

Contents

<i>How to read this document</i>	2
<i>Some preliminaries on using the package</i>	3
<i>Introduction to L^AT_EX syntax</i>	4
<i>Some commands used in this document</i>	5
<i>Commands for drawing EGs</i>	5
<i>The α-system and cuts</i>	6
<i>The β-system and ligatures</i>	9
<i>The γ-system ...</i>	12
<i>Examples of graphs</i>	13
<i>Simple cuts and ligatures</i>	14
<i>Nonstandard scrolls and complex graphs</i>	16
<i>Ugly hacks</i>	19
<i>Some difficult cases and their solutions</i>	21
<i>Mutable parameters of the package</i>	21
<i>Some handy PostScript commands</i>	22
<i>Peirce's logical symbols</i>	23
<i>Ideas on further development of the package</i>	26
<i>Introduction to L^AT_EX</i>	28
<i>What is it ...</i>	28
<i>...and why use it?</i>	29
<i>Recommended further reading</i>	30
<i>References</i>	30
<i>Keyword & Command Index</i>	30
<i>Visual Index</i>	31

How to read this document

To understand this documentation, no previous knowledge of L^AT_EX² is strictly required. Its syntax is tersely explained in the beginning and code examples later on are hopefully instructive. It is, however, tacitly assumed that the reader *is* familiar with existential graphs.³

If you've never used L^AT_EX before, I *highly* suggest you read the short history from page 28 and consult the recommended further readings listed on page 30.

The commands the `egpeirce` package provides are introduced from page 5 and a few examples are presented. The examples also introduce general strategies on coping with complex graphs. Unless you are transcribing graphs from manuscripts, you probably don't need to delve into the inner workings of the code (page 16 onward).⁴

² Pronounced [ˈlaːtɛχ]/[ˈleɪtɛχ], roughly 'LAY-tekh'. From the Greek word τέχνη.

³ Good sources to get up to speed with EGs are Roberts (1973) and Peirce (2023).

⁴ Finally, if you are a T_EXnician, we need your help!

Please see e.g. pages 7 and 19 and help us find better solutions to these nuisances or problematic cases.

Some preliminaries on using the package

egpeirce is a \LaTeX package intended for drawing *existential graphs*⁵ that were invented and developed by the philosopher and polymath Charles S. Peirce (1839–1914).

The current version of egpeirce only supports *drawing* existential graphs. Specifically, it *does not* check or assure that proper syntactic rules are obeyed (see page 6 for a crude error case).

This is because the package is primarily designed for transcribing graphs from Peirce’s manuscripts. Many of these graphs have unusual features and too much automation would have been a hindrance. See page 26 for ideas on further development of the package.

The package *does* enable your document to include graphs without having to resort to external image files and it *does* automate many of the more tedious aspects of vectorizing graphs. The graphs are described in relatively simple and straightforward code and when you compile your document, \LaTeX does the actual drawing for you.

THE PACKAGE DEPENDS HEAVILY ON PostScript commands and the PsTricks \LaTeX package that interfaces them. Therefore the standard DVI output file is not capable of displaying the graphs, although it does contain all the data. Furthermore most native PDF compilers—such as pdfLaTeX—cannot directly process the source file.

Compiling the source file with \LaTeX requires you to use a converter like DVItoPS on the output file as well (and PStoPDF on the ensuing .ps file if you want to produce PDF documents). For example Xe \LaTeX is able to process the PostScript commands natively but can be quite slow and generally seems to produce slightly larger PDF files than the $\LaTeX \rightarrow$ DVItoPS \rightarrow PStoPDF method.

There are packages that *attempt* to automatically and on-the-fly wrap the PostScript commands so compilers like pdfLaTeX could accept the source. At the time of writing this, these packages are still experimental or at least quite unreliable or inconsistent.

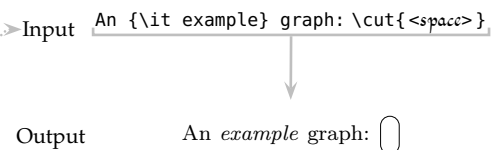
At the time I began writing the code, PGF/TIKZ was considered a newish, unfinished project and I therefore opted to use the more mature and stable PostScript backend. Using PGF/TIKZ as the drawing backend would lift the restrictions on using e.g. pdfLaTeX mentioned above. The code does rely on PsTricks and some Ps-tricks, refactoring around PGF/TIKZ would be possible, though not trivial.

THROUGHOUT THIS DOCUMENT the following typographical conventions are adopted. All verbatim code snippets are typeset in monospace typewriter text and all placeholder metasyntactic variables are typeset in *<angle bracketed aurical>*. The dotted lines point to examples, the solid lines represent the path from the input to output.

⁵ As well as linear logical operators and logical symbols (see page 23 onwards).

ON COMPATIBLE COMPILERS AND CONVERTERS, \LaTeX , Xe \LaTeX

ON PGF/TIKZ



Introduction to L^AT_EX syntax

Following is a *very concise* introduction to the syntax of L^AT_EX. The aim is to enable someone without prior knowledge about it to follow and understand *this* document.

Learning to write L^AT_EX is well beyond the purview of this documentation. Find a list for further reading to that end on page 30.

IN ORDER TO DISTINGUISH commands from text, L^AT_EX includes reserved special characters. They are: #, \$, %, &, {, }, \, ^, _ and ~. If you enter these directly as text they will not print and when misused will stop compilation and give error messages.

RESERVED SPECIAL CHARACTERS

The most interesting special character is the backslash: \. Among other things it always denotes the start of a command followed by the command name and possibly some arguments.

The curly brackets serve a double-function either as delimiting the *scope* of a command or enclosing its *arguments* if it uses them.

SCOPE OF A COMMAND

The commands \textit and \it for example *italicize* text. If a scope is not specified for \it all text after its invocation will be effected, whereas \textit{} accepts an argument. The above italicization of a single word is done either by scoping, {\it italicize}, or by using the brackets to define the argument: \textit{italicize}.

It's important to understand and pay attention to proper scoping when you draw complexly and multiply nested graph elements.

MORE COMPLEX TRANSFORMATIONS in L^AT_EX are handled with *environments*, and the egpeirce package uses them too. They are scoped and introduced with \begin{<environmentname> which must be matched with a closing \end{<environmentname>}.

COUNTERS, BOOLEAN SWITCHES,
GLOBAL SCOPE

The egpeirce package uses also counters and Boolean switches. Commands set their values. These commands have a global scope in the sense that (mostly) regardless of when or in which scope they are set, their value will remain unchanged until they are explicitly reset.

COMMANDS CAN HAVE multiple, though at most eight, arguments. The number of arguments a command does have is predefined and generally cannot be changed, although it may be possible to leave some of them empty. If an argument can be left empty, this is always specifically mentioned.

Some commands can accept *optional arguments*. They are enclosed, comma separated, in square brackets preceding the compulsory ones and default to a predefined value if not used.

OPTIONAL ARGUMENTS, STARRED
COMMANDS

A *starred command* is another common L^AT_EX mechanism to have alternative behaviour for a command. The package uses them too so that the command namespace does not inflate and that understandably named commands can have alternative behaviour. Type in the asterisk (*) after the command name to use the alternate versions.

Generally speaking, then, a full L^AT_EX command follows the form: \commandname[<opt. arg1>, <opt. arg2>, ...]{<arg1>}{<arg2>} ... {<arg8>}.

Some commands used in this document

A few other common \LaTeX commands are explained below as they are used extensively in the code examples.

BY DEFAULT \LaTeX will ignore multiple, consecutive empty spaces left in the source text. Also the space after a command name won't print in the output. Backslash with a space (\backslash) creates the extra empty space and backslash + comma ($\backslash,$) a smaller empty space.

Although $\backslash_$ is a font-specific length and you can create longer stretches of empty space with it ($\backslash_ \backslash_ \backslash_ \dots$), there are better options for creating long spaces. \backslash_h and \backslash_v clear horizontal and vertical empty space, respectively, of the length and height of the finally *typeset* dimensions of $\langle\! \langle x \! \rangle \! \rangle$. They are extremely handy when you must align graph-elements.

EXTRA SPACES & ALIGNMENT OF GRAPH-ELEMENTS

Notice that \backslash_h has no height and \backslash_v no length. Both match the dimensions of their argument *exactly*: \backslash_v differentiates e.g. between the height of descenders and ascenders, that is, for example between \bar{p} and \bar{d} . $\backslash\strut$ creates a vertical space equal to that between two consecutive lines.

DOUBLE BACKSLASHES $\backslash\backslash$ end the paragraph and start a new line.

LINEBREAKING

As is briefly discussed below in the short history of the language, \LaTeX has a very elegant and robust system for hyphenation and line-breaking. However, it's often desirable to manually prevent a line-break in a specific place and force \LaTeX to find another solution.

Notice that by default \LaTeX can choose to break the line even in the middle of a graph. Particularly with inline β -graphs, an unplanned linebreak will usually result in syntactically and semantically erroneous graphs.

The tilde character (\sim) creates a non-breaking space. It will disable a linebreak between two characters or monosyllabic words. If \sim is used between two polysyllabic words, \LaTeX may decide to hyphenate and break either of the words. In these cases you may use for example the $\backslash\mbox{\langle\! \langle x \! \rangle \! \rangle}$ command since it prevents a linebreak in its entire scope. A sufficiently long argument for $\backslash\mbox{\langle\! \langle x \! \rangle \! \rangle}$ will flood into the margin and beyond the page, so do use it with caution.

The linebreaking algorithm considers entire paragraphs and pages at a time. Therefore it can be difficult to estimate correctly the cascading effects a change early in a page can have on the subsequent paragraphs. Although you should enclose large inline graphs in an $\mbox{\langle\! \langle x \! \rangle \! \rangle}$ as a precaution, leave fine-tuning the hyphenation last.

Commands for drawing EGs

There are relatively few basic commands needed for drawing existential graphs. Even complex graphs can be constructed using these elements and there are often multiple ways to create similar results.

I will first cover the syntax and basic usage of the commands needed for the three systems.

BY DEFAULT and for legibility the package gives extra vertical space to graphs and an equal, smallest height for cuts. There are two ways to remove these restrictions and blend graphs better inline with text.

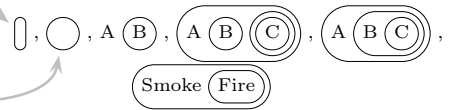
The inline environment is meant for sporadic use. It has an optional argument for linestretch that defaults to 0.5 so lineheights are halved unless you set it to another value, e.g. `\begin{inline}[0.3]... \end{inline}`. You can also declare the Boolean `\notinlinefalse` for reduced graph space all around and reset the lineheight if and when needed with the `\setstretch{<fraction>}<text>` command.

Large graphs and multiply nested cuts will still look bad inline.

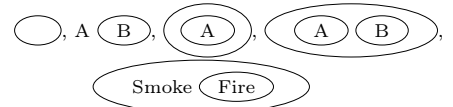
INLINE GRAPHS

The α -system and cuts

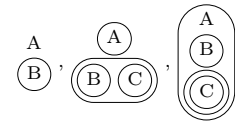
Cuts are made with the `\cut{<text>}` command. Notice that you should specify space even inside an empty cut: `\cut{ }`, `\cut{ \ }`. Cuts can be nested arbitrarily many times by simply nesting the commands: `\cut{\cut{<text>}}`.



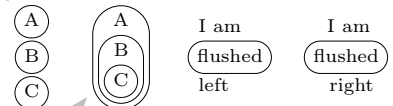
Sometimes the cuts in Peirce's manuscripts are clearly more oval shaped. To reproduce this effect, the package has a Boolean switch called `ellipsecut`. If you declare `\ellipsecuttrue`, all the cuts made with the `\cut` command will appear more ellipse-like (declaring `\ellipsecutfalse` reverts the behaviour). Be aware that the elliptical cuts do have limitations with β -graphs (see page 11) and that they are generally much harder to successfully blend inline with text.



Peirce often places elements on top of each other to clarify the graph or save space. The `\ontop{<above>\<below>}` command does this. The normal linebreak command specifies the break point.



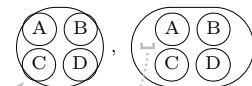
Arbitrarily many elements can be placed on top of each other. However if cuts are nested on top of each other, a new `\ontop{ }` must be declared at every level. So: `\ontop{\cut{A}\ \ \cut{B}\ \ \cut{C}}` but when nested: `\cut{\ontop{A\ \ \cut{\ontop{B\ \ \cut{C}}}}`



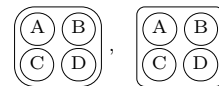
Text or graphs put ontop are horizontally centered with respect to surrounding text, and the contents of the `\ontop` command are also vertically centered. Especially in more complex graphs it is often convenient to have the contents automatically flushed left or right. The commands `\ontop_l{ }` and `\ontop_r{ }` do this.

CUTS IN PEIRCE'S MANUSCRIPTS are most often circular or elliptically shaped (though boxlike cuts exist too). The arc of a cut is defined as a fraction of its height and width and the `\cut{ }` command attempts to preserve the circularity as much as possible.

Therefore if a cut has a high and wide area, the arcs will be large too. If such a cut has cuts in it, this can easily cause them to intersect! The current version of the package does not know that this is happening and can't give you a warning. Be aware of the possibility.



In these cases you can either add space between the cuts using `_` or use the otherwise similar `\vcut{ }` and `\vvcut{ }` commands that have a more conservative arc and thus the more boxlike appearance.



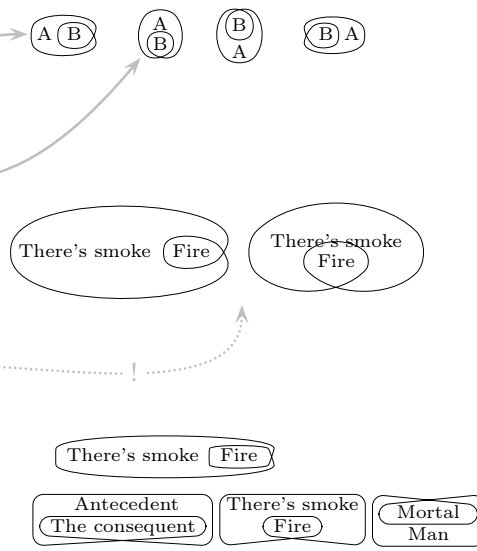
SELF-INTERSECTING CUTS *do* feature in Peirce’s system in the form of *scrolls*. The commands needed for them are unfortunately slightly more complicated than for cuts.

Scrolls can be horizontally or vertically aligned and the intersection can point up or down and left or right. None of this changes the interpretation but since they all feature in Peirce’s manuscripts, we’ll need at least four commands.

Scrolls take two arguments but you can leave either one empty. Remember to specify space for the empty argument. `\scroll{A}{B}` draws a small scroll. For the vertically aligned, use: `\vscroll{A}{B}`. These are by far the most common ones and therefore considered the basic shapes. ‘Inverse’ scrolls have their own and similarly behaving commands: `\inversescroll{}` and `\inversevscroll{}`.

Like cuts, scrolls try to match Peirce’s smooth, circular shape. If the usual scrolls have lots of text in them, they still try to preserve this shape and the results will look awful.

A ‘long’ versions of the scroll, `\longscroll{There’s smoke}{Fire}` should be used in these cases. A vertically aligned long version of scroll exists as well: `\longvscroll{Antecedent}{The consequent}`. These scrolls automatically adjust their length depending on the arguments. The ‘long’ scrolls also have ‘inverse’ versions. Following this convention, there’s e.g. the command `\longinversevscroll{}`.

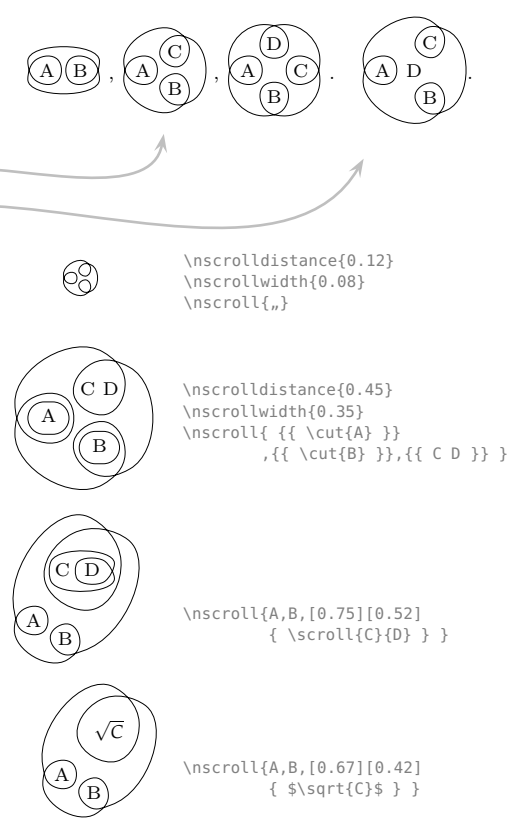


ALONGSIDE ONE-PLACED or *unary* scrolls, Peirce’s system—and his *nachlaß*—has examples of *binary*, *ternary*, and more generally *n*-ary scrolls too. The `\nscroll{}` command is used for these. Since it has to handle an arbitrary number of elements, its syntax is unusual. It accepts *one* nonempty argument and the inloops are fed as a comma-separated list: `\nscroll{A,B}`, `\nscroll{A,B,C}`, `\nscroll{A,B,C,D}`. A starred version exists and has *two* arguments `\nscroll*{A,B,C}{D}`. It places the contents of the second argument to the middle.

The command distributes the inloops evenly along the circumference of the outermost cut, but the lobes are not automatically adjusted to accommodate their contents. Commands `\nscrollwidth` and `\nscrolldistance` govern the diameters of the inloops and their distances from the center. You can also change the dimensions of any lobe(s) one at a time by adding these values, respectively, as optional arguments before the element(s).

These dimensions are reset after every `nscroll` back to their default values, contained in commands `\defaultnscrollwidth` and `\defaultnscrolldistance`. They are unitless scalars ($\mathbb{R}_{>0}^+$) and you have to figure out the proper values primarily by trial and error.

Unfortunately, if the list elements contain (1) more than one character, (2) *any* commands or (3) math mode code, they must be regrouped twice: `{{...}}` (see the code examples in the margin). This is a highly frustrating requirement arising from `\@ifnextchar` and `\pgffor`-loop interacting. **To any TeXnicians reading this:** it would be great if this regrouping could be avoided, preferably without having to change the otherwise simple syntax of the command.



By default the first inloop is placed to the leftmost side of the main cut and the rest are distributed anticlockwise. The initial alignment can be changed to an arbitrary angle by resetting a *counter* called `\nscrollangle` (the angle values follow those of the mathematical unit circle). `\defaultnscrollangle` contains the default setting.

Although the name `\nscroll` suggests that it could accept n elements such that $n \in \mathbb{N}$, this is not strictly speaking true. Firstly `\cut` already covers a `\0scroll` and although the elements themselves can be empty, the command does expect to receive a *list* of such elements. A `\1scroll` is much more efficiently covered by the different `\scroll` commands (for example, because of the need for `\longscrolls`).⁶

SCROLLS CAN BE NESTED or iterated just like cuts. With β -graphs, iterated scrolls have an unexpected side-effect: see the ‘third point’ on page 11. Slightly more complicated cases for scrolls will be introduced when I discuss further intricacies of the β -system (‘second point’ on page 11). Page 17 and onward contain general solutions for drawing rarely appearing ‘nonstandard’ scrolls.

CUTS IN PEIRCE’S manuscripts are sometimes defined by coloured regions. This is supported with a Boolean switch `\colouredcutstrue`. After this declaration every evenly-nested (or non-nested) cut will be automatically shaded. Declare `\colouredcutsfalse` to revert the behaviour. Although the Boolean can be flipped even inside a graph, *forcing* shading on *any* single cut is more easily done with the `\cutx{}` commands. See page 21 on changing the colour of the shading.

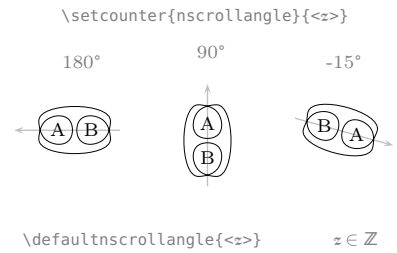
Notice that no line is drawn between the shaded cuts. Peirce emphasized that the idea of a *cut* should be taken literally. Unlike lines, cuts have no dimension which is also true of the transition between contiguous coloured and non-coloured regions.

Since scrolls are special types of cuts, also they are subject to the `\colouredcuts` Boolean. The mechanism that colours the scrolls is however quite complex and should still be considered experimental.

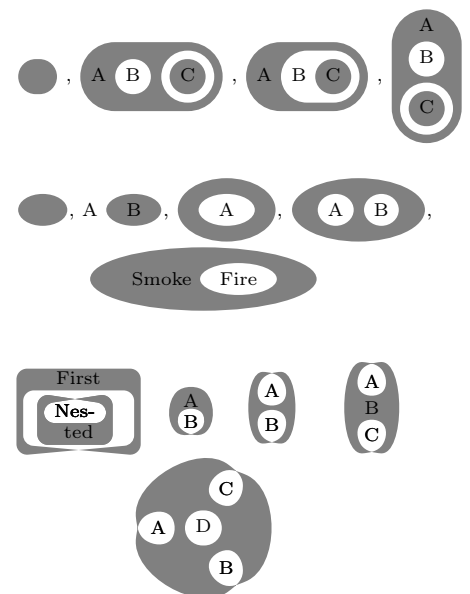
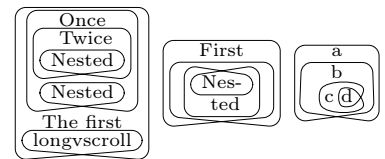
A FEW MANUSCRIPTS contain γ -graphs that employ tinctures and heraldic colouring. This, however, is an altogether different idea from shading discussed above. See page 22. Reproducing a “blot”, an emptied or blacked out inner cut of a scroll, is discussed on page 20.

So, for the α -system, there are the following basic commands:

<code>\ontop{}</code>	<code>\cut{}</code>	<code>\scroll{}</code>	<code>\longscroll{}</code>
<code>\ontopl{}</code>	<code>\vcut{}</code>	<code>\vscroll{}</code>	<code>\longvscroll{}</code>
<code>\ontopr{}</code>	<code>\vvcut{}</code>	<code>\inversescroll{}</code>	<code>\longinversescroll{}</code>
		<code>\inversevscroll{}</code>	<code>\longinversevscroll{}</code>
		<code>\cutx{}</code>	
<code>\colouredcuts(true false)</code>	<code>\vcutx{}</code>	<code>\nscroll*{}</code>	
<code>\ellipsecut(true false)</code>	<code>\vvcutx{}</code>		
<code>\notinline(true false)</code>		<code>\begin[inline] ... \end{inline}</code>	



⁶ Secondly, the current code for the command (due to the way L^AT_EX implements its counters) imposes a *hard* limit of 2^{31} , or 2 147 483 647, individual inloops.

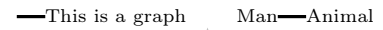


The β -system and ligatures

Drawing a line-of-identity or ligature is a two-step process.

First you must define two end points, *hooks*, to the ligature with the `\hk{}` command. You can leave the argument empty or put text in it for a *rheme*. Each hook is automatically assigned a number (by default starting from 1) following the order in which they appear in the code. Ligatures are then drawn using these numbers as reference.

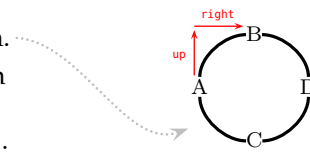
The `\li{}` command draws a simple straight line. So for example the commands `\hk{}` `\ \hk{This is a graph} \li{1}{2}` and `\hk{Man} \ \hk{Animal} \li{3}{4}` both complete a simple β -graph.



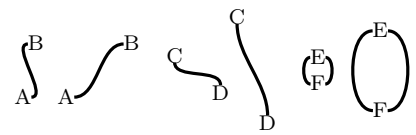
CURVED LINES are always drawn with the standard `\ncurve{}` command. Unfortunately it requires always defining—for both end points—angles at which the line meets them. Doing this would be tedious and repetitive to the extreme (and explained on page 22). Thus the package has *shorthands* for the most usual recurring types.

Consider for example a stacked graph like the one in the margin. It has two hooks at both ends (A, D) and two hooks (B, C) atop each other in the middle: `\hk{A} \ontop{\hk{B}\ \hk{C}} \hk{D}`.

The ligatures of the example graph are drawn with the commands: `\upright{1}{2}\downright{1}{3}\rightdown{2}{4}\rightup{3}{4}`. The mnemonic is that—proceeding from left to right—the ligatures first travel up or down and then to the right after which they first point right and then turn up or down.



Other constantly recurring curved ligatures that warrant a shorthand are the hopefully self-explanatory `\sligature{}`, here drawn from A to B, `\hsligature{}`, here from C to D and semicircle-shaped `\reflexivel{}` and `\reflexiver{}` commands (E, F).



Notice that the ligature's curvature, and its *exact* path on the sheet of assertion, depends also on the placement of the hooks. You may also wonder whether some shorthands are missing. Why isn't there e.g. a 'zligature'? It would be superfluous because this shape can be drawn by reversing the order of the arguments for an `\sligature`.⁷

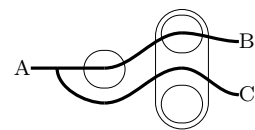
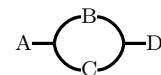


⁷ Similar considerations apply e.g. for a 'downLeft' ligature.

RECALL THAT a ligature is always drawn between a *pair* of hooks. Thus if a ligature bifurcates, the bifurcation point (or "teridentity" as Peirce called it) must be assigned its own, empty, hook.

More frustratingly, this is also true of most ligatures that have to evade—or cross—specific cuts on their path. There's no easy way to *automatically* ensure that a ligature stays inside or outside a specific area. Since the interpretation of the graph is altered if a ligature accidentally touches or traverses a cut, you must specify the proper route of the ligature by deploying 'auxiliary' empty hooks.

I haven't been able to find an easy way around this requirement (see page 26 for some ideas). For Peirce, an essential feature of ligatures is their continuity but the package handles most ligatures in a decidedly discontinuous manner. Complex (and bifurcating) ligatures must, alas, always be constructed out of discrete line segments.



LIGATURES MUST BE able to contain *gaps*. It is of course possible to make a ‘gap’ simply by adding more hooks. However, gaps are an essential part of the transformation rules of β -graphs and frequent in proofs. Therefore the shorthands accept an optional argument [-g], [g-] or [g-g] that leaves a small gap before the hook. The placement of g specifies at which end(s) the gap appears. See the examples in the margin (× marks the exact location of the hook).

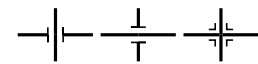
	<code>\Li[-](1){2}</code>	<code>\Li[-](2){3}</code>
	<code>\Li[-g](1){2}</code>	<code>\Li[-](2){3}</code>
	<code>\Li[-](1){2}</code>	<code>\Li[g-](2){3}</code>
	<code>\Li[-g](1){2}</code>	<code>\Li[g-](2){3}</code>
	<code>\Li[-g](1){2}</code>	<code>\Li[g-g](2){3}</code>

A SPECIAL TYPE of ‘gap’ is the *bridge*. It is needed when a graph *cannot* be drawn without two ligatures traveling across each other.

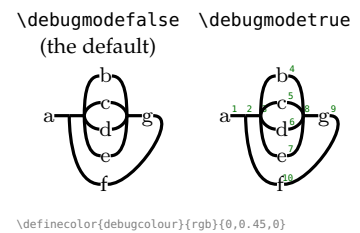
This can happen in surprisingly simple cases. The last example graph on the cover page is a very good example. Readers versed in modern graph theory will recall that somewhat more generally, simple K_5 and the complete bipartite graph $K_{3,3}$ ⁸ are both also *nonplanar*. Admittedly in usual EGS a bridge can often be avoided.

⁸ Also known as the ‘Thompson graph’ or the ‘utility graph’. It shows the impossibility of connecting two sets of three vertices, such that every vertex of the first set is connected to every vertex of the second set, without at least one edge overlapping another.

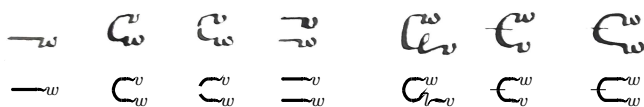
Nevertheless, this is a routing problem and it must somehow be made clear that the two ligatures don’t belong together, as it would alter the interpretation of the graph. Peirce solved this in two ways: either he left a gap and a bar at ends of the crossing ligature, or carets to denote the overlap point. The optional arguments [-b] and [-xb]—which behave otherwise similarly to a gap—draw them.



OFTEN IN MORE COMPLEX graphs the numbering and placement of hooks can become disorientating to the point that it’s hard to draw the ligatures. To forestall this, the package has a Boolean switch called `\debugmode`. When set to `true`, each hook has next to it, **by default in dark green (redefinable with ‘`debugcolour`’)**, the number that is associated with it. Compile the source with this declaration and consult the output when coding the ligatures.



IN A FEW CASES the ligatures in Peirce’s manuscripts connect directly to the rhemes with a tapering line. L^AT_EX can emulate this effect with a parameter called `variableLW` that applies to all line and curve methods available in PSTricks:



The problem is that the exact position on the glyphs where the ligature end connects to or departs from, differs not only between (most of) the letters, but is also dependent on the typeface (Computer Modern 10pt in the example above), the specific font (e.g. upright, *italic*, *slanted*, SMALLCAP) and often even on the point-size in use. For the italic letter ‘l’ in CMR10 above, commands that draw a tapered curve from the letter to the ligature or to the letter from the ligature are:

```
\newcommand{\lito} {\pscurve[variableLW,startLW=0.1pt,endLW=1.3pt](0.16,0.17)(0.09,0.16)(0.05,0.092)(-0.01,0.075) \ }
\newcommand{\liffoml}{\pscurve[variableLW,startLW=0.1pt,endLW=1.3pt](0.00,0.03)(0.023,0.065)(0.07,0.08)(0.11,0.075) \ }
```

For the reasons mentioned above, the package does not contain similar commands for all the letters in different typefaces and fonts.

A logical scheme for resetting the counter is to set it to 0 at the end of each graph (so that the first hook of the next graph is always number 1). *This* is not done automatically because there are reasonable uses for different conventions.

It is indeed possible to have multiple similarly numbered hooks on the same page. When you draw the ligature, \LaTeX simply references the last instance of the hook number it can find.

For the β -system, there are thus the commands:

<code>\hk{}</code>	<code>\li{ }{ }</code>	
<code>\setcounter{rheme}{ }</code>	<code>\upright{ }{ }</code>	<code>\scroll*{ }{ }{ }{ }</code>
	<code>\downright{ }{ }</code>	<code>\vscroll*{ }{ }{ }{ }</code>
<code>\debugmode(true false)</code>	<code>\rightdown{ }{ }</code>	<code>\inversescroll*{ }{ }{ }{ }</code>
	<code>\rightup{ }{ }</code>	<code>\inversevscroll*{ }{ }{ }{ }</code>
	<code>\sligature{ }{ }</code>	<code>\longscroll*{ }{ }{ }{ }</code>
	<code>\hsligature{ }{ }</code>	<code>\longvscroll*{ }{ }{ }{ }</code>
	<code>\reflexivel{ }{ }</code>	<code>\longinversescroll*{ }{ }{ }{ }</code>
	<code>\reflexiver{ }{ }</code>	<code>\longinversevscroll*{ }{ }{ }{ }</code>

The γ -system ...

... was not fully developed by Peirce. Therefore also this section will offer only an extremely cursory examination of the subject.

There are relatively few reliably and faithfully recurring γ -graphs in Peirce's manuscripts, although there are copious amounts of separate ideas about the γ -system.

ONE OF THE FEW RECURRING types of γ -graph present in the manuscripts is the dashed or 'broken' cut. The 'g' (for γ) series of cut-commands (`\gcut{ \ \ }`, `\gvcut{ \ \ }` and `\gvvcut{ \ \ }`) which otherwise behave similar to the normal `\cut`s, is assigned to draw them. In the manuscripts, sometimes the 'broken' cuts have clearly different types of dashes or dots producing the perimeter. Consult page 22 on how to faithfully reproduce the different types of dashes or dots.



Useful shorthand commands for representing Peirce's modal-logic part of γ -graphs are the commands `\dbcut{ }`: `\pcut{ }`: and `\ncut{ }`: as these define the modalities of necessity (`\ncut{ }`), possibility (`\pcut{ }`) and the double broken cuts, with single separate commands.

Also a special type of hook, `\shk{ }`, (example in the margin) appears in multiple places in the manuscripts. The command is only typographically different from a normal hook (that is, the hook includes the 'hat' or "envelope" as Peirce called it).



MOST IDEAS in γ -graphs seem to concern the colouration of cuts or ligatures. Doing this is discussed in the “Handy PostScript commands” section (page 22).

ANOTHER HIGHLY INTERESTING idea involves the three-dimensionality of graphs and the sheet of assertion. Peirce provided few actual examples of such graphs, so specific ideas of representing this are up for grabs. As food for thought there does exist a package, `pst-3d`, that handles actual transformations for projective 3D. It could easily be employed here too. Please consult *The L^AT_EX Graphics Companion* (Goossens, 2008, pp. 388–410).

Γ -graphs thus have only the following *separate* commands, though L^AT_EX is certainly able to reproduce the different ideas present in Peirce’s manuscripts:

```
\gcut{}    \dbcut{}  \shk{}
\gvcut{}   \pcut{}
\gvvcut{}  \ncut{}
```

FINALLY, there’s a particular command called `\everygraphhook` that gets executed inside every graph-element and every `\hk{}` command.

By default, the command does *nothing* and it’s definition is:

```
\newcommand{\everygraphhook}[1]{#1}
```

The point is that in the package code, this command or ‘hook’ is automatically included in every graph-element and you can redefine it yourself to whatever effect you’d like. If, for example, you would like to have text inside every graph automatically *italicized*, you can simply redefine the `\everygraphhook` command rather than manually add `{\it ...}` or `\textit{ ... }` to every single graph separately.

For *simple* transformations you can even use the T_EX primitive `\let`. For example, to italicize all graphs `\let\everygraphhook\it` suffices. More complex ones require `\renewcommand` available in L^AT_EX.

Examples of graphs

Next I’ll consider some example graphs from Peirce’s manuscripts and their solutions with the package. The examples perhaps better convey the look and feel of the graphs made by the package.

Although we are yet to come across a graph that the package would have been *unable* to draw, some require considerable rumination to work out. Even complex-looking graphs are solvable with some imagination and a good working knowledge of standard L^AT_EX commands.

Simple cuts and ligatures

The first example contains typical, simple inline α -graphs. The picture is taken from ms 430. Below it is a rendition of the excerpt by the package. Only the familiar and simple cut command is used, but a few things are noteworthy in this simple example too. Here is a good example of the Boolean `\notininlinefalse` which is declared for reduced space inside the cuts

Notice that the cuts produced by the package do not and cannot immediately and exactly resemble those that Peirce drew and that there is obviously some idealization going on. The other noteworthy thing is that when there are lots of cuts right next to each other, they can easily create a distracting moiré-like pattern that is starting to show in the thrice-cutted graph in the example. Adding an empty space between the cuts can lessen this effect. This space isn't added automatically since adding it is largely a matter of taste. Changing the definition of a cut to automatically and always include such a space would be trivial though.

c in the ovals and then inserting \textcircled{n} into the outer of these. In the second sense, the propositions are,

$T(c \overline{r} \textcircled{n})$ which is reducible into $T(c \textcircled{n}) T(\overline{r} \textcircled{n})$
 $T(c \textcircled{\textcircled{r} \textcircled{n}})$ which is reducible to $T(c) T(\overline{r} \textcircled{n})$
 from which the other is at once deducible by inserting \textcircled{n} in the

c in the ovals and then inserting \textcircled{n} into the outer of these. In the second sense, the propositions are

$T(\overline{c} \overline{r} \textcircled{n})$ which is reducible to $T(\overline{c} \textcircled{n}) T(\overline{r} \textcircled{n})$
 $T(\overline{c} \textcircled{\textcircled{r} \textcircled{n}})$ which is reducible to $T(\overline{c}) T(\overline{r} \textcircled{n})$
 from which the other is at once deducible by inserting \textcircled{n} in the



$\backslash\text{cut}\{T \ \backslash\text{cut}\{c \ \backslash\text{cut}\{\text{cut}\{r\} \ \backslash\text{cut}\{n\}\}\}\},$
 $\backslash\text{cut}\{T \ \backslash\text{cut}\{c \ \backslash\text{cut}\{\ \backslash\text{cut}\{r\} \ \backslash\text{cut}\{n\} \ \}\}\},$
 $\backslash\text{cut}\{T \ \backslash\text{cut}\{c \ \backslash\text{cut}\{\ \backslash\text{cut}\{r\} \ \backslash\text{cut}\{n\} \ \}\ \}\}$

The second example is of slightly nonstandard inline ligatures also from ms 430. Apologies for the bad image quality. Also the text in the output example is forced to follow the text flow in the picture, which makes it easier to compare to the picture but look strange.

All graphs are drawn with the inline-environment: changing lineheights is easy with its optional argument. Recall that L^AT_EX cannot automatically route ligatures around obstacles but needs hooks. Therefore lots of ontop-constructions are being used.

Consider the second graph in the example where B is encircled. Because the rheme just happens to be a single letter long, an alternative solution using two hooks and reflexive ligatures would suffice.

A smoother curve and a more general solution is defined with four hooks around the B-rheme. Because there are three hooks ontop and the graph is inline, `\begin{inline}[0.33]` is declared—the 0.33 makes lineheights 1/3 of the original. Empty hooks do not have any intrinsic height or width and this can make them look awkward when ontop. Therefore hooks number 3 and 5 have a `\vphantom{a}` in them. In this example this isn't strictly speaking necessary, but

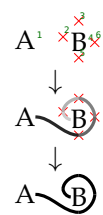
indifferent. Thus A—B and A—B will be the same; but A—B will be different. So A—B will be different from $\begin{matrix} A \\ B \end{matrix}$; because different sides of the letters are joined; but A—B and $\begin{matrix} A \\ B \end{matrix}$ will be

indifferent. Thus A—B and A—B will be the same; but A—B will be different. So A—B will be different from $\begin{matrix} A \\ B \end{matrix}$; because different sides of the letters are joined, but A—B and $\begin{matrix} A \\ B \end{matrix}$ will be



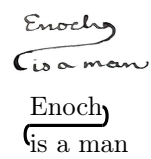
if it had cuts, you should give the empty hooks an explicit height. Finally, the second hook is slightly elevated from the baseline with the standard \LaTeX command $\text{\raisebox{<{height}>}{<{text}>}}$. Without this the final downright curve would look strange:

```
\begin{inline}[0.33]\hk{A} \ \ \ ,\raisebox{2pt}{\hk{}} \ontop{
\hk{\vphantom{a}}\hk{B}\ \ \ \hk{\vphantom{a}} \ } \hk{} \end{inline}
The ligature is drawn with:
\sligature{1}{5}\rightup{5}{6}\rightdown{3}{6}\upright{2}{3}
\downright{2}{4}. Finally \setcounter{rheme}{0} is declared.
```



THE EXAMPLE GRAPH in the margin from MS 493 illustrates yet another common point with ligatures. The ligature needs to be routed with two additional hooks between the rhemes. You must provide a suitable amount of empty space, most easily with $\text{\hphantom{}}$. \ontop takes care of the alignment. \linestretch is set to 0.5.

```
\ontopl{\hk{Enoch}\ \ \ \hk{\hphantom{Enoch}}\hk{}} \ \ \ \hk{is a man}
\reflexiver{1}{3}\li{2}{3}\reflexivel{2}{4}
```

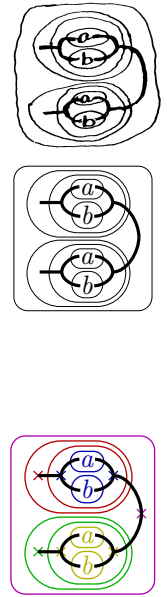


THE NEXT GRAPH is from MS 430. It is a stacked graph that has the boxlike outer cut and two identical 'subgraphs' inside it.

This example highlights a difficulty: the code is necessarily linear, but the graphs are described by a two-dimensional field which in this case is in vertical alignment. A helpful trick is to arrange the code in logical 'blocks' with the reserved character $\%$ (see below). $\%$ is used for comments in the code and when compiling the source, \LaTeX simply discards everything that follows it—including the linebreak.

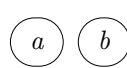
Below the code is also indented and colour-coded in a hopefully helpful way. Notice that you should never use indentations in your actual code. Although \LaTeX does ignore *multiple* consecutive spaces, it will interpret them as a space which will appear in the output.

```
\vvcut{ \ontop{ %
  \cut{ \hk{ } \cut{ %
    \hk{ } \ontop{\cut{ \hk{a} }} \ \ \ \cut{ \hk{b} }} \hk{ } %
  }} %
\ \ %
  \cut{ \hk{ } \cut{ %
    \hk{ } \ontop{\cut{ \hk{a} }} \ \ \ \cut{ \hk{b} }} \hk{ } %
  }} %
} \hk{ } }
```



The rightmost ligature goes through a single hook. In this case, a reflexiver ligature would have sufficed since it's not at risk of crossing the cut. In the original, this ligature appears perhaps less curved: the lone hook could have been replaced with two \ontop . &c.

The height difference in the cuts (a) and (b) is caused by \notinlinefalse or by the \inline -environment. Removing these forces all cuts and hooks have an equal minimal—though quite large—height. Alternatively you could add a $\text{\vphantom{b}}$ to the first cut: (a) (b).

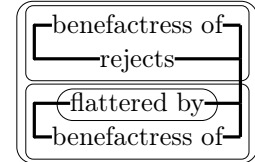
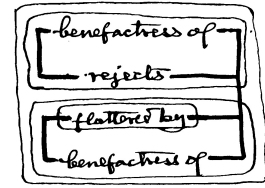


THERE ARE OFTEN multiple ways to create similar results as the next example from MS 430 shows. The difficulty here is that the `lis` meet at right angles and thus the hooks must somehow be aligned.

One could simply place four layers of three hooks on top of each other. However, because the rhemes in the middle hooks differ in length, you'd have to estimate the amount of empty space needed to align the terminating hooks between each layer.

An alternative, slightly more complex solution is offered below. The hooks are arranged in *pairs* that are put on top, so there's no need to align them vertically by hand. Putting the two larger cuts on top automatically aligns the rightmost hooks:

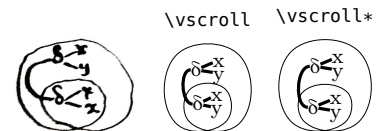
```
\vvcut{\,\ontopr{%
  \vvcut{\,\ontop{\hk{}}\hk{}} %
    \ \ontop{\hk{benefactress of}\hk{rejects}} \ %
    \ontop{\hk{}}\hk{}}\,\}\%
\vvcut{\,\ontop{\hk{}}\hk{}} %
  \ \ontop{\cut{ \hk{flattered by} }\hk{benefactress of}} \ %
  \ontop{\hk{}}\hk{}}\,\,}%
}\,}
```



The ligatures are drawn in an obvious way. However, if you look closely, the ends of the rightmost ligatures have a nasty dent. This is because by default all ligatures end with a straight edge. The leftmost `lis` are drawn with an alternative argument [`c-`], which creates a c-shaped semicircle to the end that removes the dent.

LASTLY AN EXAMPLE of a scroll is included before I venture into the more difficult cases. The picture is from MS 277. This graph is examined in more detail in the next section. `\vscroll` could almost handle it (the inner cut touches the descender of the `y`-rheme). `\vscroll*` does a better job, because empty space in the additional arguments gives more precise control over the cut.

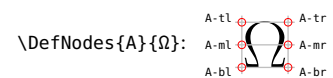
Luckily the long scrolls usually automatically resemble their drawn counterparts. Their starred versions are needed only if you must place a hook on the inner cut.



Nonstandard scrolls and complex graphs

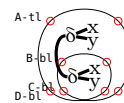
The behaviour of scrolls may seem strange. Explaining the inner workings perhaps helps. Firstly, there is a mechanism that enables the four corners and middle points of a text area to be defined as referenceable coordinates. The five arguments of the starred scrolls correspond to five such areas that are put on top. Finally, a curve drawn through the coordinates in the right order forms the cut.

The `\DefNodes{<ref>}{<text>}` command does the referencing (the huge Ω in the margin is an example). `<ref>` identifies the text area, and automatically identifies the six points as: `<ref>-tl`, `<ref>-tr`, `<ref>-ml`, etc. If `<text>` has no length, the left and right sides coalesce. If it has no height, the top, middle and bottom points coalesce. If it's empty, all the coordinates collapse to the same point.



Recall the `vscroll*` example from the previous page. The five arguments correspond to the five layers ontop each other.

```
\vscroll*{ \ \ \ \ \ \ }%           A
  { \hk{\delta} \ontop{\hk{x}\ \ \hk{y}} }%
  { \ \ \ \ \ }%                     B
  { \hk{\delta} \ontop{\hk{x}\ \ \hk{y}} }%   C
  { \ \ \ \ \ \ \ \ }                 D
```



scrolls automatically assign coordinates with `DefNodes` to four of the layers, with `<refs>` A–D (and a reference for nestedness, see below). The second layer is exempted since it automatically stays within the cut. Layers A, B and D—that is—arguments 1, 3 and 5 needn't any height but can be assigned some to alter the appearance.

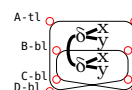
In a simple connect-the-dots kind of way, the cut is drawn with:

```
\psccurve[curvature=1 0 0](A-tl)(A-tr)(D-br)(C-bl)(B-bl)(B-br)(C-br)(D-bl)
```

Internally, the nodes also have a reference that tracks the level of nestedness so that the scrolls can be iterated easily. This reference is contained in `\egatn`. If you want to manipulate scrolls created by the package, remember to add this command to the coordinate points mentioned above: `(A-\egatn-tr)(D\egatn-br) ...` See e.g. page 20.

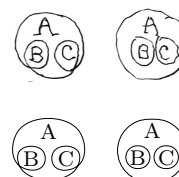
`longvscrolls` are drawn with a `pspolygon` instead of the `psccurve` for the more boxlike cut. The *non-starred* versions of `longvscrolls` also automate the drawing somewhat. They first check which line is longer, the premiss (layer 2 = the first argument) or the consequent (layer 4 = the second argument) and then make the first and last layers 6pt longer than the longest argument. This automation gives the scroll straight edges and explains why the starred versions aren't usually needed.

```
\longvscroll{ \hk{\delta} \ontop{\hk{x}\ \ \hk{y}} }%
  { \hk{\delta} \ontop{\hk{x}\ \ \hk{y}} } }
```



AFTER THIS INTRODUCTION you are now armed to tackle all kinds of nonstandard scrolls.

The first example is from `ms670` of a twice self-intersecting cut. Taking a cue from the explanations above, it's not difficult to figure out how the graph might have been drawn. Although as usual, different and more complex solutions would have been equally possible. Also, `\nscroll*{B,C}{A}` would draw a semantically identical graph, but look a bit different. In this example `Defining Nodes` for the three letters suffices. Space inside them allows for some fine-tuning. The cut is drawn through the most convenient points.



In usual scrolls the point of intersection has no coordinate, which makes it look very pronounced. In this example doing it would have required additional noded areas. Instead the `curvature`-parameter is altered to make the intersection points more distinct:

```
\ontop{\DefNodes{A}{ \ \ A \ \ } \ \ \DefNodes{B}{ B } \DefNodes{C}{ C }}
\psccurve[curvature=1 0.5 0.5](A-mr)(C-br)(C-bl)(C-tl)(C-tr)(C-br)
(B-bl)(B-tl)(B-tr)(B-br)(B-bl)(A-ml)
```



A SLIGHTLY DIFFERENT looking example of a twice self-intersecting cut from MS488 is in the margin. The graph may appear simple when drawn with a pencil, but writing its code does require some forethought.

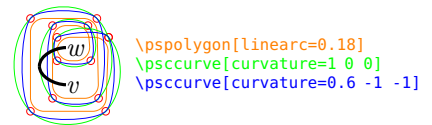


In particular, keeping the leftmost portion of the ligature in the singly cut area and having the rightmost cut protrude through the two cuts are not obviously soluble. Defining the areas and aligning them is a simple matter of providing sufficient empty space. `Linestretch` is set to 0.5, `ontopl` takes care of initial alignment:

```
\ontopl{%
\DefNodes{A}{ \ \ \ \ \ \ \ }\\
 \ \ \ \DefNodes{B}{ \ \ \ \ \vphantom{a}}\\
 \ \ \ \DefNodes{C}{ \hk{w}\vphantom{pl} } \ \ \DefNodes{D}{\vphantom{pl}}\\
 \ \hk{}\strut\\
 \ \ \ \DefNodes{E}{ \ \hk{v} \ \strut}\\
\DefNodes{F}{ \ \ \ \ \ \ \ \ }%
}
```

Notice the `\struts` and different `\vphantoms`. Keeping the empty hook in the right place, areas A and F are not indented, whereas the rest have empty space before their introduction. Area D ensures that the cut can protrude horizontally out of the graph.

The cut is defined by the sequence: (A-bl)(A-br)(E-br)(E-bl)(B-tl)(B-tr)(C-br)(C-bl)(C-tl)(D-tr)(F-tr)(F-tl). Notice that this is a general solution and that the cut can be actually drawn with different lines or curves. In the margin the reference points are marked and the different lines and curves are superimposed. The green curve is already a bit too steep, since the ligature would no longer reach the singly cut area. Changes in the placement of areas A, F and B, E could take care of this.

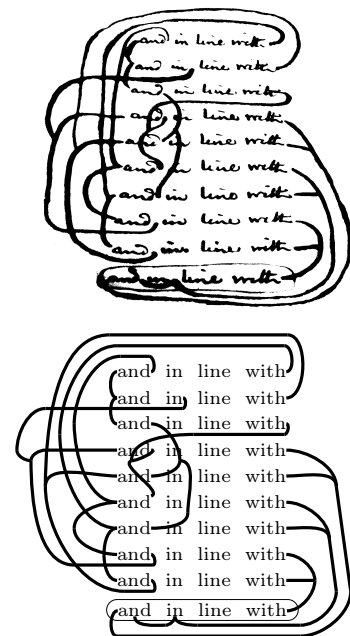


THE NEXT EXAMPLE is of a very complex graph from MS493.

Defining and placing each hook in such a complex graph separately would be a stupendous task. Instead, a far more effortless strategy is to take advantage of the repetitive structures and effectively create a matrix of hooks.

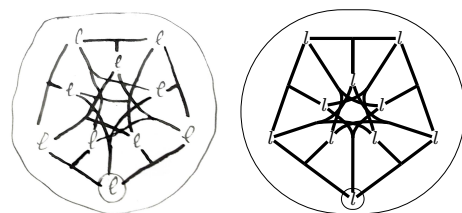
A cursory glance of the graph suggests—taking into account all the bifurcations—that no single line needs more than about fourteen hooks. Since some ligatures run or bifurcate under the text lines, it is sensible to double the number of lines. Every other line needs seven empty hooks, then the words ‘and in line with’ in separate hooks, and five empty hooks still. In every other line, the words should be defined as `\vphantoms`, totaling a matrix of hooks 14×26 .

The number of hooks per line, 14, makes it difficult to calculate their relative positions in the final matrix. It makes sense to declare e.g. `\setcounter{rheme}{100}` at the end of the first line, `{200}` at the end of the second and so on. Thus the relative position of a hook and the place of any bifurcation can be figured out easily. Drawing the ligatures for the graph this way isn’t trivial, but it is a lot easier than placing each hook individually.



PsTricks AND THUS THE `egpeirce`-package allows one to define points on the plane with vectors in a polar coordinate system. By placing hooks in such points, even graphs like the one in the margin are manageable. Because these kinds of graphs are rare, this method isn't described in detail here. Please consult e.g. *The L^AT_EX graphics companion* for more details.

Alternatively, the matrix method described previously would work almost equally well in these situations.



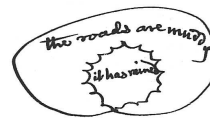
Ugly hacks

In Peirce's graph-system, there are unfortunately a few cases that still escape an elegant solution. Drawing lines on paper affords a lot of freedom. Describing the graphs in code makes them look more uniform and makes their manipulation much easier. However, this method is somewhat more limited than hand-drawing and must at times compromise. Below are two examples of such compromises.

THE EXAMPLE in the margin is a γ -graph, a scroll, that is wavy in the inner cut. Perhaps surprisingly, this is very difficult to solve.

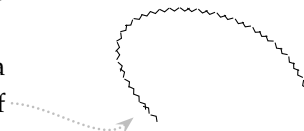
PostScript does have a mechanism for drawing zigzag lines, but the difficulty here is that the cut is also *curved*. The zigzag lines can be bent on the arc of a circle, but the code does this in an idiosyncratic way that's hard to generalize to an arbitrary curve. There is no algebraic solution to calculate a *general* wavy curve that I would be aware of. PostScript is an extremely powerful programming language and it can even solve (certain types of) differential equations automatically. I do have a nagging feeling that I must have embarrassingly overlooked something really obvious.

Of course—for our purposes—the zigzagging perimeter wouldn't have to form an actual *continuous* line. Merely a visually convincing semblance would suffice. And this is exactly what's done here. For now, the package must resort to a crude approximation.



I use one of the methods that *do* allow us to put zigzagging elements on an arbitrary curve. There is, however, a general problem in this approach with the discrete elements that form the ersatz zigzag-line. They consist of a 'zig' (\swarrow), a 'zag' (\searrow) or a 'zigzag' (\wedge) repeated one after another. Thus the line is—for lack of a better word—*quantized* on the zigs or zags (& this is one of the reasons why it's so difficult to come up with a simple algebraic solution). With this method one has to always guess or estimate the number of elements needed.

One possibility would be to create a new zigzag arrow and fill a line with it: $\sim\sim\sim\sim$. This method applies also to Bézier curves of arbitrary curvature. PostScript has a method of computing tangents to a curve at different specified points. The tangents could be used to compute control points for a Bézier curve approximating the path of the zigzag segment. This, however, is unnecessarily complex and would involve *also* guesstimating the beginning and end points.



The mechanism employed for now uses a command that makes *text* follow a curved path. After the normal scroll has been drawn, an additional line is drawn through the inner part of the cut. This line has some text in it that draws the ersatz zigzag line. Here, the zigzags are repetitions of the caret (^) whose background is filled to hide the line, that are lowered and kerned so that they can be made to form a continuous-looking line:..... . Since this is a complex operation, there's a command (\vv) that does precisely that, one caret at a time.

The command for the zigzag curve in the longscroll is:

```
\pstextpath[c](0,0){\psline[linearc=.18,linestyle=none]
(A\egatn-tr)(D\egatn-br)(C\egatn-bl)(B\egatn-ml)(B\egatn-mr)
(C\egatn-br)(D\egatn-bl)(A\egatn-tl)}{\vv\vv\vv\vv\vv\vv ... }
```

For the normal scroll, replace the \psline with a \pscurve[curvature=1 0 0,arcsep=10pt,linestyle=none].

This places the carets onto the line defined by the usual scroll sequence. Only the inner cut should be zigzagged. Since the intersection point doesn't have a referenceable coordinate, the carets are made to emanate from the middle of the line (with the [c] option). By drawing just enough carets, they will stop at the intersection.

This is an ugly hack: it does the trick, albeit not very elegantly.

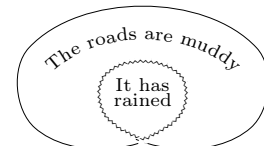
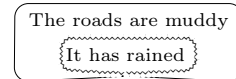
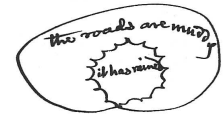
PEIRCE'S MANUSCRIPTS INCLUDE scrolls and cuts that have an emptied or blackened inner cut ("blot" as Peirce called it). This expresses the *pseudograph* and is thus entirely and fundamentally different idea from shaded cuts made with \colouredcuts (page 8).

There are two special fillrules in PostScript, *eofill* and *oefill* that fill evenly or oddly self-intersecting areas. Introducing these through \psset{} allows you to fill the inner cut of any scroll.

Notice that any content in the singly cut area is painted over in this method and would have to be reinserted through other means.

An alternative way to fill the inner cut of a scroll exists that does not fill the singly cut area. Here you specify a *solid* fillstyle and make it traverse *only* the inner cut. This is a rather hacky way to achieve the desired effect though, since you have to figure out the intersection point manually by trial and error.

Recall that the points \DefNodes creates are internally simply references to a pair of coordinate points. You can try to find the intersection using raw coordinates. In the first scroll below, this happens to be at -0.26, -0.32, so \psccurve[curvature=1 0 0,linewidth=.2 pt,fillstyle=solid,fillcolor=black](-0.26,-0.32)(C\egatn-bl)(B\egatn-ml)(B\egatn-mr)(C\egatn-br)(-0.26,-0.32) fills the cut:



```
\vscroll{A}{B}
```



```
{\psset{fillstyle=eo fill,fillcolor=black}
\vscroll{A}{B}}
```



```
{\psset{fillstyle=oe fill,fillcolor=black}
\vscroll{A}{B}}
```



Some difficult cases and their solutions

Next I will discuss a few recurring *general types* of problematic cases that arise when transcribing graphs from manuscripts. Some of these problems can be solved by changing mutable parameters of the package. Others require slightly more involved methods.

More specific cases of difficult graphs and their solutions are dealt with in the examples.

Mutable parameters of the package

The `egpeirce`-package includes many predefined parameters and dimensions that can be reset if needed. Change parameters with `\renewcommand{\<parametername>}{<value>}`. Dimensions are reset by `\setlength{\<dimensionname>}{<value>}`. You must specify all dimensions with a *unit*. Parameters can be integers, fractions or strings.

These changes have a global scope and must be explicitly reset even to get the default values back. You can of course change them for only a small period, even inside or in the middle of a graph.

Below, the initial default value of the parameter or dimension is written in the margin for easy and fast reference.

IN MOST OF Peirce's manuscripts, the cuts are drawn with a very fine black line (with a pointed nib or the narrow side) and the ligatures with a distinctly heavier pen or wider nib. This is emulated by the package. Dimensions called `cutwidth` and `ligaturewidth` are in charge of the width of the cut and ligature.

The colour of a coloured cut is defined with a parameter called `cutxfillcolour`. As was mentioned earlier, when cuts are delimited with a coloured region by declaring `\colouredcutstrue`, there is no line drawn to distinguish them. Sometimes, especially in γ -graphs there are however heavy lines around a coloured region to specify a cut. To ease the use of this, you can increase the value of the `cutxwidth`-dimension and set the `cutxcolour` parameter to a suitable colour.

It is also possible to fill a cut with a pattern instead of a solid colour. This is defined with the `xfillstyle` parameter. See the next section for more details.

Sometimes Peirce drew the cuts with a dark blue ink. Ligatures are also drawn in colours other than black, at least bright red and brown ones exist too. The colour of a cut is defined with a parameter called `cutcolour` and the colour of ligatures with `licolour`.

Finally, all vertical scrolls have reduced lineheight. Since all vertical scrolls have five layers, a logical choice would be `0.2`. For aesthetic reasons the default is slightly higher. Very much smaller values will look bad and can create problems with the arcs.

```
\setlength{\cutwidth}{0.2pt}
\setlength{\ligaturewidth}{1.2pt}

\renewcommand{\cutxfillcolour}{gray}

\setlength{\cutxwidth}{0.01pt}
\renewcommand{\cutxcolour}{white}

\renewcommand{\xfillstyle}{solid}

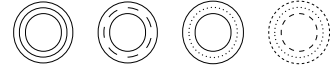
\renewcommand{\cutcolour}{black}
\renewcommand{\licolour}{black}

\renewcommand{\scrollstretch}{0.3}
```

Some handy PostScript commands

Some γ -graphs have cuts whose lines are not solid or whose areas are coloured. Since these cases are rare and not very consistent in appearance, there isn't a single parameter that governs this behaviour.

Whenever you want to draw a nonsolid cut, you must first declare `\psset{linestyle=dashed,dash=<x>pt}` before the cut. The value of `<x>` determines the length of the dashes. Changing the `linestyle` parameter to `dotted` draws dots instead of dashes.



Declare `\psset{linestyle=solid}` to get solid lines back in the middle of a graph. Values of `cutwidth` and `cutcolour` hold for the dashed or dotted cuts also.

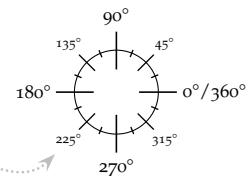
To draw coloured cuts you just need to change the `cutxfillcolour` parameter to a different colour and remember to draw all the cuts with the `\cutx{}` command. To fill a cut with a pattern, change the `xfillstyle`-parameter to a suitable option and give a value to options of `hatchwidth`, `hatchsep` and `hatchangle` with `\psset{}`. See the `PSTricks`-documentation for more options and examples.



Notice that unlike the parameters discussed in the previous section, `\psset{}` is sensitive to scoping, so you can also limit all the effects by enclosing it in curly brackets.

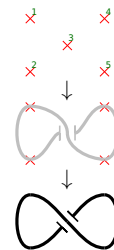
AS WAS MENTIONED EARLIER, PostScript does handle drawing arbitrarily curved lines but requires that the end angles be always defined. The `\nccurve[]{ }{ }` command does this.

`nccurve` accepts line properties as *optional arguments*, so give them comma separated inside the square brackets. Specify `linewidth=\ligaturewidth` and `linecolor=\licolour` so that changes in these parameters also effect these ligatures. The entry and exit angles are defined as `angleA=<degrees>` and `angleB=<degrees>`. Angle directions follow those of the normal mathematical unit circle. That is, to the 'right' it is 0° or 360° , for 'up' it's 90° and so on. The destination hook numbers are given in the usual way.



One situation where you may have to use `nccurve` is if the lines of bridges do not meet at right angles. The example ligature in the margin exemplifies this. The graph has five hooks. Using the shorthands makes the crossing looks strange, since they force it to be perpendicular. `nccurve` enables the crossing to be effectively rotated by 45° . The bridged parts of the example are drawn with the commands:

```
\nccurve[linewidth=\ligaturewidth,linecolor=\licolour,
  angleA=0,angleB=225]{-b}{2}{3}
\nccurve[linewidth=\ligaturewidth,linecolor=\licolour,
  angleA=45,angleB=180]{b-}{3}{4}
```



The entry and exit angles (225 and 45) define a smooth curve through the center hook. Notice that when you use `nccurve`, the bridge (or the gap) declaration must be in curly brackets instead of square ones.

Peirce's logical symbols

The package includes commands that print symbols Peirce developed for his logical system. Peirce was fastidious about the details and appearance of these symbols and signs, and every effort has been made to make the glyphs appear as close as possible to Peirce's descriptions and examples. When needed, alternative variants of symbols are also provided.

These symbols are not included in the package as new fonts but rather as vectorized pictures. This substantially decreases work needed to define them and also means that you don't need to separately install a new typeface to use them. All the symbols are contained within the package.

This technique does have two setbacks, however.

First, a separate mechanism for scaling the symbols had to be created. The symbols respond to the sizing commands (`\tiny ... \Huge`) and they also scale whenever your `.cls` file tells \LaTeX to change font sizes (e.g. inside footnotes and headers). If the symbols appear to be *overall* too small or too large compared to the typeface you use, you can change their size relative to it by redefining a command called `commoncoefficient`. Internally, the scaling mechanism tracks the `\f@size` macro, which should be a reliable source for size information. If it nevertheless causes problems, you can turn the scaling off by setting the Boolean flag `\scaledsymbols` to `false`. This won't effect the `commoncoefficient` command, so you can use it to scale the symbols independently.

Secondly, you *might* get error messages when you use the symbol commands e.g. in places where \LaTeX needs to use external files, such as in the sectioning commands (which write to the `.toc` file) and indexing commands (that use the `.idx` files). If this happens, you can easily solve these errors simply by preceding the symbol commands with the standard \LaTeX command `\protect`. This is fundamentally due to the symbol commands being 'fragile' and \LaTeX forcing their expansion too early. The concepts of command expansion, execution, and 'fragility' are too complex and low-level to be dealt with here. Be aware that early expansion—whenever it happens—will invariably result in copious amounts of errors grave enough to stop compilation. Use e.g. the `\protect` or `\expandafter` commands when needed.

Symbols that need to—at least in principle—interact with standard mathematical compositioning and typesetting commands available in \LaTeX (`\hat{}`, `\bar{}`, `\overline{}`, etc.), do play nice with them.

NEXT, LISTS OF the symbol commands and symbols are presented. The lists do not give the definitions, semantics or explanations for the symbols. References to manuscripts are provided to that end. As was mentioned in the beginning, this documentation tacitly assumes that the reader is familiar with existential graphs and Peirce's logic.

```
\renewcommand{
\commoncoefficient}{<r>}    r ∈ ℝ>0+
```

As the name suggests, it is a scaling coefficient common for all the sizes. The default is, obviously, 1.

```
\part{Part text
\protect\<symbolname>}
```

```
\section{Section text
\protect\<symbolname>}
```

```
\index{\protect\<symbolname>}
```

$$\underbrace{\check{\exists} \ \check{\exists} \ A \ \check{\times} \ B \ \rightleftharpoons \ \widehat{a\psi b} \ a \ \check{\leftarrow} \ b}_{a \ \Psi \ b \ \Psi \ c \ \cdots \ k \ \Psi \ l \ \Psi \ m} \prod_{\Psi \in i}^{\infty}$$

Some of these references are unique, but most symbols (and even their definitions) exist in multiple manuscripts. Further notes about some symbols are presented in footnotes.

FIRST, A LIST of simple symbol commands for the linear notation:

Command name	Symbol	From manuscript
<code>\aggregate</code>		MS 293
<code>\varaggregate</code> ⁹		
<code>\Paries</code> ¹⁰		MS 530
<code>\dragonhead</code>		MS 501
<code>\reversedragonhead</code>		MS 501
<code>\flatinfty</code>		MS 530
<code>\fsymbol</code>		MS L 224
<code>\implicates</code>		MS 430
<code>\cursiveimplicates</code> ¹¹		MS L 387
<code>\varinclusion</code>		MS 530
<code>\Ppropto</code>		MS 530
<code>\Pinversepropto</code>		MS 530
<code>\varwedge</code>		MS 530
<code>\weirdone</code>		MS 530
<code>\weirdtwo</code>		MS 530
<code>\weirdthree</code>		MS 530
<code>\weirdfour</code>		MS 530
<code>\napierianbase</code> ¹²		
<code>\Pratiocircdia</code> ¹²		

⁹ This variant exists *only* for use in the `\agoverline{}` commands below, and does *not* appear in Peirce's manuscripts as such.

¹⁰ The capital 'P' in the command names denotes Peirce's variant of an already existing L^AT_EX command.

This is Peirce's preferred version for the constellation symbol of Aries.

¹¹ This is merely a typographical variant of the `\implicates` command. Peirce's description for it (from a letter to Paul Carus in Oct. 1898) was: "The character [...] ought to have a somewhat Chinese effect. I have drawn the longer line pointed at both ends. But I don't know but it would look better blunt at the left. If so cut that edge must not be vertical but slanting", and a hand-drawn sketch of this variant is also provided.

In almost all manuscripts though, the symbol resembles more the 'non-cursive' variant.

¹² These symbols are Peirce's own modifications on his father's notation, for the mathematical constants e and π respectively (found e.g. in the *Century Dictionary* under the entry 'Notation').

The identity $G^{\ominus} = (-1)^{\sqrt{-1}}$ holds.

PEIRCE ALSO STUDIED and used in his manuscripts and correspondence a system of binary connectives that Max Fisch coined the "Box-X" notation. It consists of the 'X', or center cross (X) to which a kind of 'box' is constructed—piecemeal—with lines spanning the *top* and/or *bottom* (\boxtimes, \boxtimes) & the *left* and/or *right* sides (\boxtimes, \boxtimes) ultimately giving the sixteen unique operators in the margin.

Having 16 new unique *commands* for such a simple system would in my opinion be excessive. A single command, `\boxxoperator{}`, suffices that accepts the lines as a comma-separated list: *t* for the top line, *b* for the bottom, *r* and *l* for the right and left sides.

Unlike with `\nsroll`, you *can* leave the argument empty. In this case, only the central cross is drawn and this *is* a syntactically correct connective, though semantically quite strange (Peirce, 2023, Vol. 1, pp. 427–432). Notice that the order of the list elements doesn't matter. For example `\boxxoperator{t,l}` equals `\boxxoperator{l,t}` &c.

X	<code>\boxxoperator{}</code>
X X	
X X X X	
X X X X	
X X X X	
X	<code>\boxxoperator{t,b,l,r}</code>
A X B	C X D

IN A LETTER TO T.J. McCormack, Assistant Editor at Open Court, Peirce advised that instead of Π and Σ (`\Pi` and `\Sigma`)—which he uses copiously e.g. for quantifiers—simpler typefaces that he encountered in European mathematical journals, should be used.

Peirce writes in the letter: “They should be upright, all of one thickness, and devoid of the little finishing lines (whose name I forgot.)” MS R S-64, referring to sans-serif typefaces.¹³ These have been created as symbols:

Command name	Symbol
<code>\PPi</code> ¹⁴	Π
<code>\PSigma</code> ¹⁴	Σ

IN THE ‘LOGIC NOTEBOOK’ (MS 339), Peirce introduced binary & ternary symbols for existential graphs. These symbols differ from the ones above in that they must directly interact with ligatures.

These symbols thus have hooks in preconfigured places. In the third row they are highlighted with a \times and enumerated by setting `\debugmode` to true. The last four symbols have arguments which can be left empty.

Command name	Symbol	
<code>\heartright</code>		
<code>\heartleft</code>		
<code>\heartleftnofill</code>		
<code>\heartdown</code>		
<code>\heartup</code>		
<code>\norlike</code> ¹⁵		
<code>\inversenorlike</code> ¹⁵		
<code>\whiskers{}{}{}{}</code>		
<code>\inversewhiskers{}{}{}{}</code>		
<code>\whiskersdot{}{}{}{}</code>		
<code>\inversewhiskersdot{}{}{}{}</code>		

These symbols (unlike all the others) are *not* subject to the automatic scaling, since they belong to graph symbols and interact only indirectly with text.

¹³ Peirce’s ‘ Π ’ and the usual, modern, sans-serif ‘ Π ’ differ also. In Peirce’s description and in his handwriting the ‘bar’ *always* extends beyond the ‘legs’.

¹⁴ Internally, these symbols clear space equal to Π and Σ , so the commands `\let\Pi\PPi` and `\let\Sigma\PSigma` should be safe to use. Though in mathematical equations, even Peirce would have suggested that the normal symbols be used to denote product and summation.



¹⁵ This command name (as many others) is descriptive of the shape of the symbol rather than its function.

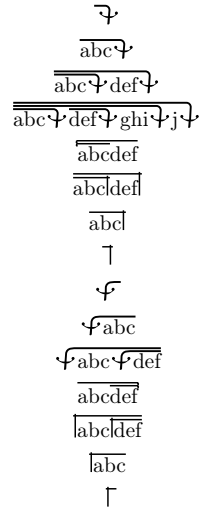
This name especially reflects the quite remarkable coincidence that this symbol very closely resembles the modern logical NOR gate symbol. Entitative graphs implement a kind of NOR logic and existential graphs, a NAND logic.

Peirce was aware of the fundamental importance—also in this respect—of his discovery. In 1886 he wrote to Allan Marquand that his logic would permit relatively simple machines to be constructed that could solve even complex mathematical problems piecemeal, and furthermore suggested that electricity be employed for the signalling.

IN ‘QUALITATIVE LOGIC’ (MS 736), Peirce introduced three new notations for illation. A line or vinculum indicates the scope.

Commands `\agoverline{}`, `\croverline{}` and `\cuoverline{}` produce them. The `\inlineagoverline{}` variant blends better inline with text. All the commands have a ‘reverse’ variant. All have one argument which can be left empty, and they can be nested.

Command name	Symbol
<code>\agoverline{}</code> , <code>\reverseagoverline{}</code>	$\overline{\quad}$, $\overleftarrow{\quad}$
<code>\inlineagoverline{}</code> , <code>\reverseinlineagoverline{}</code>	$\overline{\quad}$, $\overleftarrow{\quad}$
<code>\croverline{}</code> , <code>\reversecroverline{}</code>	$\overline{\quad}$, $\overleftarrow{\quad}$
<code>\cuoverline{}</code> , <code>\reversecuoverline{}</code>	$\overline{\quad}$, $\overleftarrow{\quad}$



These symbols—and the argument—are automatically scaled. If the argument contains text, it is typeset in `\normalsize` and also scaled with the *ad hoc* method created for the symbols. Because many typefaces do have slightly different appearances for fonts in different point sizes, scaled text inside the argument may look ever so slightly different from any surrounding text.

Ideas on further development of the package

As was mentioned in the beginning, the current version of the package is designed for drawing existential (and entitative!) graphs from Peirce’s manuscripts. Many of these graphs have highly unusual and inconsistent features. The package is designed to cope with these kinds of graphs as well and thus it doesn’t make a lot of assumptions on the structure of the graphs. It is up to the coder to provide them with the necessary structure.

It could be possible to develop the package with much more automation. Below is some food for thought on these possibilities.

PERHAPS THE EASIEST case for automation would be John Sowa’s EGIF, which is a linear notation (or an ‘interchange format’) for existential graphs. The problem is that graphs actually drawn on the sheet of assertion have no intrinsic (linear) structure. An algorithm for drawing the graphs based on EGIF could still be devised. The algorithm could even draw the ligatures in β -graphs automatically!

However, the fundamental problem persists. Again—excepting the simplest cases—there exists a multitude of possible permutations for drawing the graph-elements described by the EGIF notation. The algorithm would always have to choose arbitrary rules for the composition as the graphs wouldn’t readily resemble the user’s wishes. The algorithm could be fed with compositional hints or the user could perhaps e.g. choose from a set of the most common ‘normal forms’.

In our setting of transcribing graphs from manuscripts, this would in my opinion be a fundamentally bad idea. It would hide compo-

sitional principles behind layers of abstraction and thus most of the advantage of having any automation in the first place, would be lost. You'd be fighting against a fairly opaque algorithm. Though the current scheme can be laborious at times, it *does* preserve a relatively close homeomorphism between the code and the ensuing graph (as the example on page 15 shows).


On the other hand, for a package that would need to just draw monotonically similar existential graphs without needing to adhere at all to their counterparts in e.g. manuscripts or to the vagaries of the user, something like the EGIF approach would make a lot of sense!

THERE ARE SOME PACKAGES for L^AT_EX, e.g. XyMTeX and ChemFig, that can draw complex structural formulas of chemical compounds and molecules. These packages can handle skeletal formulas and even stereochemistry with an elegant and simple syntax, even for horribly complex molecules in organic chemistry. The question arises, whether something similar could be done for existential graphs too.

However, the laws of nature that govern chemical bonding already contain lots of structure and their usual representations include lots of conventions on how to draw these structures out. Again, the lack of obvious intrinsic structure in existential graphs thwarts most possibilities for an elegant syntax for EGS.

Taking a cue from XyMTeX, there are some ideas that could be developed further for graphs too. At least, it could be possible to create an easier syntax especially for α -graphs.

We could devise a command, say, `\graph{}` that would re-interpret some normal characters as control characters. The most obvious candidates would be '(' for the beginning of a cut and ')' for the end. Since the normal space character could take care of alignment and positioning, and the linebreak character for linebreaks, the syntax would be considerably simpler and also quite natural and intuitive.

So: `\graph{ ((A)
(B)) }` would be interpreted as `\cut{\ontop{\cut{ A }}\cut{ B }}`, or .

Clearly a simpler syntax than the current one.

Unfortunately, simplifying the drawing of ligatures in a similar way still eludes a solution. It would not be a problem to create a new control character to the `\graph` command for the end points of ligatures but routing and drawing them would still need to be done in the current, piecemeal, way.

Introduction to L^AT_EX

What is it . . .

L^AT_EX is a programming language for typesetting documents and books. It is based on the T_EX language created by Donald Knuth in the late 1970s.

L^AT_EX uses highly portable and standardized file types. It produces high-quality publication ready materials that have outstanding dimensional accuracy and excellent consistency.

Because L^AT_EX is fundamentally a programming language, its use may seem abstruse at first sight. This may especially be the case if you are accustomed to What-You-See-Is-What-You-Get programs such as Microsoft Word or the like. This obscurity soon dissolves when few fundamental concepts are introduced and explained.

LIKE ALL PROGRAMMING LANGUAGES L^AT_EX relies on a *source file*. It contains the text and its typesetting instructions called commands. A program then *compiles* this source file and produces some *output*.

SOURCE FILES, COMPILERS AND
OUTPUTS

The source file is invariably a *plaintext* file whose commands must adhere to a specific syntax. If they fail to do so the compilation will also fail and the program won't be able to produce any output.

There are quite a few programs that can compile and process L^AT_EX files and the outputs range from structured plaintext files and HTML documents to a variety of vector and raster graphics formats. Most commonly though L^AT_EX is used to produce a wholly self-contained PostScript or PDF file that is ready for printing, such as this one.

THERE ARE VARIOUS Integrated Development Environments designed for L^AT_EX that automate or help with many of the tedious tasks of codewriting and compilation. Some IDEs even come close to resembling WYSIWYG programs.

There are however fundamental differences between the two. The most important being a strict separation of form and content.

SEPARATION OF FORM AND CON-
TENT

A decent analogy is a web-page. The browser, though, is both the compiler and the output viewer for the HTML source file. The Hypertext Markup Language also uses its commands to describe the logical content of the page and the browser does the formatting.

Much like HTML's Document Type Declaration defines a grammar and vocabulary, in L^AT_EX a class file gives the structuring commands. So-called package files introduce new commands or override old ones—not entirely unlike what CSS modules do with HTML.

Often when using WYSIWYG programs the author is constantly distracted by appearances and formatting issues. When writing L^AT_EX, the key point is to write and describe the content of the document in a logical manner and let the program do all the formatting. It is especially important to understand this mindset when writing existential graphs with the `egpeirce` package. The graphs are described in text, and L^AT_EX then takes care of actually drawing them.

... and why use it?

Even though T_EX was created in the 1970s, L^AT_EX is still actively developed and new technologies are added to it. It is widely used in academia and the publishing industry.

One of Donald Knuth's original reasons for creating T_EX was the abysmal state of mathematics typesetting. L^AT_EX is *still* unparalleled in this respect.

Another singular feature in L^AT_EX is its versatile and eloquent line-breaking system. A good introduction to its manipulation and use is e.g. in Knuth (1984), and a detailed account of its development is presented in Knuth (1999, ch. 3). L^AT_EX searches through *all* possible linebreaking places. A simple but ingenious algorithm chooses a combination of them that creates the least inter-word space stretches and hyphenations one paragraph at a time for the entire page.

SOME PEOPLE MAY VIEW the fact that L^AT_EX is a programming language as a hindrance. Arguably it does create a small learning curve. But it is this very background that also sets L^AT_EX apart from other desktop publishing systems in many positive ways.

Firstly, L^AT_EX documents are extremely portable because they are simple plaintext files. Practically any computer can be used to edit them and the files are human readable as such. This is in contrast to the binary or XML files that most document editors use.

As mentioned earlier, compilation is not dependent on a single program and nearly all compilers are free and open source software. They are ported to practically all operating systems.

L^AT_EX also has an excellent track record on backwards compatibility. The first version of T_EX was made available in 1977. By 1982 the typesetting features were changed so that reproducibility across different hardware was guaranteed. The language features were frozen in 1989 when a few seldom used commands were deprecated.

Most T_EX documents that were written in 1977 will compile fine even today. Any documents written after 1982 will even hyphenate and look exactly the same. Not many other typesetting systems can claim to be able to reliably process and reproduce, at the time of writing this, almost four decades old documents.

Lastly, the unit that ties L^AT_EX dimensions and lengths to real-life measurements is the T_EX point (pt). It is defined to be $1/72.27$ inches or about 0,3515 millimeters. The fundamental smallest unit in T_EX is called a scaled point (sp) and all other length types are internally represented as integer multiples of it. T_EX uses 16 bits to represent the fractional part of a point, so $1\text{ sp} = 1/2^{16}\text{ pt}$. This is a staggeringly small quantity—roughly a hundredth of the average wavelength of visible light. L^AT_EX dimensions are thus two orders of magnitude more accurate than any normal physical manifestations of them.

Because only so many bits can be reserved for multiples of sp, the biggest single page an unadulterated version of L^AT_EX can handle is about 5 m² in size. Well within the needs of ordinary typesetting.

For a challenge, try typesetting these formulae with anything other than T_EX: $A = \prod_{k=2}^{\infty} \left[\frac{\phi k}{k-1} \right]$

$$\int_{-1}^{+1} \frac{f(x)}{\sqrt{1-x^2}} dx \approx \frac{\pi}{n} \sum_{i=1}^n f\left(\cos\left(\frac{2i-1}{2n}\right)\right)$$

PORTABILITY

BACKWARDS COMPATIBILITY

DIMENSIONAL ACCURACY

Recommended further reading

This introduction has hardly scratched the surface to $\text{T}_{\text{E}}\text{X}$ and $\text{L}_{\text{A}}\text{T}_{\text{E}}\text{X}$.

The internet is full of guides to $\text{L}_{\text{A}}\text{T}_{\text{E}}\text{X}$. *The not so Short Introduction to $\text{L}_{\text{A}}\text{T}_{\text{E}}\text{X} 2_{\epsilon}$* by Tobias Oetiker et alii (originally from 1995 and updated continuously) is considered a classic. The Wikibook for $\text{L}_{\text{A}}\text{T}_{\text{E}}\text{X}$ is also well organized, quite thorough on basic commands and perhaps also a good place to start. It's very handy as a quick-'n-dirty reference.

The quintessential tome is *The $\text{T}_{\text{E}}\text{X}$ book* by Donald Knuth himself. Although required reading for any $\text{T}_{\text{E}}\text{X}$ nician, it is also quite arcane and as the name suggests, deals exclusively with $\text{T}_{\text{E}}\text{X}$. A famous reference book is *The $\text{L}_{\text{A}}\text{T}_{\text{E}}\text{X}$ companion* by Mittelbach et al. (2004). It isn't (and cannot be) a complete listing of all packages for $\text{L}_{\text{A}}\text{T}_{\text{E}}\text{X}$ but offers a decent idea of the kinds of things it is capable of.

Lastly, *The $\text{L}_{\text{A}}\text{T}_{\text{E}}\text{X}$ graphics companion* Goossens (2008) is definitely a very important reference, especially chapters 5 and 6 on PostScript.

[Click me ↓](#)

The not so short... as a PDF file at CTAN

<http://en.wikibooks.org/wiki/LaTeX>

References

- Goossens, M. (2008). *The $\text{L}_{\text{A}}\text{T}_{\text{E}}\text{X}$ Graphics Companion*. Addison-Wesley series on tools and techniques for computer typesetting. Addison-Wesley.
- Knuth, D. E. (1984). *The $\text{T}_{\text{E}}\text{X}$ book*. Addison-Wesley, 1 edition.
- Knuth, D. E. (1999). *Digital typography*. CLSI lecture notes. Stanford, Calif. CLSI Publ. cop.
- Mittelbach, F., Goossens, M., Braams, J., Carlisle, D., and Rowley, C. (2004). *The $\text{L}_{\text{A}}\text{T}_{\text{E}}\text{X}$ Companion*. Pearson Education.
- Oetiker, T. (2021). *The Not So Short Introduction to $\text{L}_{\text{A}}\text{T}_{\text{E}}\text{X} 2_{\epsilon}$* .
- Peirce, C. S. (2019–2023). *Logic of the Future*, volume 1–3 of *Peirceana*. De Gruyter, Berlin.
- Roberts, D. D. (1973). *The Existential Graphs of Charles S. Peirce*. De Gruyter Mouton, Berlin, Boston.

Keyword & Command Index

Page numbers typeset in **bold** contain the command definition or its introduction. Entries marked with '(boolean)' indicate a Boolean switch that can be set true or false. Entries with '(counter)' receive integers: \mathbb{Z} . '(rational)' receive rational numbers greater than zero: $\mathbb{R}_{>0}^+$. '(dim)' receive $\mathbb{R}_{>0}^+$ with a valid unit. Entries with '(param)' valid strings.

- `%`, 1, 4, **15**, 16, 17
- `\`, 5, 6
- `\aggregate`, 24
- `\agoverline`, 26
- bifurcation, 9, 18
- blot, 8, **20**
- `\boxxoperator`, 24
- bridge, 1, **10**, 22
- `\colouredcuts` (boolean), 1, **8**
- `\commoncoefficient` (rational), 23
- `\croverline`, 26
- `\cuoverline`, 26
- `\cursiveimplicates`, 24
- `\cut`, 1, 6, 7–9, 11, 14, 18, 27
 - and ligatures, 9, 11
- `\cutcolour` (param), 21
- `\cutwidth` (dim), 21
- `\cutxfillcolour` (param), 1, 8, 21
- `\dbcut`, 12
- `\debugmode` (boolean), 1, **10**, 11, 14, 22, 25
- `defaultnscrollangle` (counter), 8
- `\DefNodes`, 16–18, 20, *see also*
 - `\egatn`
- `\downright`, 1, 9
- `\dragonhead`, 24
- `\egatn`, 17, 20
- EGIF, 26, 27
- `\ellipsecut` (boolean), 6, 11
- `\everygraphhook`, 1, 13
- `\flatinfy`, 24
- `\fsymbol`, 24
- gap, 1, **10**
- `\gcut`, 12
- `\graph`, 27
- `\gvvcut`, 12
- `\gvvcut`, 12
- `\heartdown`, 25
- `\heartleft`, 25
- `\heartleftnofill`, 25
- `\heartright`, 25
- `\heartup`, 25
- `\hk`, 1, 9, 10–12, *see also*
 - `\debugmode`
- `\hphantom`, 5, 15, 18
- `\hsligature`, 9
- `\implicates`, 24
- inline (environment), 1, 6, 11, 14, 15, *see also* `\notinline`
- `\inlineagoverline`, 26
- `\inversenorlike`, 25
- `\inversescroll`, 7
- `\inversevscroll`, 7
- `\inversewhiskers`, 25
- `\inversewhiskersdot`, 25
- `\li`, 1, 9, 16
- `\licolour` (param), 1, 21
- `\ligaturewidth` (dim), 1, 21
- `\longinversescroll`, 7
- `\longinversevscroll`, 7
- `\longscroll`, 7
- `\longvscroll`, 7, 17
- `\mbox`, 5
- `\napierianbase`, 24
- `\nccurve`, 1, 9, 22
- `\ncut`, 12
- `\norlike`, 25
- `\notinline` (boolean), 6, 14, 15,
 - see also* inline
- `\nscroll`, 7, 17
- `nscrollangle` (counter), 8
- `\nscrollldistance` (rational), 7
- `\nscrollwidth` (rational), 7
- `\ontop`, 1, 6, 9, 15, 27
- `\ontopl`, 1, 6, 15
- `\ontopr`, 6, 16
- `\Paries`, 24
- `\pcut`, 12
- `\Pinversepropto`, 24
- `\PPi`, 25
- `\Ppropto`, 24
- `\Pratiocircdia`, 24
- `\protect`, 23
- `\PSigma`, 25
- `\reflexivel`, 1, 9
- `\reflexiver`, 1, 9, 14, 15
- `\reverseagoverline`, 26
- `\reversecoverline`, 26
- `\reversécuoverline`, 26
- `\reversedragonhead`, 24
- `\reverseinlineagoverline`, 26
- rheme (counter), 1, **11**, 18
- `\rightdown`, 9
- `\rightup`, 9
- `\scaledsymbols` (boolean), 23
- `\scroll`, 7, 16
- `\scrollstretch` (rational), 21
- `\setstretch` (rational), 6, 15
- `\shk`, 12
- `\sligature`, 9
- `\strut`, 1, 5, 8, 15
- `\upright`, 1, 9
- `\varinclusion`, 24
- `\varwedge`, 24
- `\vcut`, 1, 6
- `\vphantom`, 5, 11, 14, 15
- `\vscroll`, 1, 7, 16, 17, 19
- `\vv`, 20
- `\vvvcut`, 6, 15, 16
- `\weirdfour`, 24
- `\weirdone`, 24
- `\weirdthree`, 24
- `\weirdtwo`, 24
- `\whiskers`, 25
- `\whiskersdot`, 25
- `\xfillstyle` (param), 21


Visual index

The visual (or graphical) index contains a visual reference of the most important *types* of existential graphs and all the logical symbols. Refer to the the Keyword and Command Index above for *all* the commands the package provides.

 ... , 6

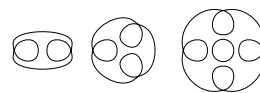
 ...  ... , 6

 ...   , 12

 ... , 6


 , 7


 , 7

 ... , 7


 ... , 8

 , 20

 , 20


 ... , 9


 , 9


 , 10

 , 10


 , 10

 , 12

 , 24

 , 24

 , 24


 , 24

 , 24

 , 24


 , 24

 , 24


 , 25

 , 24

 , 24

 , 24

 , 24

 , 26

 , 25

 , 25

 , 25