguments, because `\FancyVerbBreakAnywhereBreak` is inserted *before* the token. Groups themselves are added without any special handling. So a macro would end up right next to its original arguments, without anything being inserted. Optional arguments will cause this approach to fail; there is currently no attempt to identify them, since that is a much harder problem.

If it is ever necessary, it would be possible to create a more sophisticated version involving catcode checks via `\ifcat`. Something like this:

```
\begingroup
\catcode`\a=11%
\catcode`\+=12%
\gdef\FV@Break...
  \ifcat\noexpand#1a%
    \g@addto@macro{\FV@TmpLine}...
  \else
...
\endgroup
```

```
\def\FV@Break@AnyToken#1{%
  \g@addto@macro{\FV@TmpLine}{\FancyVerbBreakAnywhereBreak#1}%
  \FV@Break@Scan}
```

\FV@Break@BeforeAfterToken

Deal with breaking around only specified tokens. This is a bit trickier. We only break if a macro corresponding to the token exists. We also need to check whether the specified token should be grouped, that is, whether breaks are allowed between identical characters. All of this has to be written carefully so that nothing is accidentally inserted into the stream for future scanning.

Dealing with tokens followed by empty groups (for example, `\x{}`) is particularly challenging when we want to avoid breaks between identical characters. When a token is followed by a group, we need to save the current token for later reference (`\x` in the example), then capture and save the following group, and then—only if the group was empty—see if the following token is identical to the old saved token.

```
\def\FV@Break@BeforeAfterToken#1{%
  \ifcsname FV@BreakBefore@Token\detokenize{#1}\endcsname
    \let\FV@Break@Next\FV@Break@BeforeTokenBreak
  \else
    \ifcsname FV@BreakAfter@Token\detokenize{#1}\endcsname
      \let\FV@Break@Next\FV@Break@AfterTokenBreak
    \else
      \let\FV@Break@Next\FV@Break@BeforeAfterTokenNoBreak
    \fi
```

```
  \fi
  \FV@Break@Next{#1}%
}
\def\FV@Break@BeforeAfterTokenNoBreak#1{%
  \g@addto@macro{\FV@TmpLine}{#1}%
  \let\FV@LastToken=#1%
  \FV@Break@Scan}
\def\FV@Break@BeforeTokenBreak#1{%
  \ifthenelse{\boolean{FV@BreakBeforeGroup}}%
   {\ifx#1\FV@LastToken\relax
      \ifcsname FV@BreakAfter@Token\detokenize{#1}\endcsname
        \let\FV@Break@Next\FV@Break@BeforeTokenBreak@AfterRescan
        \def\FV@RescanToken{#1}%
      \else
        \g@addto@macro{\FV@TmpLine}{#1}%
        \let\FV@Break@Next\FV@Break@Scan
        \let\FV@LastToken=#1%
      \fi
    \else
      \ifcsname FV@BreakAfter@Token\detokenize{#1}\endcsname
        \g@addto@macro{\FV@TmpLine}{\FancyVerbBreakBeforeBreak}%
        \let\FV@Break@Next\FV@Break@BeforeTokenBreak@AfterRescan
        \def\FV@RescanToken{#1}%
      \else
        \g@addto@macro{\FV@TmpLine}{\FancyVerbBreakBeforeBreak#1}%
        \let\FV@Break@Next\FV@Break@Scan
        \let\FV@LastToken=#1%
      \fi
    \fi}%
   {\ifcsname FV@BreakAfter@Token\detokenize{#1}\endcsname
      \g@addto@macro{\FV@TmpLine}{\FancyVerbBreakBeforeBreak}%
      \let\FV@Break@Next\FV@Break@BeforeTokenBreak@AfterRescan
      \def\FV@RescanToken{#1}%
    \else
      \g@addto@macro{\FV@TmpLine}{\FancyVerbBreakBeforeBreak#1}%
      \let\FV@Break@Next\FV@Break@Scan
      \let\FV@LastToken=#1%
    \fi}%
  \FV@Break@Next}
\def\FV@Break@BeforeTokenBreak@AfterRescan{%
  \expandafter\FV@Break@AfterTokenBreak\FV@RescanToken}
\def\FV@Break@AfterTokenBreak#1{%
  \let\FV@LastToken=#1%
  \@ifnextchar\FV@Space@ifx%
   {\g@addto@macro{\FV@TmpLine}{#1}\FV@Break@Scan}%
   {\ifthenelse{\boolean{FV@BreakAfterGroup}}%
     {\ifx\@let@token#1\relax
        \g@addto@macro{\FV@TmpLine}{#1}%
        \let\FV@Break@Next\FV@Break@Scan
      \else
```

```
        \ifx\@let@token\bgroup\relax
          \g@addto@macro{\FV@TmpLine}{#1}%
          \let\FV@Break@Next\FV@Break@AfterTokenBreak@Group
        \else
          \g@addto@macro{\FV@TmpLine}{#1\FancyVerbBreakAfterBreak}%
          \let\FV@Break@Next\FV@Break@Scan
        \fi
      \fi}%
     {\g@addto@macro{\FV@TmpLine}{#1\FancyVerbBreakAfterBreak}%
      \let\FV@Break@Next\FV@Break@Scan}%
    \FV@Break@Next}%
}
\def\FV@Break@AfterTokenBreak@Group#1{%
  \g@addto@macro{\FV@TmpLine}{{#1}}%
  \ifstrempty{#1}%
   {\let\FV@Break@Next\FV@Break@AfterTokenBreak@Group@i}%
   {\let\FV@Break@Next\FV@Break@Scan\let\FV@LastToken=\FV@Undefined}%
  \FV@Break@Next}
\def\FV@Break@AfterTokenBreak@Group@i{%
  \@ifnextchar\FV@LastToken%
   {\FV@Break@Scan}%
   {\g@addto@macro{\FV@TmpLine}{\FancyVerbBreakAfterBreak}%
    \FV@Break@Scan}}
```

**Line scanning and break insertion macros for pdfTeX with UTF-8**

The macros above work with the XeTeX and LuaTeX engines and are also fine for pdfTeX with 8-bit character encodings. Unfortunately, pdfTeX works with multi-byte UTF-8 code points at the byte level, making things significantly trickier. The code below re-implements the macros in a manner compatible with the inputenc package with option `utf8`. Note that there is no attempt for compatibility with `utf8x`; `utf8` has been significantly improved in recent years and should be sufficient in the vast majority of cases. And implementing variants for `utf8` was already sufficiently painful.

All of the `UTF` macros are only needed with pdfTeX, so they are created conditionally, inspired by the approach of the iftex package. The pdfTeX test deals with the possibility that a previous test using `\ifx` rather than the cleaner `\ifcsname` has already been performed.

```
\ifcsname pdfmatch\endcsname
\ifx\pdfmatch\relax
\else
```

`\FV@UTF@two@octets`
`\FV@UTF@three@octets`
`\FV@UTF@four@octets`

These are variants of the `utf8.def` macros that capture all bytes of a multi-byte code point and then pass them on as a single argument for further processing. The current `\FV@Break` (or other invoking macro) will have `\let \FV@Break@NextNext` to an appropriate macro that performs further processing. All code points are checked for validity here so as to raise errors as early as possible. Otherwise an invalid terminal byte sequence might gobble `\FV@EndBreak`, `\FV@Undefined`, or another delimiting macro, potentially making debugging much more difficult. It

would be possible to use `\UTFviii@defined{`⟨*bytes*⟩`}` to trigger an error directly, but the current approach is to attempt to typeset invalid code points, which should trigger errors without relying on the details of the `utf8.def` implementation.

```
\def\FV@UTF@two@octets#1#2{%
  \ifcsname u8:\detokenize{#1#2}\endcsname
  \else
    #1#2%
  \fi
  \FV@Break@NextNext{#1#2}}
\def\FV@UTF@three@octets#1#2#3{%
  \ifcsname u8:\detokenize{#1#2#3}\endcsname
  \else
    #1#2#3%
  \fi
  \FV@Break@NextNext{#1#2#3}}
\def\FV@UTF@four@octets#1#2#3#4{%
  \ifcsname u8:\detokenize{#1#2#3#4}\endcsname
  \else
    #1#2#3#4%
  \fi
  \FV@Break@NextNext{#1#2#3#4}}
```

`\FV@U8:<byte>` Define macros for each active byte. These are used for determining whether the current token is the first byte in a multi-byte sequence, and if so, invoking the necessary macro to capture the remaining bytes. The code is adapted from the beginning of `utf8.def`. Completely capitalized macro names are used to avoid having to worry about `\uppercase`.

```
\begingroup
\catcode`\~=13
\catcode`\"=12
\def\FV@UTFviii@loop{%
  \uccode`\~\count@
  \uppercase\expandafter{\FV@UTFviii@Tmp}%
  \advance\count@\@ne
  \ifnum\count@<\@tempcnta
  \expandafter\FV@UTFviii@loop
  \fi}
```

Setting up 2-byte UTF-8:

```
\count@"C2
\@tempcnta"E0
\def\FV@UTFviii@Tmp{\expandafter\gdef\csname FV@U8:\string~\endcsname{%
  \FV@UTF@two@octets}}
\FV@UTFviii@loop
```

Setting up 3-byte UTF-8:

```
\count@"E0
\@tempcnta"F0
\def\FV@UTFviii@Tmp{\expandafter\gdef\csname FV@U8:\string~\endcsname{%
  \FV@UTF@three@octets}}
```

```
\FV@UTFviii@loop
```

Setting up 4-byte UTF-8:

```
\count@"F0
\@tempcnta"F4
\def\FV@UTFviii@Tmp{\expandafter\gdef\csname FV@U8:\string~\endcsname{%
  \FV@UTF@four@octets}}
\FV@UTFviii@loop
\endgroup
```

\FV@BreakBeforePrep@UTF

We need UTF variants of the breakbefore and breakafter prep macros. These are only ever used with inputenc with UTF-8. There is no need for encoding checks here; checks are performed in \FV@FormattingPrepHook (checks are inserted into it after the non-UTF macro definitions).

```
\def\FV@BreakBeforePrep@UTF{%
  \ifx\FV@BreakBefore\@empty\relax
  \else
    \gdef\FV@BreakBefore@Def{}%
    \begingroup
    \def\FV@BreakBefore@Process##1{%
      \ifcsname FV@U8:\detokenize{##1}\endcsname
        \expandafter\let\expandafter\FV@Break@Next\csname FV@U8:\detokenize{##1}\endcsname
        \let\FV@Break@NextNext\FV@BreakBefore@Process@ii
      \else
        \ifx##1\FV@Undefined
          \let\FV@Break@Next\@gobble
        \else
          \let\FV@Break@Next\FV@BreakBefore@Process@i
        \fi
      \fi
      \FV@Break@Next##1%
    }%
    \def\FV@BreakBefore@Process@i##1{%
      \expandafter\FV@BreakBefore@Process@ii\expandafter{##1}}%
    \def\FV@BreakBefore@Process@ii##1{%
      \g@addto@macro\FV@BreakBefore@Def{%
        \@namedef{FV@BreakBefore@Token\detokenize{##1}}{}}%
      \FV@BreakBefore@Process
    }%
    \FV@EscChars
    \expandafter\FV@BreakBefore@Process\FV@BreakBefore\FV@Undefined
    \endgroup
    \FV@BreakBefore@Def
  \fi
}
```

\FV@BreakAfterPrep@UTF

```
\def\FV@BreakAfterPrep@UTF{%
  \ifx\FV@BreakAfter\@empty\relax
  \else
```

```
    \gdef\FV@BreakAfter@Def{}%
    \begingroup
    \def\FV@BreakAfter@Process##1{%
      \ifcsname FV@U8:\detokenize{##1}\endcsname
        \expandafter\let\expandafter\FV@Break@Next\csname FV@U8:\detokenize{##1}\endcsname
        \let\FV@Break@NextNext\FV@BreakAfter@Process@ii
      \else
        \ifx##1\FV@Undefined
          \let\FV@Break@Next\@gobble
        \else
          \let\FV@Break@Next\FV@BreakAfter@Process@i
        \fi
      \fi
      \FV@Break@Next##1%
    }%
    \def\FV@BreakAfter@Process@i##1{%
      \expandafter\FV@BreakAfter@Process@ii\expandafter{##1}}%
    \def\FV@BreakAfter@Process@ii##1{%
      \ifcsname FV@BreakBefore@Token\detokenize{##1}\endcsname
        \ifthenelse{\boolean{FV@BreakBeforeGroup}}%
         {\ifthenelse{\boolean{FV@BreakAfterGroup}}%
           {}%
           {\PackageError{fvextra}%
            {Conflicting breakbeforegroup and breakaftergroup for "\detokenize{##1}"}%
            {Conflicting breakbeforegroup and breakaftergroup for "\detokenize{##1}"}}}%
         {\ifthenelse{\boolean{FV@BreakAfterGroup}}%
           {\PackageError{fvextra}%
             {Conflicting breakbeforegroup and breakaftergroup for "\detokenize{##1}"}%
             {Conflicting breakbeforegroup and breakaftergroup for "\detokenize{##1}"}}%
           {}}%
      \fi
      \g@addto@macro\FV@BreakAfter@Def{%
        \@namedef{FV@BreakAfter@Token\detokenize{##1}}{}}%
      \FV@BreakAfter@Process
    }%
    \FV@EscChars
    \expandafter\FV@BreakAfter@Process\FV@BreakAfter\FV@Undefined
    \endgroup
    \FV@BreakAfter@Def
  \fi
}
```

`\FV@Break@AnyToken@UTF` Instead of just adding each token to `\FV@TmpLine` with a preceding break, also check for multi-byte code points and capture the remaining bytes when they are encountered.

```
\def\FV@Break@AnyToken@UTF#1{%
  \ifcsname FV@U8:\detokenize{#1}\endcsname
    \expandafter\let\expandafter\FV@Break@Next\csname FV@U8:\detokenize{#1}\endcsname
    \let\FV@Break@NextNext\FV@Break@AnyToken@UTF@i
  \else
```

```
    \let\FV@Break@Next\FV@Break@AnyToken@UTF@i
  \fi
  \FV@Break@Next{#1}%
}
\def\FV@Break@AnyToken@UTF@i#1{%
  \g@addto@macro{\FV@TmpLine}{\FancyVerbBreakAnywhereBreak#1}%
  \FV@Break@Scan}
```

\FV@Break@BeforeAfterToken@UTF

Due to the way that the flow works, #1 will sometimes be a single byte and sometimes be a multi-byte UTF-8 code point. As a result, it is vital use use \detokenize in the UTF-8 leading byte checks; \string would only deal with the first byte. It is also important to keep track of the distinction between \FV@Break@Next#1 and \FV@Break@Next{#1}. In some cases, a multi-byte sequence is being passed on as a single argument, so it must be enclosed in curly braces; in other cases, it is being re-inserted into the scanning stream and curly braces must be avoided lest they be interpreted as part of the original text.

```
\def\FV@Break@BeforeAfterToken@UTF#1{%
  \ifcsname FV@U8:\detokenize{#1}\endcsname
    \expandafter\let\expandafter\FV@Break@Next\csname FV@U8:\detokenize{#1}\endcsname
    \let\FV@Break@NextNext\FV@Break@BeforeAfterToken@UTF@i
  \else
    \let\FV@Break@Next\FV@Break@BeforeAfterToken@UTF@i
  \fi
  \FV@Break@Next{#1}%
}
\def\FV@Break@BeforeAfterToken@UTF@i#1{%
  \ifcsname FV@BreakBefore@Token\detokenize{#1}\endcsname
    \let\FV@Break@Next\FV@Break@BeforeTokenBreak@UTF
  \else
    \ifcsname FV@BreakAfter@Token\detokenize{#1}\endcsname
      \let\FV@Break@Next\FV@Break@AfterTokenBreak@UTF
    \else
      \let\FV@Break@Next\FV@Break@BeforeAfterTokenNoBreak@UTF
    \fi
  \fi
  \FV@Break@Next{#1}%
}
\def\FV@Break@BeforeAfterTokenNoBreak@UTF#1{%
  \g@addto@macro{\FV@TmpLine}{#1}%
  \def\FV@LastToken{#1}%
  \FV@Break@Scan}
\def\FV@Break@BeforeTokenBreak@UTF#1{%
  \def\FV@CurrentToken{#1}%
  \ifthenelse{\boolean{FV@BreakBeforeGroup}}%
   {\ifx\FV@CurrentToken\FV@LastToken\relax
      \ifcsname FV@BreakAfter@Token\detokenize{#1}\endcsname
        \let\FV@Break@Next\FV@Break@BeforeTokenBreak@AfterRescan@UTF
        \def\FV@RescanToken{#1}%
      \else
```

```
        \g@addto@macro{\FV@TmpLine}{#1}%
        \let\FV@Break@Next\FV@Break@Scan
        \def\FV@LastToken{#1}%
      \fi
    \else
      \ifcsname FV@BreakAfter@Token\detokenize{#1}\endcsname
        \g@addto@macro{\FV@TmpLine}{\FancyVerbBreakBeforeBreak}%
        \let\FV@Break@Next\FV@Break@BeforeTokenBreak@AfterRescan@UTF
        \def\FV@RescanToken{#1}%
      \else
        \g@addto@macro{\FV@TmpLine}{\FancyVerbBreakBeforeBreak#1}%
        \let\FV@Break@Next\FV@Break@Scan
        \def\FV@LastToken{#1}%
      \fi
    \fi}%
   {\ifcsname FV@BreakAfter@Token\detokenize{#1}\endcsname
      \g@addto@macro{\FV@TmpLine}{\FancyVerbBreakBeforeBreak}%
      \let\FV@Break@Next\FV@Break@BeforeTokenBreak@AfterRescan@UTF
      \def\FV@RescanToken{#1}%
    \else
      \g@addto@macro{\FV@TmpLine}{\FancyVerbBreakBeforeBreak#1}%
      \let\FV@Break@Next\FV@Break@Scan
      \def\FV@LastToken{#1}%
    \fi}%
  \FV@Break@Next}
\def\FV@Break@BeforeTokenBreak@AfterRescan@UTF{%
  \expandafter\FV@Break@AfterTokenBreak@UTF\expandafter{\FV@RescanToken}}
\def\FV@Break@AfterTokenBreak@UTF#1{%
  \def\FV@LastToken{#1}%
  \@ifnextchar\FV@Space@ifx%
   {\g@addto@macro{\FV@TmpLine}{#1}\FV@Break@Scan}%
   {\ifthenelse{\boolean{FV@BreakAfterGroup}}%
     {\g@addto@macro{\FV@TmpLine}{#1}%
      \ifx\@let@token\bgroup\relax
        \let\FV@Break@Next\FV@Break@AfterTokenBreak@Group@UTF
      \else
        \let\FV@Break@Next\FV@Break@AfterTokenBreak@UTF@i
      \fi}%
     {\g@addto@macro{\FV@TmpLine}{#1\FancyVerbBreakAfterBreak}%
      \let\FV@Break@Next\FV@Break@Scan}%
    \FV@Break@Next}%
}
\def\FV@Break@AfterTokenBreak@UTF@i#1{%
  \ifcsname FV@U8:\detokenize{#1}\endcsname
    \expandafter\let\expandafter\FV@Break@Next\csname FV@U8:\detokenize{#1}\endcsname
    \let\FV@Break@NextNext\FV@Break@AfterTokenBreak@UTF@i
  \else
    \def\FV@NextToken{#1}%
    \ifx\FV@LastToken\FV@NextToken
    \else
```

```
      \g@addto@macro{\FV@TmpLine}{\FancyVerbBreakAfterBreak}%
    \fi
    \let\FV@Break@Next\FV@Break@Scan
  \fi
  \FV@Break@Next#1}
\def\FV@Break@AfterTokenBreak@Group@UTF#1{%
  \g@addto@macro{\FV@TmpLine}{{#1}}%
  \ifstrempty{#1}%
   {\let\FV@Break@Next\FV@Break@AfterTokenBreak@Group@UTF@i}%
   {\let\FV@Break@Next\FV@Break@Scan\let\FV@LastToken=\FV@Undefined}%
  \FV@Break@Next}
\def\FV@Break@AfterTokenBreak@Group@UTF@i{%
  \@ifnextchar\bgroup%
   {\FV@Break@Scan}%
   {\FV@Break@AfterTokenBreak@Group@UTF@ii}}
\def\FV@Break@AfterTokenBreak@Group@UTF@ii#1{%
  \ifcsname FV@U8:\detokenize{#1}\endcsname
    \expandafter\let\expandafter\FV@Break@Next\csname FV@U8:\detokenize{#1}\endcsname
    \let\FV@Break@NextNext\FV@Break@AfterTokenBreak@Group@UTF@ii
  \else
    \def\FV@NextToken{#1}%
    \ifx\FV@LastToken\FV@NextToken
    \else
      \g@addto@macro{\FV@TmpLine}{\FancyVerbBreakAfterBreak}%
    \fi
    \let\FV@Break@Next\FV@Break@Scan
  \fi
  \FV@Break@Next#1}
```

End the conditional creation of the pdfTeX `UTF` macros:

```
\fi\fi
```

### Line processing before scanning

`\FV@makeLineNumber`

The lineno package is used for formatting wrapped lines and inserting break symbols. We need a version of lineno's `\makeLineNumber` that is adapted for our purposes. This is adapted directly from the example `\makeLineNumber` that is given in the lineno documentation under the discussion of internal line numbers. The `\FV@SetLineBreakLast` is needed to determine the internal line number of the last segment of the broken line, so that we can disable the right-hand break symbol on this segment. When a right-hand break symbol is in use, a line of code will be processed twice: once to determine the last internal line number, and once to use this information only to insert right-hand break symbols on the appropriate lines. During the second run, `\FV@SetLineBreakLast` is disabled by `\let`ting it to `\relax`.

```
\def\FV@makeLineNumber{%
  \hss
  \FancyVerbBreakSymbolLeftLogic{\FancyVerbBreakSymbolLeft}%
```

```
  \hbox to \FV@BreakSymbolSepLeft{\hfill}%
  \rlap{\hskip\linewidth
    \hbox to \FV@BreakSymbolSepRight{\hfill}%
    \FancyVerbBreakSymbolRightLogic{\FancyVerbBreakSymbolRight}%
    \FV@SetLineBreakLast
  }%
}
```

\FV@SaveLineBox This is the macro that does most of the work. It was inspired by Marco Daniel's code at http://tex.stackexchange.com/a/112573/10742.

This macro is invoked when a line is too long. We modify the `\linewidth` to take into account `breakindent` and `breakautoindent`, and insert `\hbox`es to fill the empty space. We also account for `breaksymbolindentleft` and `breaksymbolindentright`, but *only* when there are actually break symbols. The code is placed in a `\parbox`. Break symbols are inserted via lineno's `internallinenumbers*`, which does internal line numbers without continuity between environments (the `linenumber` counter is automatically reset). The beginning of the line has negative `\hspace` inserted to pull it out to the correct starting position. `\struts` are used to maintain correct line heights. The `\parbox` is followed by an empty `\hbox` that takes up the space needed for a right-hand break symbol (if any).

```
\def\FV@SaveLineBox#1{%
  \savebox{\FV@LineBox}{%
    \advance\linewidth by -\FV@BreakIndent
    \hbox to \FV@BreakIndent{\hfill}%
    \ifthenelse{\boolean{FV@BreakAutoIndent}}%
     {\let\FV@LineIndentChars\@empty
      \FV@GetLineIndent#1\FV@Undefined
      \savebox{\FV@LineIndentBox}{\FV@LineIndentChars}%
      \hbox to \wd\FV@LineIndentBox{\hfill}%
      \advance\linewidth by -\wd\FV@LineIndentBox
      \setcounter{FV@TrueTabCounter}{0}}%
     {}%
    \ifdefempty{\FancyVerbBreakSymbolLeft}{}%
     {\hbox to \FV@BreakSymbolIndentLeft{\hfill}%
      \advance\linewidth by -\FV@BreakSymbolIndentLeft}%
    \ifdefempty{\FancyVerbBreakSymbolRight}{}%
     {\advance\linewidth by -\FV@BreakSymbolIndentRight}%
    \parbox[t]{\linewidth}{%
      \raggedright
      \leftlinenumbers*
      \begin{internallinenumbers*}%
      \let\makeLineNumber\FV@makeLineNumber
      \noindent\hspace*{-\FV@BreakIndent}%
      \ifdefempty{\FancyVerbBreakSymbolLeft}{}{%
        \hspace*{-\FV@BreakSymbolIndentLeft}}%
      \ifthenelse{\boolean{FV@BreakAutoIndent}}%
       {\hspace*{-\wd\FV@LineIndentBox}}%
       {}%
```

```
      \strut\FancyVerbFormatText{%
        \FancyVerbBreakStart #1\FancyVerbBreakStop}\nobreak\strut
      \end{internallinenumbers*}
    }%
    \ifdefempty{\FancyVerbBreakSymbolRight}{}%
     {\hbox to \FV@BreakSymbolIndentRight{\hfill}}%
  }%
}
```

\FV@ListProcessLine@Break

This macro is based on the original `\FV@ListProcessLine` and follows it as closely as possible. The `\linewidth` is reduced by `\FV@FrameSep` and `\FV@FrameRule` so that text will not overrun frames. This is done conditionally based on which frames are in use. We save the current line in a box, and only do special things if the box is too wide. For uniformity, all text is placed in a `\parbox`, even if it doesn't need to be wrapped.

If a line is too wide, then it is passed to `\FV@SaveLineBox`. If there is no right-hand break symbol, then the saved result in `\FV@LineBox` may be used immediately. If there is a right-hand break symbol, then the line must be processed a second time, so that the right-hand break symbol may be removed from the final segment of the broken line (since it does not continue). During the first use of `\FV@SaveLineBox`, the counter `FancyVerbLineBreakLast` is set to the internal line number of the last segment of the broken line. During the second use of `\FV@SaveLineBox`, we disable this (`\let\FV@SetLineBreakLast\relax`) so that the value of `FancyVerbLineBreakLast` remains fixed and thus may be used to determine when a right-hand break symbol should be inserted.

```
\def\FV@ListProcessLine@Break#1{%
  \hbox to \hsize{%
  \kern\leftmargin
  \hbox to \linewidth{%
  \ifx\FV@RightListFrame\relax\else
    \advance\linewidth by -\FV@FrameSep
    \advance\linewidth by -\FV@FrameRule
  \fi
  \ifx\FV@LeftListFrame\relax\else
    \advance\linewidth by -\FV@FrameSep
    \advance\linewidth by -\FV@FrameRule
  \fi
  \ifx\FV@Tab\FV@TrueTab
    \let\FV@TrueTabSaveWidth\FV@TrueTabSaveWidth@Save
    \setcounter{FV@TrueTabCounter}{0}%
  \fi
  \sbox{\FV@LineBox}{%
    \FancyVerbFormatLine{%
      %\FancyVerbHighlightLine  %<-- Default definition using \rlap breaks breaking
       {\FV@ObeyTabs{\FancyVerbFormatText{#1}}}}}%
  \ifx\FV@Tab\FV@TrueTab
    \let\FV@TrueTabSaveWidth\relax
  \fi
```

```
\ifdim\wd\FV@LineBox>\linewidth
  \setcounter{FancyVerbLineBreakLast}{0}%
  \ifx\FV@Tab\FV@TrueTab
    \let\FV@Tab\FV@TrueTab@UseWidth
    \setcounter{FV@TrueTabCounter}{0}%
  \fi
  \FV@SaveLineBox{#1}%
  \ifdefempty{\FancyVerbBreakSymbolRight}{}{%
    \let\FV@SetLineBreakLast\relax
    \setcounter{FV@TrueTabCounter}{0}%
    \FV@SaveLineBox{#1}}%
  \FV@LeftListNumber
  \FV@LeftListFrame
  \FancyVerbFormatLine{%
    \FancyVerbHighlightLine{\usebox{\FV@LineBox}}}%
  \FV@RightListFrame
  \FV@RightListNumber
  \ifx\FV@Tab\FV@TrueTab@UseWidth
    \let\FV@Tab\FV@TrueTab
  \fi
\else
  \FV@LeftListNumber
  \FV@LeftListFrame
  \FancyVerbFormatLine{%
    \FancyVerbHighlightLine{%
      \parbox[t]{\linewidth}{%
        \noindent\strut\FV@ObeyTabs{\FancyVerbFormatText{#1}}\strut}}}%
  \FV@RightListFrame
  \FV@RightListNumber
\fi}%
\hss}\baselineskip\z@\lineskip\z@}
```