

The `pseudo` Package

Magnus Lie Hetland

June 28, 2022

Abstract

The `pseudo` package permits writing pseudocode without much fuss and with quite a bit of configurability. Its main environment combines aspects of `enumeration`, `tabbing` and `tabular` for nonintrusive line numbering, indentation and highlighting, and there is functionality for typesetting common syntactic elements such as keywords, identifiers and comments.

1 Introduction

The `pseudo` package lets you typeset pseudocode in a straightforward and not all too opinionated manner. You don't need to use separate commands for different constructs; the indentation level is controlled in a manner similar to in a `tabbing` environment:

```
1 while  $a \neq b$ 
2   if  $a > b$ 
3      $a = a - b$ 
4   else  $b = b - a$ 
5 return  $a$ 
```

```
[
  \begin{pseudo}
    while $a \neq b$          \\\+
      if $a > b$              \\\+
        $a = a - b$          \\\-
      else $b = b - a$       \\\-
    return $a$
  \end{pseudo}
]
```

If you prefer having **end** at the end of blocks, or you'd rather wrap them in C-style braces, you just put those in. Fonts, numbering, indentation levels, etc., may be configured. You import `pseudo` with:

```
\usepackage[<options>]{pseudo}
```

The only option usable here at the moment is `kw` (used in the example above), as the `\usepackage` command is a bit too eager in expanding its arguments, but there are several options that may be provided to the `\pseudoset` command, to configure things (see section 4.2).

Algorithm 1.1 Euclid’s algorithm, $\text{EUCLID}(a, b)$

Input: Two positive integers, a and b .

Output: The greatest common divisor of a and b .

1	while $a \neq b$	If equal, both are gcd
2	if $a > b$	Reduce greatest with multiple of smallest
3	$a = a - b$	a is largest
4	else $b = b - a$	b is largest
5	return a	Both are gcd, so return one

The running time is quadratic in the number of bits in the input.

Alternatives

There are many ways of typesetting code and pseudocode in \LaTeX , so if you’re unhappy with `pseudo`, you have several alternatives to choose from. I wrote `pseudo` based on my needs and preferences, but yours may differ, of course. For example, I’ve built on tabular layouts to get (i) automatic width calculations; (ii) line/row highlighting; and (iii) easy embedding in `tikz` nodes and the like. I have also set things up inspired by existing mechanisms for numbering and indenting lines, and treat the pseudocode as a form of text, rather than as a form of markup in itself. The latter point means that I don’t have separate commands for conditionals, loops, etc.

The basic style of pseudocode is inspired by the standard reference *Introduction to Algorithms* by Cormen et al. [1] (i.e., similar to that of `newalg`, `clrcode` and `clrcode3e`). Rather than locking down all aspects of pseudocode appearance, however, I’ve tried to make `pseudo` highly configurable, but if it’s not flexible enough, or just not to your liking, you might want to have a look at the following packages:

alg, algobox, algorithm2e, algorithmicx, algorithms, algpseudocodex,
algxpar, clrcode, clrcode3e, latex-pseudocode, newalg, program,
pseudocode

There are also code-typesetting packages like `listings` and `minted`, of course.

Using older \TeX distributions

The implementation of `pseudo` uses some functionality that isn’t available in older \TeX distributions, in particular, older versions of `xparse` and `expl3`. Some care has been taken to make the code backward compatible to the point where it works on \TeX Live 2016, which is what is used (at the time of writing) on arXiv. If you run into issues somewhere else (e.g., when submitting to some publisher with a custom setup), feel free to file an issue, or even provide a pull request with a fix. One thing to look out for is that older versions of `xparse` parse arguments differently, so things like

[foo\<1>	
	bar\<[h1]]

would work, but separating the arguments with spaces, as in

```
foo\\ <1>
bar\\ [hl]
```

will *not* work, though this works with more recent versions (as seen from some of my examples, later). For more advice on working around an older distribution, see also section 5.9.

2 Overview

The main component of the `pseudo` package is the `pseudo` environment, which is, in a sense, a hybrid of `enumerate`, `tabular` and `tabbing`, in that it provides numbered lines, each placed in a tabular row (for ease of highlighting and automatic column width calculation), with functionality for increasing and decreasing indentation similar to the `tabbing` commands `\+` and `\-` (in `pseudo`, combined with the row separator `\\`). Here, for example, is Euclid’s algorithm for finding the gcd of a and b :

- 1 repeat the following while $a \neq b$
- 2 if $a > b$, let $a = a - b$
- 3 otherwise, let $b = b - a$

```
\begin{pseudo}
repeat the following while $a\neq b$      \\+
  if $a > b$, let $a = a - b$             \\
  otherwise, let $b = b - a$
\end{pseudo}
```

Spacing is handled similarly to in L^AT_EX lists, with `\topsep` and `\parskip` added before and after, as well as `\partopsep` whenever the environment starts a new paragraph. The left margin (how much the pseudocode is indented wrt. the surrounding text) is set by the `left-margin` key (initially `0pt`).*

There are also some styling commands for special elements of the pseudocode:

while, **FALSE**, *rank*, “Hello!”, $\text{EUCLID}(a, b)$, $\text{length}(A)$, (*Important!*)

```
\kw{while},           % or \pseudokw    -- keywords
\cn{false},           % or \pseudocn    -- constants
\id{rank},            % or \pseudoid    -- identifiers
\st{Hello!},          % or \pseudost    -- strings
\pr{Euclid}(a, b),    % or \pseudopr    -- procedures
\fn{length}(A),       % or \pseudofn    -- functions
\ct{Important!}       % or \pseudoct    -- comments
```

* If `pseudo` occurs in a box such as `fbox`, or a `tikz` node, this spacing is dropped. See also the `compact` key for overriding this behavior.

The longer names (`\pseudokw`, `\pseudocn`, etc.) are always available; the more convenient short forms (`\kw`, `\cn`, etc.) are prone to name collisions, and are only defined if the names are not already in use when `pseudo` is imported.

The `indent-length` option, which determines the length of each indentation step, is initially set via the secondary `indent-text` key, so that the any code after `\kw{else}` aligns with the indented text (a stylistic choice from `clrscode3e`):

```
1 if  $x < y$ 
2    $x = x + 1$ 
3 else  $x = x - 1$ 
```

```
[
  \begin{pseudo}
  \kw{if}  $x < y$                                 \\\+
     $x = x + 1$                                 \\\-
  \kw{else}  $x = x - 1$ 
  \end{pseudo}
]
```

The indentation may also be configured with `indent-mark`, which inserts a mark at every indenation step:

```
1 while  $x \leq n$ 
2   |    $x = x + 1$ 
3   |   if  $f(x) < y$ 
4   |   |    $x = x + 1$ 
5   |   print  $x$ 
6 return  $x$ 
```

```
[
  \begin{pseudo}[indent-mark]
    while  $x \leq n$                                 \\\+
       $x = x + 1$                                 \\\
      if  $f(x) < y$                                 \\\+
         $x = x + 1$                                 \\\-
      print  $x$                                     \\\-
    return  $x$ 
  \end{pseudo}
]
```

The default is a vertical line, but anything else may be supplied as an argument. The color may be modified using `indent-mark-color`:

```
1 while  $x \leq n$ 
2 ▷    $x = x + 1$ 
3 ▷   if  $f(x) < y$ 
4 ▷   ▷    $x = x + 1$ 
5 ▷   print  $x$ 
6 return  $x$ 
```

```

\begin{pseudo}[
  indent-mark=$\triangleright$, indent-mark-color=teal]
  while $x \leq n$          \\\+
    $x = x + 1$             \\\
    if $f(x) < y$           \\\+
      $x = x + 1$           \\\-
    print $x$               \\\-
  return $x$
\end{pseudo}

```

The default indent mark scales with the `line-height`, to avoid gaps in the vertical lines. The width of the (default) mark can be set with `indent-mark-width`:

```

1 the
2 | lines
3 | are
4 | scaled

```

```

\begin{pseudo}[
  indent-mark, indent-mark-width=3pt, line-height=1.5]
  the          \\\+
    lines      \\\+
      are      \\\-
    scaled
\end{pseudo}

```

If you want, you can certainly create shortcuts, e.g., with simple macro definitions like `\def\While{\kw{while}}`, or with declaration commands such as `\DeclarePseudoKeyword` or `\DeclarePseudoConstant`. Procedures and functions capture parenthesized arguments and set them in math mode; this carries over in shortcuts, so if you define `\Euclid` to mean `\pr{Euclid}`, then `\Euclid(a, b)` yields `EUCLID(a, b)`.*

These commands are not used in the internals of the package, so they may be freely redefined for different styling, such as `\let\id\textsf`. They generally do some extra work, though, such as wrapping the styled text in `\textnormal` to avoid having the styles blend, adding quotes (`\st`) and handling parenthesized arguments (`\pr`). To let you hook into their appearance without messing with their definitions, each command has a corresponding font command (`\kwfont`, `\cnfont`, `\idfont`, etc.), which you may redefine. These fonts may even be set using correspondingly named options, either with `\pseudoset` or via optional keyword arguments to the `pseudo` environment:[†]

* Note that `\Euclid (a, b)`, with a space before the parenthetical, yields `EUCLID (a, b)`.

† Because of L^AT_EX expansion behavior, they can *not* be set globally when importing `pseudo`.

Euclid's algorithm is initiated with the call `Euclid(a, b)`.

```
[
  \pseudoset{prfont=\textsf}
  Euclid's algorithm is initiated with the call \pr{Euclid}(a, b).
]
```

You can also configure the quotes and comment markers:

```
1 print 'Hello, world!' // Greeting
```

```
[
  \pseudoset{
    st-left=', st-right=', stfont=\textit,
    ct-left=\texttt{[/\!/]\}, ct-right=, ctfont=
  }
  \begin{pseudo}
  \kw{print} \st{Hello, world!} \quad \ct{Greeting}
  \end{pseudo}
]
```

Note that `\stfont` and friends may either be font-switching commands like `\itshape` or formatting commands like `\textit`, though the latter are generally preferable when available. They need not be restricted to actual fonts, but may include color commands, for example.

You can also set the font for the entire code lines, using the `font` option. The command you provide there should just switch the font (i.e., not take an argument to typeset); initially, `\kwfont` is such a command:

```
1 while  $a \neq b$ 
2   if  $a > b$ 
3      $a = a - b$ 
4   else  $b = b - a$ 
```

```
[
  \begin{pseudo}[font=\kwfont]
  while $a \neq b$                                \\\+
    if $a > b$                                     \\\+
      $a = a - b$                                  \\\-
    else $b = b - a$
  \end{pseudo}
]
```

Though not the default, this is in fact an intended configuration, to reduce the markup noise for pseudocode that consists primarily of keywords and mathematics. The setting `font = \kwfont` is also available by using the `kw` option (with no arguments), e.g., by importing the package with `\usepackage[kw]{pseudo}`. If you need to typeset normal text in your pseudocode after using `font`, you can use `\textnormal` or `\normalfont`, for which `pseudo` defines aliases `\tn` and `\nf`:

```
1 for each node  $v \in V$ 
2   do something
3 for each edge  $e \in E$ 
4   do something else
```

<pre> \begin{pseudo}[kw] for \tn{each node} \$v\$ in \$V\$ \tn{do something} for \nf each edge \$e\$ in \$E\$ \nf do something else \end{pseudo} </pre>	<pre> \\+ \\- \\+ </pre>
---	--------------------------

The row separator may have multiple pluses or (more commonly) multiple minuses appended, indicating multiple increments or decrements to the indentation level:

<pre> 1 for \$k = 1\$ to \$n\$ 2 for \$i = 1\$ to \$n\$ 3 for \$j = 1\$ to \$n\$ 4 \$t_{ij} = t_{ij} \vee (t_{ik} \wedge t_{kj})\$ 5 return \$t\$ </pre>	<pre> \\+ \\+ \\+ \\+ \\+ </pre>
--	----------------------------------

The code is normally typeset in a two-column `tabular` (whose preamble, and thus number of columns, is configurable via the option `preamble`), but the first column is handled by an automatic `prefix` inserted before each line, containing the numbering and column separator (`&`). You disable the prefix for the following line by using `*`:

<pre> 1 this line has an automatic prefix this line does not 2 but this one does </pre>	<pre> \\+ \\+ \\+ </pre>
---	--------------------------

This star also works after `\begin{pseudo}`. Note that in order to prevent your code from ending up in the numbering column, you must insert a column separator manually. A version of the `\pr` command, called `\hd` (or `\pseudo hd`,

where `\hd` stands for *header*) instead wraps a procedure call in a `\multicolumn`, so it can be used, for example, as an unnumbered header line:

```

EUCLID(a, b)
1  if b == 0
2      return a
3  else return EUCLID(b, a mod b)

```

```

[
    \begin{pseudo}[kw]*
    \hd{Euclid}(a, b)
    if $b \== 0$
        return $a$
    else return \pr{Euclid}(b, a \bmod b)
    \end{pseudo}
]

```

As can be seen in this example, `\==` (or `\eqs`) is a notational convenience defined by `pseudo`, along with interval dots `\..` (or `\dts`). Other special symbols may be found in other packages. For example, if you want to use `:=` for assignment, you can use `\coloneqq` from `mathtools` (perhaps with `\let\gets\coloneqq`).^{*}

As can be seen, one use of `*` is to get an unnumbered line, but you could also insert custom material in the first column. The lines are numbered by the counter `pseudoline`, so you could, for example, do:

```

A Look!
B We're using letters!

```

```

[
    \begin{pseudo}*
    \stepcounter{pseudoline}\Alph{pseudoline} & Look! \\*
    \stepcounter{pseudoline}\Alph{pseudoline} & We're using letters!
    \end{pseudo}
]

```

This is a bit cumbersome, so there are some shortcuts. First of all, rather than replacing the entire `prefix`, you can replace only a *part* of it, namely the `label`, retaining counter increments and column separators. You can set this key for each line individually with an optional argument to the row separator, i.e., `\\[label = <commands>]`, or at some higher level. Within the `pseudo` environment, there is also a counter named `*` that is simply a local clone of `pseudoline`, letting you use starred versions of counter commands, similarly to how label definitions work in `enumitem`:[†]

```

1: Look!
2: We're using something custom!

```

^{*} Tip: If you want to use a left-arrow for assignment, but think it's a bit large in Computer Modern or Latin Modern, you can use the `old-arrows` package, so `x \gets y` yields $x \leftarrow y$.

[†] Also like in `enumitem`, there's a `start` key for setting the first line number.


```

\pseudoset{label=\small\arabic*:}
\begin{pseudo}
Look! \\\
We're using something custom! \label{custom-line}
\end{pseudo}

```

Note that if I refer to the labeled line with `\ref`, I'll just end up with 2, which is probably what I'd want in this case. If you want a custom reference format as well, you can set that with the `ref` key, in the same way as with `label`. If you use the key without arguments, it'll use the same format as the one provided to `label`:

- (i) Look!
- (ii) We're using Roman numerals!
- (iii) And here's a reference to line (ii).

```

\pseudoset{label=(\textit{\roman*}), label-align=l, ref}
\begin{pseudo}
Look! \\\
We're using Roman numerals! \label{roman-line} \\\
And here's a reference to line \ref{roman-line}.
\end{pseudo}

```

The `label-align` key sets the alignment of the label column, and can be `l`, `r` or `c` (or really any other column type compatible with the `array` package; you could use a `p{...}` column to get fixed width, for example).

Highlighting can also be done in a similar manner, by, e.g., inserting a `\rowcolor` at the start of the first column. Rather than doing this manually, you could use the `bol` key, which inserts a command at the beginning of the line—or the `hl` key, which is equivalent to `bol-prepend = \pseudohl`:

I'm not highlighted
But I am!

```

\begin{pseudo*}
I'm not highlighted \[hl]
But I am!
\end{pseudo*}

```

Initially, the `\pseudohl` command that is inserted is simply a `\rowcolor` that uses `hl-color`, but you're free to redefine this command to whatever you'd like.

In the previous example, there is no spacing to the sides of the table contents. This is normally what you'd want, for example, to keep the pseudocode aligned with the surrounding text. However, when using row highlighting (e.g., because you are stepping through the code in some presentation), that alignment may

be less of an issue—and you’d rather widen the highlight a bit. The horizontal padding on each side is controlled by the `hpad` key. You can either specify a length, or just turn on the default, by not supplying an argument. There’s a similar option, `hsep`, which controls the separation between the two columns.

```
1 let's
2 use
3 some
4 padding!
```

```
\begin{pseudo}[hpad, hsep=1em, indent-length=1em]
  let's                                     \\\+
      use                                 \\\-
      some                               \\\+ [h1]
      padding!
\end{pseudo}
```

For ease of use with `beamer`, the various `pseudo` options support `beamer` overlay specifications. For example, using `h1<1>` means that the `h1` specification would only take effect on slide 1. If you use such an overlay specification on a key when *not* using `beamer`, the key is simply ignored.

What is more, the row separator *itself* takes an overlay specification as a shortcut for the one on `h1`, so `\\<1,2-4>` is equivalent to `\\[h1<1,2-4>]`. Just like with the optional arguments, space before the overlay specification is ignored, so you’re free to put the specification in front of the line in question:

1 Go to line 3	1 Go to line 3	1 Go to line 3	1 Go to line 3
2 Go to line 4	2 Go to line 4	2 Go to line 4	2 Go to line 4
3 Go to line 2	3 Go to line 2	3 Go to line 2	3 Go to line 2
4 Go to line 1	4 Go to line 1	4 Go to line 1	4 Go to line 1

```
% In a beamer presentation
\begin{pseudo}
  <1> Go to line 3                               \\\
  <3> Go to line 4                               \\\
  <2> Go to line 2                               \\\
  <4> Go to line 1                               \\\
\end{pseudo}
```

You might have expected these overlay specifications to indicate *visibility*, as they do for the `\item` command in `\enumerate`, for example. However, in step-wise animations, highlighting patterns (showing which line is currently executed, for example) tend to be more complex than, say, a gradual uncovering—and therefore in greater need of abbreviation.

To control visibility, you could, for example, add `\pause` at the end of each line, before the row separator. You can also do this using the `eo1` key, either

per line or at the top level, with `eol = \pause`. There is even the shortcut key `pause` for this specific purpose (equivalent to `eol-append = \pause`):

1 Eeny	1 Eeny	1 Eeny	1 Eeny
2 Meeny	2 Meeny	2 Meeny	2 Meeny
3 Miny	3 Miny	3 Miny	3 Miny
4 Moe	4 Moe	4 Moe	4 Moe

```

% In a beamer presentation
\setbeamercovered{transparent}
\begin{pseudo}[pause]
    Eeny
    Meeny
    Miny
    Moe
\end{pseudo}

```

The `eol` value is only inserted wherever `\\` starts a new line (i.e., not at the end of the environment), so in this case only three `\pause` commands are inserted.

The previously discussed configuration keys are described in more detail in section 4. You can create your own presets or *styles* using `\pseudodefestyle`. This command takes two arguments; the first is the name of a key, and the second is a key-value list, as you would have supplied it to `\pseudoset`. This is exactly how the `starred` style is defined (see page 70), clearing the prefix and reducing the preamble to a single column. This style is what's used in the starred, unnumbered version of the `pseudo` environment:

```

while a ≠ b
    if a > b
        a = a - b
    else b = b - a
return a

```

```

\begin{pseudo*}
    while $a \neq b$
        if $a > b$
            $a = a - b$
        else $b = b - a$
    return $a$
\end{pseudo*}

```

3 Boxes and floats

There are (at least) two different ways of viewing a block of pseudocode: as an inline element, like equations, or as a float, like figures and tables. For example,

Cormen et al. [1] place their pseudocode inline, and refer to the algorithms by name (e.g., “DIJKSTRA”), while Williamson and Shmoys [5] place them in floats, and refer to them by number (e.g., “Algorithm 3.1”).*

Just using the `pseudo` environment is sufficient for typesetting pseudocode as part of the body text. If you wish to place your pseudocode in a float, you can easily use a package such as `float`.† You could also use the float environments supplied with packages such as `algorithms`, `algorithmicx` and `algorithm2e`.‡

The `pseudo` package does provide some specialized setup, however, using `tcolorbox`. This also lets you typeset non-float pseudocode with a colored background, for example, like Cormen et al. do in the most recent version of their textbook [2].

Note: For performance reasons, `pseudo` does *not* automatically import `tcolorbox`; if you want to use the float functionality, you will need to import `tcolorbox` yourself:

```

\usepackage{pseudo}
% ...
\usepackage{tcolorbox}           % possibly with options
\tcbuselibrary{skins,theorems} % remember these

```

It does not matter whether you import `tcolorbox` before or after `pseudo`, but make sure you also import the two libraries `skins` and `theorems`, as in the example above.

The styles defined by `pseudo` are versions of the commonly used *boxed* and *ruled* styles, as found in, e.g., `float`, as well as the *boxruled* and *tworuled* styles found in `algorithm2e`. In addition, there’s a *filled* style, with a colored background. If you wish to customize and extend the box style, `pseudo/boxruled` is probably the best starting point, as the other styles disable the default frame drawing.

The ruled style is one of the more common ones in use in publications. This is a style originally used for (non-floating) tables in *Concrete Mathematics* [3]. Rather than reproducing the look of those tables directly, `pseudo` aims to match the style of `booktabs`, with spacing and line thicknesses taken from its constants such as `\aboverulesep`, `\heavyrulewidth`, etc. (with defaults provided if `booktabs` has not been imported).§

The `pseudo` box styles can be used directly to style `tcolorbox` environments, possibly with additional `tcolorbox` options for customization:

* A third option that is sometimes used is to use a theorem-like environment for your algorithms. There are many packages to help with this; just search CTAN for “theorem”.

† Or you could do a quick CTAN search for “float”, or a look at the recommendations related to the `float` package, will give you many options, with varying functionality.

‡ The definition of `\==` doesn’t properly carry over into floats. It’s properly redefined inside `pseudo`, so you probably won’t notice, but if you wish to use the symbol outside the `pseudo` environment, but in a float (e.g., inside `\caption`), you’ll need to either call `\pseudoeq` to re-establish `pseudo`’s redefinition of `\=` or simply use `\eqs` instead of `\==`.

§ In `booktabs`, the contents between the top rules is the header row, whereas in the *Concrete Mathematics* style, it’s the caption.

```

BORUVKA( $V, E, w, T$ )
1  while  $E$  is not empty
2      for each  $u \in V$ 
3          add light edge  $uv \in E$  to  $T$ 
4      for each  $e \in T$ 
5          contract  $e$ 

```

```

% In preamble:
% \usepackage[cmk]{xcolor}
\definecolor{lighttan}{cmk}{0,0.05,0.17,0} % from clrscde4e.sty

\begin{tcolorbox}[pseudo/filled, colback=lighttan]
  \begin{pseudo}*
    \hd{Bor}{r}{u}{vka}(V, E, w, T) \\\
    \kw{while}  $E$  is not empty \\\
    \kw{for} each  $u \in V$  \\\
      add light edge  $uv \in E$  to  $T$  \\\
    \kw{for} each  $e \in T$  \\\
      contract  $e$ 
  \end{pseudo}
\end{tcolorbox}

```

You can also new environments with `\newtcolorbox`,* but the most common use-case will probably be to define a (possibly floating) theorem-style environment, using `\newtcbtheorem` (probably in the preamble):

```

\newtcbtheorem{algorithm}{Algorithm}{pseudo/ruled}{alg}

```

Here `algorithm` is the name we’ve chosen for our new environment, `Algorithm` is the label to be used when numbering (i.e., “Algorithm 1,” etc.), `pseudo/ruled` is the ruled box style, and `alg` is a prefix that will be used in automatically labeling our boxes.

If you want a *floating* box (like figures and tables, for example), simply add the key `float` alongside the box style, such as:

```

\newtcbtheorem{algorithm}{Algorithm}{pseudo/ruled, float}{alg}

```

Other `tcolorbox` styling options may be inserted in the same place. One can also supply some *init options* as a first argument, for configuring the automatic numbering. For example, if we want our algorithms to be numbered within sections, and we wish to provide `cleveref` with the appropriate names, we could define the environment like this:†

* See the `tcolorbox` for details and alternatives.

† If you use the `crefname` option, you should make sure to place your `\newtcbtheorem` command in the preamble, and not in the document body, for the naming to take effect.

```

\newtcbtheorem[
    number within = section,
    crefname = {algorithm}{algorithms}
]{%
    {algorithm}{Algorithm}{pseudo/ruled, float}{alg}

```

Once our environment has been defined with `\newtcbtheorem`, it can be used as follows:

Algorithm 3.1 Sort an array A of n elements.

```

1   $i = 1$ 
2  while  $i < n$ 
3      if  $i == 1$  or  $A[i - 1] \leq A[i]$ 
4           $i = i + 1$ 
5      else swap  $A[i - 1]$  and  $A[i]$ 
6           $i = i - 1$ 

```

```

\begin{algorithm}{Sort an array  $A$  of  $n$  elements.}{gnome}
\begin{pseudo}
     $i = 1$                                 \\\
    while  $i < n$                              \\\+
        if \nf  $i == 1$  or  $A[i-1] \leq A[i]$       \\\+
             $i = i + 1$                          \\\-
        else \nf swap  $A[i-1]$  and  $A[i]$           \\\+
             $i = i - 1$ 
    \end{pseudo}
\end{algorithm}

```

The first argument is the title, or “caption,” and the second argument (`gnome`) is the *marker*, which is combined with the prefix (in our case, `alg`) to create the label, `alg:gnome`, which can be used with `\ref` or (using `cleveref`) `\cref`, etc.*

Algorithm 3.1 is the well-known *gnome sort*, by Sarbazi-Azad and Grune.

```

\Cref{alg:gnome} is the well-known \emph{gnome sort}, by
Sarbazi-Azad and Grune.

```

Algorithms 3.2 to 3.5 are typeset with the remaining box styles.

* The separator (`:`) can be configured; see the `tcolorbox` docs.

Algorithm 3.2 pseudo/boxed

```

1   $i = 1$ 
2  while  $i < n$ 
3      if  $i == 1$  or  $A[i - 1] \leq A[i]$ 
4           $i = i + 1$ 
5      else swap  $A[i - 1]$  and  $A[i]$ 
6           $i = i - 1$ 

```

Algorithm 3.3 pseudo/boxruled

```

1   $i = 1$ 
2  while  $i < n$ 
3      if  $i == 1$  or  $A[i - 1] \leq A[i]$ 
4           $i = i + 1$ 
5      else swap  $A[i - 1]$  and  $A[i]$ 
6           $i = i - 1$ 

```

Algorithm 3.4 pseudo/tworuled

```

1   $i = 1$ 
2  while  $i < n$ 
3      if  $i == 1$  or  $A[i - 1] \leq A[i]$ 
4           $i = i + 1$ 
5      else swap  $A[i - 1]$  and  $A[i]$ 
6           $i = i - 1$ 

```

Algorithm 3.5 pseudo/filled

```

1   $i = 1$ 
2  while  $i < n$ 
3      if  $i == 1$  or  $A[i - 1] \leq A[i]$ 
4           $i = i + 1$ 
5      else swap  $A[i - 1]$  and  $A[i]$ 
6           $i = i - 1$ 

```

Unnumbered boxes may be constructed with `\newtcolorbox`, using the same styles. These boxes, by default, have no title part—only the main body, containing the pseudocode itself:

```

SUM( $a, b$ )
1  if  $b == 0$ 
2      return  $a$ 
3  return SUM( $a, b - 1$ ) + 1

```

```

\newtcolorbox{pseudobox}{pseudo/filled}

\begin{pseudobox}
\begin{pseudo}*
  \hd{Sum}(a, b)
  if $b \le 0$
    return $a$
  return $\pr{Sum}(a, b - 1) + 1$
\end{pseudo}
\end{pseudobox}

```

It is possible to add titles using the `title` key, but then the definition must be expanded slightly, to permit arguments, e.g.:

```

\newtcolorbox{pseudobox}[1][]{pseudo/filled, #1}

```

Here we've added a single argument (`[1]`), with an empty default (`[]`), and this is spliced into the box options at the end. Now we may configure each box individually, as we please:

SUM(a, b)
<pre> 1 if b == 0 2 return a 3 return SUM(a, b - 1) + 1 </pre>

```

\begin{pseudobox}[title={\pr{Sum}(a, b)}]
\begin{pseudo}
  if $b \le 0$
    return $a$
  return $\pr{Sum}(a, b - 1) + 1$
\end{pseudo}
\end{pseudobox}

```

Colors (e.g., `colback`, `colbacktitle` or `colframe`), fonts (e.g., `fonttitle`), line thicknesses (`boxrule` or `titlerule`), punctuation (`separator sign` and `terminator sign`) etc., may also be configured, either for all the boxes of this type (directly in the call to `\newtcolorbox`) or for any individual box, as with the `title` key in the previous example. The style that is closest to a plain, default `tcolorbox` is `pseudo/boxruled`, which may be a good starting-point for this kind of configuration. However, if you just wish to add some minor tweaks to one of the existing `pseudo` styles (e.g., changing the colors of `pseudo/filled`), starting with that style may be easier.

The contents of one of these boxes need not be restricted to pseudocode—the spacing is set up to handle plain text as well. For example, you may want to specify inputs and outputs:*

Algorithm 3.6 Gnome sort

Input: An array A of length n .

Output: A , sorted in nondescending order.

```

1  $i = 1$ 
2 while  $i < n$ 
3   if  $i == 1$  or  $A[i - 1] \leq A[i]$ 
4      $i = i + 1$ 
5   else swap  $A[i - 1]$  and  $A[i]$ 
6      $i = i - 1$ 

```

The running time of the algorithm is quadratic.

```

\begin{algorithm}{Gnome sort}{} % environment defined earlier

\textbf{Input:} An array  $A$  of length  $n$ .

\textbf{Output:}  $A$ , sorted in nondescending order.

\begin{pseudo}
   $i = 1$                                 \\\
  while  $i < n$                              \\\+
    if \nf  $i == 1$  or  $A[i-1] \leq A[i]$        \\\+
       $i = i + 1$                              \\\-
    else \nf swap  $A[i-1]$  and  $A[i]$            \\\+
       $i = i - 1$ 
\end{pseudo}

The running time of the algorithm is quadratic.

\end{algorithm}

```

4 Reference

This section gives an overview of all the moving parts of the package. A *default* value is one used implicitly if the key is specified with no explicit value given, while an *initial* value is one provided to the key at the point where `pseudo` is imported. Several commands (such as, e.g., `\pseudoprefix`) may be modified using corresponding keys (e.g., `prefix`). When the behavior of such commands is described, the description references their initial behavior.

* Common alternatives to “Input/Output” are “Require/Ensure” and “Data/Result.”

4.1 Line structure

Each line of a `pseudo` environment is (initially) structured as follows:

bol	step	label	&	save	ind.	font	body	eol	\\
prefix				setup					
Inserted by \\ (not *)				Part of preamble			Inserted by \\ (not last)		

The components in the **prefix** are populated by the `\\` command (or the beginning of the environment), the ones in the **setup** by the **preamble**, and the actual body is supplied by the user, inside the environment, terminated by the row separator `\\` (which then goes on to populate the next row, and so on). The **eol** part is also inserted by `\\`, except if it's used after the last line (where it doesn't really do anything).^{*} The following describes the default behavior, which can be modified substantially by setting the appropriate options (e.g., **prefix** and **setup**).

bol This field is inserted by `\\` (and `\begin{pseudo}`) at the beginning of the following line, using the `\pseudobol` command. Because it's the very beginning of the tabular row, it may be used for things like `\rowcolor` when highlighting lines (as with the `hl` key).

step This refers to a call to `\stepcounter*` (where `*` is an alias for `pseudoline`), getting the counter ready for the label itself. Note that this does *not* use `\refstepcounter`, so at this point the counter has not been saved yet (and so you should not use `\label` to refer to it at this point).

label This is where the numbering label is inserted, using `\pseudolabel`; initially, this inserts `\arabic*`.

& At the end of the prefix is the column separator, closing the label column and beginning the code line column.

save Now that we're in the column where the user will normally insert text and code, we save `pseudoline` so it may be used with `\label` and `\ref`, etc. This is done using `\pseudosavelabel`, which first *decrements* the counter (to undo the increment before the label) and then calls `\refstepcounter`.

ind. Inserts the appropriate amount of indentation (with an indent step length set by `indent-length` or `indent-text` and the indentation level set by `+/- flags` or `indent-level`), using `\pseudoindent`.

font Inserts the base font, using `\pseudofont`.

body This is where the manually written body of the code line appears.

eol Inserted by the terminating `\\` (using `\pseudoeol`), unless we're at the end of the environment. Useful, e.g., for taking actions such as a `beamer \pause` (cf., `pause`) between the lines.[†]

^{*} Thus, **eol** acts more as a line *separator* than a line *terminator*.

[†] If the same action must be taken after the last line, you can simply insert it there manually.

`\` The row/line separator. Ends one line (inserting `eol`) and begins another (inserting `prefix`). As in tabulars in general, this command is also permitted after the final line of the environment, but there it does no real work (i.e., it does not insert `eol` and does not start a new line).

4.2 Command and key reference

In addition to descriptions of the various commands and options/keys (in alphabetical order), you'll find definitions of a couple of counters here (`*` and `pseudoline`).

`*`

This counter is a duplicate of `pseudoline`, available inside `pseudo`. It makes it possible to simplify calls such as `\arabic{pseudoline}` to starred forms such as `\arabic*`, like in `enumitem`. These short forms are available (and intended) for use in `label` and `ref`.

`\.`

This is a shortcut that hijacks the normal `\.` accent command, so that if it is called with `.` as an argument, the result is `\dts`. In other words, the command `\.` is really the call `\.{.}`. For any other arguments, the original `\.` is used, so while `$1\..n$` produces `1..n`, `\.o` still yields `ô`.

`\=`

This is a shortcut that hijacks the normal `\=` accent command, so that if it is called with `=` as an argument, the result is `\eqs`. In other words, the command `\=` is really the call `\={=}`. For any other arguments, the original `\=` is used, so while `$x\==y$` produces `$x = y$` , `\=o` still yields `ô`. In some contexts, this may not work because `\=` has reverted to its original meaning (as is currently the case if you try to use it within a custom float, as in section 3, or a standard one such as `figure`). In this case, you can restore the `pseudo` meaning (and the `\=` shortcut) by using `\pseudoeq`. In some cases, you may want to just use `\eqs` instead.

`\+ - * <⟨overlay specification⟩> [⟨line options⟩]`

This row separator is the workhorse of the `pseudo` package. Just as in a `tabular` environment, it signals the end of a line. It is optional after the list line, where it doesn't do any work. The command may be followed by a series of one or more plus (+) signs, each of which will increment the indentation level before starting a new line; similarly, it may be followed by one or more minus (-) signs, each of which will decrement the indentation level. Normally, the command will insert a `prefix` at the beginning of the new line; if the star (*) flag is used, this prefix is not inserted.

The optional overlay specifications refer to the `hl` key, so `\<3>` is equivalent to `\[hl<3>]`. This applies to the following line, as do other options set explicitly as optional arguments. Note that options are set locally, *before* the new line (and a new scope) is started, so unless they are handled specifically (in order to carry over), they will have no effect. Thus, even though all options are available here, not all make sense. (Consult individual option keys for intended use.)

The pluses and minuses are conceptually part of the command name, and there should be no whitespace before the star (*). You are, however, free to insert whitespace before the overlay specification and the line options. This means that you may, for example, place the overlay specification at the beginning of the following line in the source.

\arabic*

See *.

begin-tabular = $\langle commands \rangle$ (no default)

The actual command for beginning the **tabular** or **tabular**-like environment used by **pseudo**. Normally not needed, as the **tabular** behavior may be modified by other keys, but could be used to use some other tabular environment, e.g., from packages such as **tabularx** or **longtable**. Rather than **\begin{tabularx}** and **\end{tabularx}**, the command versions **\tabularx** and **\endtabularx** should be used. Commands such as **\pseudopos** and **\pseudopreamble** may be used as part of the setup:

```

\pseudoset{
  begin-tabular =
    \tabularx{\linewidth}{\pseudopos}{\pseudopreamble},
  end-tabular = \endtabularx
}

```

bol = $\langle commands \rangle$ (no default, initially empty)

Used to set **\pseudobol**, which is inserted at the beginning of each line. See also **bol-append** and **bol-prepend**.

bol-append = $\langle commands \rangle$ (no default)

Locally appends $\langle commands \rangle$ to **bol**.

bol-prepend = $\langle commands \rangle$ (no default)

Similar to **bol-append**, except that $\langle commands \rangle$ are added to the *beginning* of **bol**.

\cn{ $\langle name \rangle$ }

Indicates a constant (such as **TRUE** or **NIL**). First wraps the argument in **\textnormal** and then uses **\cnfont**. See also **\DeclarePseudoConstant**. This is a convenience for typesetting constants, and you may freely redefine it to whatever you prefer. If some package defines **\cn** before **pseudo** is loaded, **pseudo** will not overwrite it. The command will still be available, as **\pseudocn**.

cnfont = $\langle command \rangle$ (no default, initially **\textsc**)

Used to set **\cnfont**, which is used as part of **\cn**. May be set to take a single argument or none. Not restricted to actual font commands; you may also mix in **\textcolor** or the like.

\cnfont

The command set by the **cnfont** option. Used as part of **\cn**.

compact = $\langle \text{boolean} \rangle$ (default **true**, initially **false**)

The **pseudo** environment emulates the built-in L^AT_EX lists when it comes to spacing above and below, in normal text. If the environment is part of an ongoing paragraph, paragraphs will be inserted above and below, along with whitespace specified by **topsep** (and **\parskip**). If the environment begins a paragraph of its own, additional whitespace is added, as specified by **partopsep**. It is also possible to specify space to insert to the left of the environment, using **left-margin**.

However, these spacing commands don't work well inside **\mbox**, **\fbox**, etc. To avoid getting into trouble, **pseudo** determines that the environment should be *compact*, and drop this surrounding space, if we're in inner horizontal mode at the beginning of the environment. This will also turn off setting **\prevdepth** (cf. **prevdepth**).

1 if we're in a node
2 there's no added space

```
% In preamble:
% \usepackage{tikz}
\begin{tikzpicture}
  \draw (0,0) node [draw] {%
    \begin{pseudo}
      if we're in a node \\\+
        there's no added space
    \end{pseudo}};
\end{tikzpicture}
```

This may not be enough, however. For example, if you're using **standalone** to produce individual pseudocode images, this compactness may not be triggered automatically. In such cases, you can override the behavior using the **compact** key, manually specifying whether you want the pseudocode to be compact or not.

\ct{ $\langle \text{text} \rangle$ }

Indicates that $\langle \text{text} \rangle$ is a comment, (*typeset like this*). You can customize the comment appearance using **ctfont**, **ct-left** and **ct-right**:

```
1 y = 1
2 x = 2     /* this is a comment */
3 z = 345   /* this is another comment */
```

```

\pseudoset{
  ctfont=\color{black!75},
  ct-left=\unskip\qqquad\texttt{ /* },
  ct-right=\texttt{ */}
}
\begin{pseudo}
  $y=1$ \
  $x=2$ \ct{this is a comment} \
  $z=345$ \ct{this is another comment}
\end{pseudo}

```

An alternative to using `\ct` is to simply set comments in a separate column, as demonstrated in section 5.4. Or even without a separate column, if you use a `tabularx` as described there, and set the tabular width explicitly, you could insert an `\hfill` into `ct-right` and get all end-markers aligned at the right-hand side:

```

1  x = 1
2  y = 2      /* this is a comment                               */
3  z = 345    /* this is another comment                         */

```

Or if you'd rather have the comments right-aligned (like you can in, e.g., `algorithm2e`), you could use insert the `\hfill` at the beginning of the `ct-left`:

```

1  x = 1
2  y = 2                                     /* this is a comment */
3  z = 345                                  /* this is another comment */

```

ct-left = $\langle text \rangle$ (no default, initially `()`)

Text or commands inserted at the start of a comment, when using `\ct`.

ct-right = $\langle text \rangle$ (no default, initially `()`)

Text or commands inserted at the end of a comment, when using `\ct`.

ctfont (no default, initially `\textit`)

The font of the main text of a comment, when using `\ct`.

\ctfont

The command set by the `ctfont` option. Used as part of `\ct`.

\DeclarePseudoComment $\{\langle shortcut \rangle\}\{\langle comment \rangle\}$

Used to declare a macro that expands to a comment. For example:

```

x = y      (Important!)

```

```

\DeclarePseudoComment \Imp {Important!}
$x = y$ \qqquad \Imp

```

See also `\ct`. (Note that `\pseudoct` is used internally here.)

\DeclarePseudoConstant{*<shortcut>*}{*<constant>*}

Used to declare a macro that expands to a constant. For example:

FALSE

```
\DeclarePseudoConstant \False {false}
\False
```

See also \cn. (Note that \pseudocn is used internally here.)

\DeclarePseudoFunction{*<shortcut>*}{*<function>*}

Used to declare a macro that expands to a function. For example:

length(*A*) or *length*[*A*]

```
\DeclarePseudoFunction \Ln {length}
\Ln(A) or \Ln[A]
```

See also \fn. (Note that \pseudofn is used internally here.)

\DeclarePseudoIdentifier{*<shortcut>*}{*<identifier>*}

Used to declare a macro that expands to an identifier. For example:

rank

```
\DeclarePseudoIdentifier \Rank {rank}
\Rank
```

See also \id. (Note that \pseudoid is used internally here.)

\DeclarePseudoKeyword{*<shortcut>*}{*<keyword>*}

Used to declare a macro that expands to a keyword. For example:

while

```
\DeclarePseudoKeyword \While {while}
\While
```

See also \kw. (Note that \pseudokw is used internally here.)

\DeclarePseudoNormal{*<shortcut>*}{*<text>*}

Used to declare a macro that expands to normal text. For example:

if *x* == NIL

halt with an error message

```

\DeclarePseudoNormal \Error {halt with an error message}
\begin{pseudo*}[kw]
  if $x \!= \cn{nil}$ \!+
    \Error
\end{pseudo*}

```

See also `\tn`. (Note that `\pseudotn` is used internally here.)

`\DeclarePseudoProcedure``{<shortcut>}{<procedure>}`

Used to declare a macro that expands to a procedure. For example:

`EUCLID(a, b)`

```

\DeclarePseudoProcedure \Euclid {Euclid}
\Euclid(a, b)

```

See also `\pr`. (Note that `\pseudopr` is used internally here.)

`\DeclarePseudoString``{<shortcut>}{<string>}`

Used to declare a macro that expands to a string. For example:

`“Hello!”`

```

\DeclarePseudoString \Hello {Hello!}
\Hello

```

See also `\st`. (Note that `\pseudost` is used internally here.)

`dim`

Dims the following line. Equivalent to:

```

\pseudodefestyle{dim}{
  bol-append = \color{\pseudodimcolor},
  setup-append = \color{\pseudodimcolor}
}

```

May be used to dim out inactive or currently less relevant lines (possibly using overlays; see page 10).

`GNOME-SORT(A)`

```

1  i = 1
2  while i ≤ length[A]
3    if i == 1 or A[i] ≥ A[i − 1]
4      i = i + 1
5    else swap A[i] and A[i − 1]
6      i = i − 1

```



```

\begin{pseudo}[kw, dim-color=black!25]*
\hd{Gnome-Sort}(A)                                \\\
[dim]  $i = 1$                                       \\\
[dim]  while $i \leq \fn{length}[A]$                 \\\+
        if $i \neq 1$ or $A[i] \geq A[i-1]$          \\\+
            $i = i + 1$                               \\\-
        else \nf swap $A[i]$ and $A[i-1]$           \\\+
[dim]
[dim]  $i = i - 1$
\end{pseudo}

```

See also `bol-append`, `setup-append` and `dim-color`.

dim-color = $\langle color \rangle$ (no default, initially `\pseudohlcolor`)

Sets the color used by `dim` (available as `\pseudodimcolor`). The initial value is the one set by `hl-color`.

\dts

A two-dot ellipsis, for use in the Wirth interval notation $1..n$, typeset as Graham, Knuth, and Patashnik did in *Concrete Mathematics* [3]. Its definition is the same as in `gkpmac`. Also accessible via the `\dots` shortcut.

end-tabular (no default, initially `\end{tabular}`)

The actual command for ending the `tabular` or `tabular-like` environment used by `pseudo`. (See `begin-tabular`.)

eol = $\langle commands \rangle$ (no default, initially empty)

Sets `\pseudoeol`, which is inserted at the end of all but the last line by `\`. See also `eol-append` and `eol-prepend`.

eol-append = $\langle commands \rangle$ (no default)

Locally appends $\langle commands \rangle$ to `eol`.

eol-prepend = $\langle commands \rangle$ (no default)

Similar to `eol-append`, except that $\langle commands \rangle$ are added to the *beginning* of `eol`.

\eqs

Two equality signs typeset together as a binary relation, as in $x == y$ (as opposed to the wider $x = y$, resulting from `$x = y$`). It emulates the `stix` symbol `\eqeq`, but for use with Computer Modern (the default L^AT_EX font) or Latin Modern (available via the `lmodern` package). It should work just fine with other fonts. Also accessible via the `\==` shortcut, and configurable via `eqs-pad`, `eqs-scale` and `eqs-sep`.

eqs-pad = $\langle muskip \rangle$ (no default, initially 0.28mu)

The amount of space inserted on each side of `\eqs`.

eqs-scale = $\langle number \rangle$ (no default, initially 0.6785)

The amount of horizontal scaling applied to the `=` signs in `\eqs`.

eqs-sep = $\langle muskip \rangle$ (no default, initially 0.63mu)

The amount of space inserted between the two = signs in `\eqs`.

\fn{ $\langle name \rangle$ }($\langle arguments \rangle$)

Indicates a function name, such as *length*, and is initially more or less an alias for `\id`. The optional arguments (given in parentheses) are typeset in math mode, so `\fn{length}(A)` yields *length*(*A*). Sometimes square brackets are used with functions that are meant to indicate array lookups or some property access or the like. This works in the same manner, so `\fn{length}[A]` yields *length*[*A*]. This behavior of picking up arguments carries over if you define a shortcut, of course:

We're not in math mode, but the argument of *length*[*A*] is.

```
\def\Ln{\fn{length}}
We're not in math mode, but the argument of \Ln[A] is.
```

See also `\DeclarePseudoFunction`. This is a convenience for typesetting function names, and you may freely redefine it to whatever you prefer. If some package defines `\fn` before `pseudo` is loaded, `pseudo` will not overwrite it. The command will still be available, as `\pseudofn`.

fnfont = $\langle font \rangle$ (no default, initially `\idfont`)

Used to set `\fnfont`, which is used as part of `\fn`. May be set to take a single argument or none. Not restricted to actual font commands; you may also mix in `\textcolor` or the like.

\fnfont

The command set by the `fnfont` option. Used as part of `\fn`.

font = $\langle command \rangle$ (no default, initially `\normalfont`)

Sets the base font used in the code lines. Initially, this is just `\normalfont`, but the `kw` switch is a convenient way to set it to the keyword font `\kwfont`. This is presumed to be a common case, under the assumption that most of the pseudocode will consist of either keywords or mathematics. If you'd rather explicitly mark up your keywords, leaving `font` as it is, you could use `\kw` (or `\DeclarePseudoKeyword` for common cases):

while pigs don't fly
keep waiting

```
\begin{pseudo*}
\kw{while} pigs don't fly          \\\+
  keep waiting
\end{pseudo*}
```

\hd{*<name>*}(*<arguments>*)

Typesets a procedure signature, like `\pr`, but is intended for use as a *header* for a procedure, rather than a procedure call. The difference is that `\hd` wraps its contents in a `\multicolumn`, spanning two columns (i.e., both the label column and the main code column, but not any additional columns added using `preamble` or `begin-tabular`), using the preamble set with `hd-preamble`. For this to work, you need to use the star flag (*) to suppress the automatic insertion of the `prefix`:

```
ALGORITHM(x, y, z)
1  setup
2  while condition
3      iterative step
4  return result
```

```
\begin{pseudo}*
  \hd{Algorithm}(x, y, z)           \\
  setup                           \\
  \kw{while} condition             \\+
      iterative step               \\-
  \kw{return} result
\end{pseudo}
```

Note that the arguments are mandatory; in order to function properly, `\hd` must be *expandable*, and therefore cannot end with an optional argument, the way `\pr` does. If some package defines `\hd` before `pseudo` is loaded, `pseudo` will not overwrite it. The command will still be available, as `\pseudohd`.

hd-preamble = *<columns>* (no default)

Sets the preamble used by `\hd`. The result is available as the column type with name `\pseudohdpreamble`. (Note that this is the literal column name, and not a macro containing the name. See `preamble` for more information.) Initially, a single left-aligned column with `\pseudohpad` on either side (see page 62). If you introduce more columns in `preamble`, you might want to increase the number of columns in `hd-preamble` as well, or at least remove the right-hand `\pseudohpad`.

hl (takes no value)

Prepends `\pseudohl` to `bol`. Normally used with `beamer` (see page 10).

hl-color = *<color>* (no default, initially `black!12`)

Sets the color used by `\pseudohl` (available as `\pseudohlcolor`).

hpad = *<length>* (default `0.3em`, initially `0em`)

Horizontal padding on either side of the pseudocode. Useful, among other things when highlighting lines, to have some of the highlighting (i.e., row color) protrude beyond the text.

hsep = $\langle length \rangle$ (no default, initially 0.75em)

The space between the line labels and the code lines, i.e., between the two columns of numbered **pseudo** environments.

\id{ $\langle name \rangle$ }

Indicates an identifier, and is simply an alias for **\textit** wrapped in **\textnormal**. See also **\DeclarePseudoIdentifier**. This is a convenience for typesetting identifiers, and you may freely redefine it to whatever you prefer. If some package defines **\id** before **pseudo** is loaded, **pseudo** will not overwrite it. The command will still be available, as **\pseudoid**.

It might seem more natural to use **\mathit** (without **\tn**), but that may not give the desired results. First of all, special characters will not behave as if they're parts of a name:

foo – bar : baz

$$\mathit{foo-bar:baz}$$

This may be remedied, e.g., by using the (internal) command **\newmcodes@** from **amsopn**, but the kerning, spacing and font application in the result still leaves something to be desired:

foo-bar: baz

$$\mathit{\newmcodes@ foo-bar:baz}$$

Compare this to a simple **\textit**:

foo-bar:baz

$$\textit{foo-bar:baz}$$

The decision to use **\textit** means that you can't use, say, subscripts or the like as parts of an identifier, or mix in greek letters or other mathematical symbols. Though you can still easily typeset things like *foo- α* , you'll have to mix in the math mode more explicitly (in this case, **\id{foo- α }**). If some package defines **\id** before **pseudo** is loaded, **pseudo** will not overwrite it. The command will still be available, as **\pseudoid**.

idfont = $\langle font \rangle$ (no default, initially **\textit**)

Used to set **\idfont**, which is used as part of **\id**. May be set to take a single argument or none. Not restricted to actual font commands; you may also mix in **\textcolor** or the like.

\idfont

The command set by the **idfont** option. Used as part of **\id**.

indent-length = $\langle length \rangle$ (no default, initially empty)

How large each indentation step is. If this key is not specified, **indent-text** is used to calculate one the indent length instead.

indent-level = $\langle length \rangle$ (no default, initially 0)

Sets the current indentation level. This is most usefully set on **pseudo** environment, in concert with **start**.*

```
1 this is
2     the first part
```

This is some text interrupting the code.

```
3     this is the
4 second part
```

```
[
    \begin{pseudo}
        this is
        the first part
    \end{pseudo}
    \medskip \strut
    This is some text interrupting the code.
    \medskip
    \begin{pseudo}[start=3, indent-level=1]
        this is the
        second part
    \end{pseudo}
]
```

indent-mark = $\langle mark \rangle$ (default vertical line, initially empty)

A mark used to indicate the start of each step of indentation. The following example uses **indent-mark** = $\$ \cdot \$$:

```
1 repeat  $n - 1$  times
2   · for each edge  $uv$ 
3   ·   · update estimate for  $v$  via  $u$ 
4 for each edge  $uv$ 
5   · if estimate for  $v$  improves via  $u$ 
6   ·   · return FALSE
7 return TRUE
```

By default, the indent mark is a vertical line with the same height and depth as each row of the tabular used to typeset the pseudocode (i.e., it scales with **line-height**), so each indented block is indicated by a single

* The **\strut** here is just to even out spacing above and below the text, which doesn't have fixed-height lines like the pseudocode.

unbroken vertical line. The following example uses the `indent-mark` key with no argument:

```

1 repeat  $n - 1$  times
2 |   for each edge  $uv$ 
3 | |   update estimate for  $v$  via  $u$ 
4 for each edge  $uv$ 
5 |   if estimate for  $v$  improves via  $u$ 
6 | |   return FALSE
7 return TRUE

```

This default mark may be configured by using `indent-mark-color` and `indent-mark-width`.

indent-mark-color = $\langle color \rangle$ (no default, initially `lightgray`)

Sets the color to be used by the default `indent-mark`.

indent-mark-width = $\langle width \rangle$ (no default, initially `.6pt`)

Sets the width of the default `indent-mark`. The default value of `.6pt` corresponds to the `tikz` line width `semithick`.

indent-text = $\langle text \rangle$ (no default, initially `\pseudofont\kw{else}_`)

The size of each indentation step is set to the width of the $\langle text \rangle$. The default is set up so that code following on the same line as **else** will be properly aligned, as in:

```

if condition
    something
else something else

```

If you're not going to put code on the same line as **else**, for example, you might want a different indentation size. To set it to some specific length, you could use the `indent-length` key.

kw (takes no value)

Sets `font` to `\kwfont`.

\kw{ $\langle name \rangle$ }

Indicates a keyword. First wraps the argument in `\textnormal` and then adds `\kwfont`. See also `\DeclarePseudoKeyword`. This is a convenience for typesetting keywords, and you may freely redefine it to whatever you prefer. If some package defines `\kw` before `pseudo` is loaded, `pseudo` will not overwrite it. The command will still be available, as `\pseudokw`.

kwfont = $\langle font \rangle$ (no default, initially `\fontseries{b}\selectfont`)

Used to set `\kwfont`, which is used as part of `\kw`. May be set to take a single argument or none. Not restricted to actual font commands; you may also mix in `\textcolor` or the like. Note, however, that with the `kw` switch, you set `font = \kwfont`, which is then applied as a font-switching command for each entire line, taking no argument. If you provide an command requiring an argument, the `\kw` command will still work, but the `kw` switch won't:

foo bar

vs.

foo bar

```
\pseudoset{kw}
\begin{pseudo*}[kwfont=\textsf] % breaks kw option
  foo \kw{bar}
\end{pseudo*}
vs.\
\begin{pseudo*}[kwfont=\sffamily] % works with kw option
  foo \kw{bar}
\end{pseudo*}
```

The initial value isn't *quite* as straightforward as indicated, however. For more info, see `\kwfont`.

`\kwfont`

The command set by the `kwfont` option. Used as part of `\kw`. Its initial definition is *essentially* `\fontseries{b}\selectfont`, except the first time it's called (normally when evaluating the initial value of `indent-text`), it also runs a check to see if the font selection *worked*, as in some cases (such as in a default `beamer` presentation), the non-extended bold may not be available. In that case, it defaults to an extended bold (`\bfseries`) instead. At this point, the command is redefined to `\fontseries{b}\selectfont` or `\bfseries`, as appropriate (i.e., without this check). So, while `\kw{hello}` produces the non-extended **hello** in a default `LATEX` document, it yields the extended **hello** in a default `beamer` presentation. Perhaps more clearly, this is the result in plain `LATEX` (using `lmodern`):

while
while
while

```
\textbf{while}\\ % Extended
\kw{while}\\ % Keyword
{\fontseries{b}\selectfont while} % Non-extended
```

The same code results in the the following in `beamer`:

while
while
while

You'll also get a font warning,* though only once, as it's suppressed after the first occurrence, so the fact that the font selection doesn't work on the

* Of course, if you use a different font or theme, e.g., with the `beamer` command `\usefonttheme{serif}`, you may not have any issues to begin with.

last line isn't reported. Note, however, that the current implementation of `\kwfont` actually *piggybacks* on this warning to determine if the non-extended bold is available. This means that if you've tried (and failed) to use `\fontseries{b}` *before* the first use of `\kwfont`, the fallback (i.e., extended bold) won't be triggered.

Also note that `indent-text` (which will tend to be the first occurrence use of `\kwfont`) won't be evaluated (to determine `indent-length`) until you actually start a `pseudo` environment, so if you're *aware* that you don't have non-extended bold available, and you set `kwfont = \bfseries`, for example, there will be no attempt to use the non-extended version, and you won't get the font warning that the default implementation produces in that case.

label = $\langle \textit{commands} \rangle$ (no default, initially `\arabic*`)

1: **print** "Hello, label!"
2: **goto** 1

```
\pseudoset{kw, label=\footnotesize\arabic*:}

\begin{pseudo}
print \st{Hello, label!} \label{li:label}      \\
goto \tn{\ref{li:label}}
\end{pseudo}
```

Note that `\label` should be used in the actual code line, as here, and not in the number cell (which is generally not explicitly written, anyway).

As can be seen from the example, `\ref` is unaffected by `label`, and in many cases that's what you want—as apposed to, say, “**goto** 1:”. In some cases, however (especially when using one of the other formatting commands, such as `\alph` or `\roman`), you *do* want the reference format to reflect the original, or be similar in some way. To do that, you use the `ref` key.

label-align = $\langle \textit{column} \rangle$ (no default, initially `r`)

Used to specify the alignment of the `label` of each line. Whatever is provided is stored as a column type (named `\pseudolabelalign`), which is a part of the default `preamble`. In other words, beyond the basic `l` and `r` (for left- and right-justified), you can supply anything that would be valid as part of the preamble (possibly using functionality from the `array` package). If you want to get creative here, though, it might be easier to get the results you want by specifying your own `preamble` in full.

left-margin = $\langle \textit{length} \rangle$ (no default, initially `0pt`)

Sets the left margin of the `pseudo` environment, i.e., how far it is indented wrt. the surrounding text:

Lorem ipsum dolor sit amet:

- 1 consetetur sadipscing elitr
- 2 sed diam nonumy eirmod tempor

Invidunt ut labore et dolore magna.

```

Lorem ipsum dolor sit amet:

\begin{pseudo}[left-margin=1.25em]
consetetur sadipscing elitr \\
sed diam nonumy eirmod tempor
\end{pseudo}

Invidunt ut labore et dolore magna.
```

To have the environment indented as (the beginning of) any normal paragraph, you could use `left-margin = \parindent`. Note that `left-margin`, as well as the spacing above and below the `pseudo` environment, is turned off inside `\mbox` and the like:

I'm a livin' in a box
I'm a livin' in a cardboard box

```

\pseudoset{left-margin=1cm} % Won't affect box contents
\fbbox{\begin{pseudo*}
I'm a livin' in a box \\
I'm a livin' in a cardboard box
\end{pseudo*}}
```

Note that as opposed to `topsep` and `partopsep`, we are *not* working with one of the built-in list spacing commands; `\leftmargin` has no effect on this key (which is why the hyphenated naming style of other keys such as `label-align` or `indent-text` is also adopted for `left-margin`). See also `compact`.

line-height = $\langle factor \rangle$ (no default, initially 1)

The $\langle factor \rangle$ with which to multiply the ordinary line height. For simple, sparse pseudocode, the ordinary line height works well, but if your code gets too crowded with text and notation, you may wish to increase `line-height`. To emulate, e.g., the `\jot` set by `amsmath` (which is `0.25\baselineskip`), you could use `1.25`, though even `1.1` should help in many cases.

\nf

Switch to the normal font (i.e., without bold or italics, etc.). If some package defines `\nf` before `pseudo` is loaded, `pseudo` will not overwrite it. The command will still be available, as `\normalfont`. See also `\tn`.

partopsep = $\langle length \rangle$ (no default, initially `\partopsep`)

Sets a `pseudo`-local copy of `\partopsep` for use in vertical spacing above and below the `pseudo` environment. See also `compact`.

pause (takes no value)

Equivalent to `eol-append = \pause` (see section 2).

pos = $\langle depth \rangle$ (no default, initially `t`)

Specifies the vertical position of the `pseudo` environment, i.e., whether it should be vertically aligned on the top (`t`) or bottom (`b`) row, or be vertically centered (no value). This is equivalent to the (optional) `pos` argument to `tabular`, and is in fact supplied to the internal `tabular` environment. The initial value is `t`, which makes sure the spacing above is consistent, regardless of the depth of the previous line. Here are two examples, set side by side:

$$x \qquad f(x)$$

1	foo	1	frozz
2	bar	2	bozz

The `pseudo` environments are properly aligned. If, instead, we set `pos = {}`, they will not be, because $f(x)$ has more depth than x .*

$$x \qquad f(x)$$

1	foo	1	frozz
2	bar	2	bozz

If `compact` is set to `true`, `pos` is automatically emptied like this—a behavior which can, of course, be overridden:

	foo	
Lorem	foo	ipsum bar dolor foo.
	bar	bar

```

    Lorem
    \begin{pseudo*}[compact]
      foo \\\ bar
    \end{pseudo*}
    ipsum
    \begin{pseudo*}[compact, pos=b]
      foo \\\ bar
    \end{pseudo*}
    dolor
    \begin{pseudo*}[compact, pos=t]
      foo \\\ bar
    \end{pseudo*}.

```

\pr{ $\langle name \rangle$ }($\langle arguments \rangle$)

* This was the behavior in older versions of `pseudo`.

Indicates a procedure name, such as `QUICKSORT`, and is initially more or less an alias for `\cn`. The optional arguments (in parentheses) are typeset in math mode, so `\pr{Quicksort}(A,p,r)` yields $\text{QUICKSORT}(A, p, r)$. See also `\DeclarePseudoProcedure`. This is a convenience for typesetting procedure names, and you may freely redefine it to whatever you prefer. If some package defines `\pr` before `pseudo` is loaded, `pseudo` will not overwrite it. The command will still be available, as `\pseudopr`.

preamble = $\langle columns \rangle$ (no default)

Sets the preamble to be used by the internal `tabular`. The result is available as the column type with name `\pseudopreamble`. (Note that this is the literal column name, and not a macro containing the name. Initially, `pseudo` uses a `tabular` as redefined by the `array`, which prevents the expansion of whatever is provided as its preamble, and so we supply the preamble in the form of a single “column” instead.) For the default value, see the actual implementation on page 61 as well as the explanation in section 4.1.

prefix = $\langle commands \rangle$ (no default)

This is the text inserted at the beginning of the following line by `\` (and by `\begin{pseudo}`), unless you use the star (*) flag. Unless modified, it inserts the code necessary to label the line and to move into the second column, where the actual code is inserted by the user. For the default value, see the actual implementation on page 62 as well as the explanation in section 4.1.

prevdepth = $\langle depth \rangle$ (no default, initially `.3\baselineskip`)

This value is used to properly adjust the vertical distance to any following text, by setting `\prevdepth` to $\langle depth \rangle$, unless `compact` is set to `true`. In general, it should not be necessary to change its value.*

prfont = $\langle font \rangle$ (no default, initially `\cnfont`)

Used to set `\prfont`, which is used as part of `\pr`. May be set to take a single argument or none. Not restricted to actual font commands; you may also mix in `\textcolor` or the like.

\prfont

The command set by the `prfont` option. Used as part of `\pr`.

`\begin{pseudo}` [$\langle options \rangle$] * $\langle overlay specification \rangle$ [$\langle line options \rangle$]
 $\langle pseudocode \rangle$
`\end{pseudo}`

The actual environment in which the pseudocode is typeset. The $\langle options \rangle$ are local to the environment, while the $\langle line options \rangle$ are local to the following line (in the same manner as those set in `\`; i.e., only some will actually have any effect). The star (*) and $\langle overlay specification \rangle$ act just like those on `\`. Note that if you wish to specify $\langle line options \rangle$ without the star or the $\langle overlay specification \rangle$, you need to supply at least an empty pair of brackets for the global options:

* In previous versions, `\prevdepth` was not set. To get the old behavior, set `prevdepth = 0pt`.

```
1 First line
2 Second line
```

vs.

```
1 First line
2 Second line
```

```
[
    \begin{pseudo}[] [hl]
    First line \
    Second line
    \end{pseudo}
    vs.\
    \begin{pseudo}[hl]
    First line \
    Second line
    \end{pseudo}
]
```

There are no +/- flags here, unlike for `\`; if needed, you can use `indent-level`.

```
\begin{pseudo*}[\langle options \rangle] * <\langle overlay specification \rangle> [\langle line options \rangle]
  \langle pseudocode \rangle
\end{pseudo*}
```

An unnumbered version of the `pseudo` environment. Equivalent to `pseudo`, but with the `starred` style applied (see page 70). Unless this style is altered, this means that the label column is removed from the preamble, and the prefix is reduced to only `bol`.

`\pseudobol`

The command set by the `bol` option. Used as part of `\pseudoprefix`.

`\pseudodefstyle{\langle name \rangle}{\langle options \rangle}`

Used to define “styles” or meta-keys, i.e., shortcuts for setting several keys to given values (used, e.g., to define `starred`). The `\langle name \rangle` is simply the name of the new meta-key, and the `\langle options \rangle` are just what you’d provide to, e.g., `\pseudoset`.

`\pseudoeol`

The command set by the `eol` option. Used as part of `\`. It is inserted between lines, but not after the last one.

`\pseudoeq`

Similar to `\pseudoslash`. Switches the definition of `\=` to the one used by `pseudo`. Useful if `\=` reverts to its original definition in some context (see `\==`).

`\pseudofont`

The command set by the `font` option. Used as part of `\pseudosetup`. It is used to set up the font for each pseudocode line. (See also `kw`.)

`\pseudohl`

This is the command inserted as `bol` by the `hl` switch. Initially, it's just a `\rowcolor` using the color set by `hl-color`, but you could redefine it to whatever you wish.

`\pseudohpad`

Used on the left- and right-hand sides of `preamble`. Conceptually, it inserts the horizontal space specified by `hpad`. To play nice with `\rowcolor`, however, it is not used in a `@{...}` column; rather, it's placed in `>{...}` and `<{...}` modifiers, and the actual space inserted has `\tabcolsep` subtracted.

`\pseudoindent`

The command set by the `indent-length` and `indent-mark` options. Used in `\pseudosetup`. More precisely, `indent-length` is stored textually, and is converted to the length `\pseudoindentlength` when entering a `pseudo` environment (so that units like `em` and `ex` adapt to the current font). If no `indent-mark` is set, the `\pseudoindent` command then inserts a horizontal space of length `\pseudoindentlength × current indent level`. Otherwise, one `indent-mark` and a horizontal space of length `\pseudoindentlength` is inserted for each level of indentation up to the current indentation level. (This horizontal space is measured from the left edge of the `indent-mark`.)

`\pseudolabel`

The command set by the `label` option. Used as part of `\pseudoprefix`.

`pseudoline`

Counter for pseudocode lines. See also `*`.

`\pseudopos`

The command set by the `pos` option. Used as part of the initial value of `begin-tabular`.

`\pseudopreamble`

The command set by the `preamble` option. Used as part of the initial value of `begin-tabular`.

`\pseudoprefix`

The command set by the `prefix` option. Used as part of `\\`.

`\pseudosavelabel`

Used as part of `\pseudosetup` to save the `pseudoline` counter for use in `\label` and `\ref`. The `pseudoline` counter is *incremented* as part of the `\pseudolabel` command, but that's done using a plain `\stepcounter`, as any use of `\label` will presumably be placed in the pseudocode line (i.e., the next column). To save the value there, `\pseudosavelabel` first *decrements* the counter, and then uses `\refstepcounter`.

`\pseudoset{<options>}`

Used to set the configuration keys of the `pseudo` package (using `l3keys` with `pseudo` as the module). These may also be set as optional arguments to the `pseudo` and `pseudo*` environments. For example, if you'd like to switch to `\rm` as your base font, you could use `\pseudoset{font = \rm}`.

\pseudosetup

The command set by the **setup** option. Used as part of the **preamble**.

Not to be confused with \pseudoset.

\pseudoslash

Command similar to the **\arrayslash** of the **array** package. Switches the definition of **** to the one used by **pseudo**. Useful if you've used some code that modifies **** for its own purposes (such as **\raggedleft** or the like).

ref = *<commands>* (initially empty, default **\pseudolabel**)

Shortcut for setting the **\thepseudoline** command. If used without arguments, it will use the value supplied to **label**.

(A) print "Hello, ref!"

(B) goto A

```
\pseudoset {
  label = (\textsc{\alph*}),
  ref   = \Alph*,
  hsep  = .5em
}

\begin{pseudo}
print \st{Hello, ref!} \label{li:ref} \\\
goto \tn{\ref{li:ref}}
\end{pseudo}
```

setup = *<commands>* (no default)

The setup part of each pseudocode line: Save the line counter (using the **\pseudosavelabel** command), insert the proper indentation (with **\pseudoindent**) and switch to the correct font (**\pseudofont**).

Rather than setting **setup** directly, you may wish to add commands using **setup-append** or **setup-prepend**.

setup-append = *<commands>* (no default)

Locally appends *<commands>* to **setup**.

setup-prepend = *<commands>* (no default)

Similar to **setup-append**, except that *<commands>* are added to the *beginning* of **setup**.

\st{<string>}

Typesets *<string>* with added quotes using **\stfont**. (The entire thing is wrapped in **\textnormal**.) For example, **print \st{42}** yields:

print "42"

See also `\DeclarePseudoString`. This is a convenience for typesetting strings, and you may freely redefine it to whatever you prefer. If some package defines `\st` before `pseudo` is loaded, `pseudo` will not overwrite it. The command will still be available, as `\pseudost`.

st-left = $\langle text \rangle$ (no default, initially ‘ ‘)
Text or commands inserted at the start of a string, when using `\st`.

st-right = $\langle text \rangle$ (no default, initially ‘ ’)
Text or commands inserted at the end of a string, when using `\st`.

starred (takes no value)
The style (defined by `\pseudodefestyle`) used by the `pseudo*` environment. You may modify this (again using `\pseudodefestyle`) if you wish.

start = $\langle number \rangle$ (no default, initially 1)
Sets the starting line number:

10 Maybe we’re continuing from some earlier code?
11 Anyway, let’s keep going!

```

\begin{pseudo}[start=10]
Maybe we’re continuing from some earlier code?  \\\
Anyway, let’s keep going!
\end{pseudo}

```

See also `indent-level`.

stfont
Used to set `\stfont`, which is used as part of `\st`. May be set to take a single argument or none. Not restricted to actual font commands; you may also mix in `\textcolor` or the like.

\stfont
The command set by the `stfont` option. Used as part of `\st`.

\tn{ $\langle text \rangle$ }
An alias for `\textnormal`, to break out of the font set using the `font` key, for inserting ordinary prose between the keywords. For example, to get the result “**for** every node $v \in V$ ”, one might write:

`for \tn{every node} $v\in V$`

This is equivalent to using `\textnormal{every node}`. If some package defines `\tn` before `pseudo` is loaded, `pseudo` will not overwrite it. The command will still be available, as `\textnormal`.

topsep = $\langle length \rangle$ (no default, initially `\topsep`)
Sets a `pseudo`-local copy of `\topsep` for use in vertical spacing above and below the `pseudo` environment. See also `compact`.

unknown

Unknown keys are checked for `beamer` overlay specifications. That is, if an unknown key has the form

$$\langle name \rangle \langle overlay\ specification \rangle = \langle value \rangle$$

then it does not trigger an error, but, if `beamer` is used, is rewritten to:

$$\backslash only \langle overlay\ specification \rangle \{ \backslash pseudoset \{ \langle name \rangle = \langle value \rangle \} \}$$

If `beamer` is *not* used, the key is simply ignored. Note that because of current limitations on how keys are handled, unknown keys cannot have defaults, and so there is no way to insert a marker for when no value is provided, which could be used to determine whether to use `\pseudoset{\langle name \rangle = \langle value \rangle}` or simply `\pseudoset{\langle name \rangle}`. Instead, if an empty value is provided to the unknown key, that is treated in the same way as when the key is used without a value, resulting in `\pseudoset{\langle name \rangle}` rather than `\pseudoset{\langle name \rangle = }`.

5 But how do I...

Some functionality is not built in, but is still fairly easy to achieve. Some streamlining may be added in future versions.

5.1 ... prevent paragraph indentation after pseudo?

If you want to keep the pseudocode as part of a surrounding paragraph, you could have it not start its own, i.e., not have an empty line before it. This will reduce the amount of spacing as well; if you'd rather have that reduced, you could simply drop the empty line *after* the environment:

```
Text before

\begin{pseudo}
  pseudocode
\end{pseudo}
%
Text after
```

The effect would then be the following:

```
1 pseudocode
```

No indentation here, and normal spacing. If, however, you wish to suppress indentation after *all* instances of `pseudo`, you could use the `noindentafter` package, as follows:

```
\usepackage{noindentafter}
\NoIndentAfterEnv{pseudo}
```


If you wish to override this, and indent a given paragraph after all, you can simply use the `\indent` command.

5.2 ... get log-like functions?

There's no built-in command for math-roman function names, as used in `log` and `sin`, etc. (other than just setting `fnfont`, if you want it everywhere). If you wish to define your own, you could use `\operatorname` or `\DeclareMathOperator`. For example:

```
1 if my-func  $x == 1$ 
2    $y = \text{my-func}(z + 1)$ 
```

```
% In preamble:
% \usepackage{amsmath}
% \DeclareMathOperator{\MyFunc}{my-func}
\begin{pseudo}[kw]
if $\MyFunc x == 1$ \+
    $y = \MyFunc(z + 1)$
\end{pseudo}
```

The spacing is then correct whether you enclose the arguments in parentheses or not.

5.3 ... unbold punctuation?

If you use the `kw` key, all pseudocode not in math mode will end up using the keyword font (`\kwfont`), which initially is bold. Though some *do* typeset, e.g., grouping braces in boldface, you might not want to do that; the same goes for, say, line-terminating semicolons. The `theoremfont` option of, e.g., `newtx` does something similar (for italics), but uses a custom font for that. Packages like `emrac` rely on straightforward textual substitution, replacing certain characters with marked-up ones, but the way things are set up at the moment, our font command won't have access to the entire line when it's executed.

If you're adventurous, it's not hard (using the `xparse` argument type `u`) to make a version that *does* gobble up the entire line, up to and including `\` (and you could then use the regular expression functionality from `expl3`, presumably also reinserting `\`). A simpler solution is to just use `\DeclarePseudoNormal`. Here's an example based on pseudocode from Knuth [4]:

```
procedure printstatistics;
begin integer  $j$ ;
    write("Closed sets for rank",  $r$ , ":", " ");
     $j := L[h]$ ;
    while  $j \neq h$  do
        begin writeln( $S[j]$ );  $j := L[j]$  end;
end;
```

```

% In preamble:
% \usepackage{mathtools}
\let\gets\coloneqq

\pseudoset{kw, indent-length=2em, line-height=1.1}

\DeclarePseudoNormal \; ;

\begin{pseudo*}
procedure \id{printstatistics}\;           \\\
begin integer $j$;                         \\\+
  $\id{write}(\st{Closed sets for rank}, r, \st{:})$; \\\
  $j \gets L[h]$;                          \\\
  while $j \neq h$ do                       \\\+
    begin $\id{writeon}(S[j])$; $j \gets L[j]$ end; \--
  end\;
\end{pseudo*}

```

If you'd really like to avoid the extra backslashes, you could make the relevant punctuation active (though that's probably a bit risky; make sure to only do it locally, at the very least):

begin integer j ;

```

\DeclarePseudoNormal \semi ;

\catcode'\;=\active
\let;\semi

\begin{pseudo*}[kw]
  begin integer $j$; % Look! The semicolon isn't bold!
\end{pseudo*}

```

5.4 ... use tabularx?

You can use other tabular packages such as `tabularx` via `begin-tabular` and `end-tabular`. Let's say, for example, that you wish to extend the `pseudo` environment to fill out the entire line, and set up a new column for comments. You could achieve that as follows:

COUNTING-SORT(A, k)	Find positions by counting
1 C = an array of k zeros	Element frequencies
2 for $i = 1$ to $A.length$	Count all elements
3 ...	Etc.

```

\pseudodefestyle{fullwidth}{
  begin-tabular =
  \tabularx{\linewidth}[t]{@{}
    r                                     % Labels
    >{\pseudosetup}                     % Indent, font, ...
    X                                   % Code (flexible)
    >{\leavevmode\small\color{gray}}    % Comment styling
    p{0.45\linewidth}                  % Comments (fixed)
    @{}},
  end-tabular = \endtabularx,
  setup-append = \pseudoeq
}
\begin{pseudo}[kw, fullwidth, line-height=1.1]*
  \hd{Counting-Sort}(A, k) & Find positions by counting \\
  $C = \tn{an array of $k$ zeros}$ & Element frequencies \\
  for $i = 1$ to $A.\id{length}$ & Count all elements \\
  $\dots$ & Etc.
\end{pseudo}

```

Note that using the `\color` command in a `>{...}` modifier with a `p` column places the text in a new paragraph, on the next line; you'll need to insert `\leavevmode` or the like to prevent that. This is true also of normal `tabular` environments. Also note that `tabularx` environments with `X` columns don't interact nicely with `\=`; so if you wish to use `\==`, you can reassert the definition by adding `>{\pseudoeq}` before each column.*

See the `tabularx` documentation (page 4) for an explanation of why we can't use `\begin{tabularx}` and `\end{tabularx}`. Also note that because `tabularx` passes its contents as the argument to a macro, the parsing `pseudo` uses to determine if `\\` is at the end of the last line doesn't work; if you add `\\` at the end here, you'll introduce an empty line.

For simplicity, I've used `@{}` to remove space on either side. For `hpad` to work, you should use `>{\pseudohpad}` and `<{\pseudohpad}` instead, as in the standard `preamble` (see page 61). To keep things configurable, you might also want to use `\pseudolabelalign`, rather than `r`.

5.5 ... get tab stops?

Some packages, such as `clrscode3e`, use an actual `tabbing` environment internally. While this may be a bit brittle (e.g., creating problems if you wish to insert your pseudocode into a `tikz` node—one of the goals of `pseudo`), it does mean that you can use the `tabbing` command `\>` manually, to align various construct.

If all your `tabbing` is done *before* the text on a given code line, you can achieve this in `pseudo` as well, by using the `+` and `-` modifiers. (For example, the tab stops in `clrscode3e` are set at fixed intervals, just like in `pseudo`.) But what if you'd like to align something that comes later, such as comments after code

* You can also, of course, just use `\eqs` instead.

lines? You can't simply use `\hspace`, of course, unless the code lines themselves have exactly the same length.

One solution is to use an additional column, as discussed in section 5.4, but you could also make creative use of the `\rlap` command, which prevents its contents from taking up horizontal space:*

~~This is some text~~ more

```
\noindent\rlap{This is some text}%
And here is some more
```

By using `\rlap` on the code lines in question, you can insert `\hspace` that begins at the beginning of the code line (here with an example convenience command defined using `xparse`):

```
1  $x = 42$  (first comment)
2  $y = \sin x$  (second comment)
```

```
\NewDocumentCommand \C { +u{ /* } +u{ */ } } {%
  \rlap{#1}\hspace{3cm}\ct{#2}\%
}
\begin{pseudo}
\C $x = 42$ /* first comment */
\C $y = \sin x$ /* second comment */
\end{pseudo}
```

See also the discussion of the `\ct` command for ideas on typesetting comments. If you wish to align things across different indentation levels, you'll have to add or subtract multiples of `\pseudoindentlength` (see `\pseudoindent`).

5.6 ... use horizontal lines?

Many opt for a table-like appearance when typesetting algorithms, with horizontal lines above and below, and generally a header row on top. While this may be part of a surrounding floating environment (see section 3), you may also wish to include such lines in your actual pseudocode. In this case, you can simply use existing `tabular`-based tools such as `booktabs`, making sure to suppress the `pseudo` prefix using the star flag (*):

 $\text{BORŮVKA}(G, w)$

```
1 while  $E(G)$  is not empty
2   for each  $u \in V(G)$ 
3     add light  $uv \in E(G)$  to  $T$ 
4   for each  $e \in T$ 
5     contract  $e$ 
```

* Note that `\rlap` doesn't start a new paragraph, which is why I use `\noindent`, here. You could replace `\noindent\rlap{...}` with `\makebox[0pt][l]{...}`. This isn't an issue in `pseudo` code lines, however.

```

% \usepackage{booktabs}
\begin{pseudo}*
\toprule

\hd{Bor\r{u}vka}(G, w)                                \\\
                                                         \%

[bol=\midrule]

\kw{while} $E(G)$ is not empty                          \\\+
\kw{for} each $u\in V(G)$                                \\\+
  add light $uv \in E(G)$ to $T$                          \\\-
\kw{for} each $e \in T$                                    \\\+
  contract $e$                                             \\\*

\bottomrule
\end{pseudo}

```

Rather than `\\[bol=\midrule]`, you could also have used `*`, followed by `\midrule\pseudoprefix`. (Note that the paragraph break between `\\` and its argument has been commented out.)

5.7 ... handle object attributes?

In the `clrscode3e` package, you'll find an assortment of commands for handling object attributes such as *A.length*. The manual says (here with emulated kerning of the dot operator):

You might think you could typeset *A.length* by `$A.\id{length}$`,
 but that would produce *A.length*, which has not quite enough space
 after the dot. (page 3)

However, this is a font issue, more than anything. If, for example, if you want Times New Roman (like Cormen et al.) and use `mathptm`, you at times run into the problem described; with `newtx` it's less pronounced. With other fonts (e.g., `fourier`, `mathpple` or `newtxmath` with `libertine`), or even without any font packages (or possibly using `lmodern`), the kerning works just fine.

In general, then, I suggest you try to use `$A.\id{length}$` and the like, and see if the result is satisfactory:

v.prev.next = v.next

```

$ v.\id{prev}.\id{next} = v.\id{next}$

```

If you *do* need to adjust the kerning (with `\mkern` commands or perhaps using `microtype`), you may of course do so, but `pseudo` does not (at present) include any special attribute lookup commands that do it for you.

5.8 ... indicate blocks with braces or the like?

Some packages (such as `algorithm2e`) have support for using vertical lines to indicate the block structure; `pseudocode` uses large braces. In `pseudo`, there is support for using an `indent-mark`, for which the default is a semithick, gray vertical line (see page 4). However, by using `tikz`, you could draw all kinds of indentation decorations.

You could, for example, add a `node` at the start of each code line, containing an `\@arstrut`, the `(array)` strut used to indicate the extent of a tabular row:

```
% \usepackage{xparse,tikz}
% \usetikzlibrary{decorations.pathreplacing,calligraphy}
\makeatletter
\NewDocumentCommand \pseudoanchor { m } {%
  \tikz[baseline, overlay, remember picture]
    \node[anchor=base, inner sep=0] (#1) {\@arstrut};%
  \ignorespaces
}
\makeatother
```

You can then use the resulting nodes to draw braces or lines or whatever. First some example setup:

```
\pseudoset{
  kw,
  indent-length = 3.5em,
  setup-append = {\pseudoanchor{L-\arabic*}}
}
\tikzset{
  braces/.style =
    {thick, decoration = {calligraphic brace, raise=.2em}},
  label/.style =
    {midway, left=3em, anchor=west, font=\strut\kwfont}
}
```

You would then get something like the following:

```
1 if  $x < y$ 
2   then  $\begin{cases} x = y \\ y = 0 \end{cases}$ 
3
```

```

\begin{pseudo}
if $x < y$                                \\\+
    $x = y$                                \\\
    $y = 0$
\end{pseudo}
\tikz[overlay, remember picture, braces] {
    \draw[decorate] (L-3.south) -- (L-2.north) node[label] {then};
}

```

If multiple blocks are closed at the same time, the bottom coordinates could be things like (L-2.north |- L-3.south) instead. To adjust the end points, you could also use things like $(\$(L-3.south)+(0,.2em)\$)$.

The actual drawing of the brace (or line or whatever) isn't automated here, of course. This could be done by some hook triggered by the `-` flags in `\\`. If it turns out there's a demand for something like that, I might add it in a future version.

5.9 ... use pseudo with older T_EX distributions?

As mentioned in the introduction, I've tried to make `pseudo` work with at least *somewhat* outdated T_EX distributions. In these cases, the package itself won't be available as part of the distribution, of course, but you can simply download the file `pseudo.sty` and place it in the directory where you're compiling your document (or anywhere else where your L^AT_EX executable can find it).

However, there may be cases where this just doesn't work, such as when submitting to a publisher with a really old setup.* In that case, the simplest solution is probably to use the `standalone` package to produce individual PDFs of your algorithms, and then to include those in your document. Then you can submit the PDFs rather than the L^AT_EX, so that the pseudocode need not be compiled on the old system. Each algorithm could go in a file like this:

```

\documentclass{standalone}
\usepackage{pseudo}
\begin{document}
\begin{pseudo}
...
\end{pseudo}
\end{document}

```

Let's say this is compiled to `algo1.pdf`. You then include this file:

* If possible, though, feel free to file an issue or provide a pull request to address the issue.

```

\documentclass{article}
\usepackage{graphicx} % For \includegraphics
...
\begin{document}
... sanctus est Lorem ipsum dolor sit amet:

\medskip\noindent
\includegraphics{algo1}

\smallskip
Lorem ipsum dolor sit amet, consetetur sadipscing ...
\end{document}

```

Of course, you can adjust the spacing (e.g., using `\vspace` or the like) to your liking. Using this method, you can achieve results essentially identical to if you compiled the pseudocode directly as part of the document. Of course, you won't have access to other functionality (such as `\DeclarePseudoIdentifier` or the like) for use in the main tex, but most of that should be possible to emulate by hand (possibly peeking at the implementation in section 6).

5.10 ... use a header with no arguments?

Normally, `\hd` has a mandatory set of arguments; at the very least, you'll need to supply the parentheses:

NO-ARGUMENTS()
1 ... but still with parentheses

```

\begin{pseudo}*
\hd{No-Arguments}() \\\
\dots\ but still with parentheses
\end{pseudo}

```

This is because `\hd` has to be fully expandable to be able to insert the requisite `\multicolumn`, and then it cannot have any (final) optional arguments. If you'd like, though, you can just use `\multicolumn` yourself (see also `hd-preamble`):

NO-ARGUMENTS
1 ... and no parentheses!

```

\begin{pseudo}*
\multicolumn{2}{\pseudohdpreamble}
{\pr{No-Arguments}} \\\
\dots\ and no parentheses!
\end{pseudo}

```


6 Implementation

Note: In the following, `_@@` and `@@` represent an internal prefix (`__pseudo`), the same way they do with `l3docstrip`.

First, we just define some metadata:

```
\def \pseudoversion {1.1.3}
\def \pseudodate    {2022/06/28}
```

The `pseudo` package is implemented using experimental L^AT_EX 3, so we start by importing `expl3`:

```
\RequirePackage{expl3}
```

Then we're ready start the package:

```
\ProvidesExplPackage
  {pseudo}
  {\pseudodate}
  {\pseudoversion}
  {Straightforward pseudocode}
```

Tools for defining user commands:

```
\RequirePackage{xparse}
```

For defining `tcolorbox` styles, without importing `tcolorbox`:

```
\RequirePackage{pgfkeys}
```

The `pseudo` environment is built upon tabular functionality, and we're using some extensions:

```
\RequirePackage{array, xcolor, colortbl}
```

Though *most* keys aren't available as `\usepackage` arguments, we still use the mechanism:

```
\RequirePackage{l3keys2e}
```

Inside the `pseudo` environment, `*` is an alias for `pseudoline`. To perform the proper aliasing, we use `aliascnt`:

```
\RequirePackage{aliascnt}
```

As part of the initial setup, we also record whether we're part of a `beamer` presentation; this will affect the overlay functionality:

```

\bool_new:N \c_@@_beamer_bool
\@ifclassloaded{beamer}
  {\bool_set_true:N \c_@@_beamer_bool}
  {\bool_set_false:N \c_@@_beamer_bool}

```

We’re now ready to begin the actual implementation.

6.1 Variable declarations

Many variables are created as needed by various `set` commands, but some are declared initially. First, we create a plain-vanilla \LaTeX counter for the line number, as well as an outer one for the environment, the latter just to avoid duplicate labels:

```

\newcounter{pseudoenv}
\newcounter{pseudoline}[pseudoenv]

```

Eventually, we’ll be saving the line counter so that `\label` commands will work, but we’ll only do so if the counter has *changed* (again, to avoid duplicate labels). To determine whether, in fact, it has, we keep the previous one we saved:

```

\int_new:N \g_@@_last_saved_line_int

```

Normally a counter is just saved when it’s incremented (with `\refstepcounter`), but in our case, we want to increment and typeset it based on a (potentially) user-configured `label`, and then actually save it and make it the target of `\label` commands in a *different scope* (i.e., the next cell in the tabular row).

The indent size is set through the configuration key `indent-length` (or indirectly through `indent-text`), while the current indent level is manipulated by `\;`; their product determines the actual length by which the current line is indented. The initial indent level may be set using `indent-level`.

```

\dim_new:N \pseudoindentlength
\int_new:N \g_@@_indent_level_int
\int_new:N \l_@@_initial_indent_level_int

```

6.2 Utilities

Variants. First, let’s just generate a couple of expansion variants we’ll need of some standard commands. (I’m using the `\q_no_value` machinery rather than `\c_novalue_tl` for compatibility with older \TeX distributions.)

```

\cs_generate_variant:Nn \quark_if_no_value:nTF { VTF }
\cs_generate_variant:Nn \tl_if_novalue:nTF { VTF }
\cs_generate_variant:Nn \tl_set:Nn { Ne }

```

Defining columns. The `preamble` is is configurable, but the `array` package makes sure it doesn’t expand any part of its preamble. One way of inserting a

dynamically generated one is to simply define it all as a single column type. To avoid getting an error when overwriting this definition through the configuration, we'll also need to be able to *un*-define column types:

```
\cs_new:Nn \@@_undef_col:n {
  \tl_set_eq:cN { NC@find@ \token_to_str:N #1 } \scan_stop:
}
```

Note that the implementation specifically targets the `array` package. The following command then will either define or *re*-define a column type:

```
\cs_new:Nn \@@_def_col:nn {
  \@@_undef_col:n { #1 }
  \newcolumnntype { #1 } { #2 }
}
```

Defining commands. This command creates a new command with a `pseudo` prefix, and defines the prefixless version as well, *if the name is available* (i.e., undefined):

```
\cs_new:Nn \@@_meta_new_cmd:NNnn {
  \tl_set:Nn \l_tmpa_tl {pseudo \cs_to_str:N #2}
  \exp_args:Nc
    #1 \l_tmpa_tl #3 {#4}
  \cs_if_free:NT #2 {\cs_gset_eq:Nc #2 \l_tmpa_tl}
}

\cs_new:Nn \@@_new_cmd:Nnn {
  \@@_meta_new_cmd:NNnn
  \NewDocumentCommand #1 {{#2}} {
    #3
  }
}

\cs_new:Nn \@@_new_ecmd:Nnn {
  \@@_meta_new_cmd:NNnn
  % \NewExpandableDocumentCommand #1 {{#2}} {
  % Replaced for compatibility:
  \def #1 {#2} {
    #3
  }
}
```

This is for defining commands that declare styled shortcuts:

```
\cs_new:Nn \@@_new_dec:nn {
  \tl_set:Nn \l_tmpa_tl { DeclarePseudo #1 }
  \exp_args:Nc
    \DeclareDocumentCommand \l_tmpa_tl { mm } {
      \DeclareDocumentCommand ##1 { } {
        \use:c { #2 } { ##2 }
      }
    }
```

```

    }
}

```

You use this with a capitalized name for the kind of thing you’re declaring, and the name of the style command to use. For example,

```
\@@_new_dec:nn{Keyword}{kw}
```

will create the command `\DeclarePseudoKeyword`, which takes a `csname` and a word, and binds the `csname` as a shortcut for the word, properly styled as a keyword.

Argument parsing. In processing the multiple `+` and `-` arguments to `\`, we’ll gobble up one character at a time, each time performing some action. We also supply code to be performed once we’re done.

```

\cs_new:Nn \@@_per_char:nnn {
  \peek_charcode_remove:NTF { #1 } {
    #2 % body
    \@@_per_char:nnn{#1}{#2}{#3}
  } {
    #3 % tail
  }
}

```

Indentation. The indent size (i.e., the length of a single step of indentation) is either set directly through `indent-length`, or indirectly through `indent-text`. The latter is there the default is provided, but `indent-text` is only used if there is no `indent-length`.

```

\cs_new:Nn \@@_set_indent_length: {

  \quark_if_no_value:VTF \l_@@_indent_length_tl {
    \hbox_set:Nn \l_tmpa_box { \l_@@_indent_text_tl }
    \dim_set:Nn \pseudoindentlength { \box_wd:N \l_tmpa_box }
  } {
    \dim_set:Nn \pseudoindentlength \l_@@_indent_length_tl
  }

}

```

Note that the configured indent length is stored in a `tl`, which is expanded in the `pseudo` environment.

The indent size is subsequently used by the `indent` command, which takes the number of indentation steps as its only argument. If no `indent-mark` is set, it just inserts an appropriate horizontal space. Otherwise, it iterates over the indent levels, inserting one indent marker for each level.

```

\cs_new:Nn \@@_indent:N {
  \tl_if_novalue:VTF \l_@@_indent_mark_tl {
    \skip_horizontal:n{ \pseudoindentlength * #1 }
  } {

```

```

\group_begin:
\color{\l_@@_indent_color_tl}
\int_step_inline:nn { \g_@@_indent_level_int } {
  \smash{\hbox_overlap_right:n {
    \l_@@_indent_mark_tl
  }}
  \skip_horizontal:n{ \pseudointentlength }
}
\group_end:
}
\ignorespaces
}

```

Counter copying. Inside the `pseudo` environment, we want `*` to be a duplicate of `pseudoline`, for convenience. This requires a bit of work. We use the `aliascnt` package to deal with much of the book-keeping, but in order for `\newaliascnt` to work whenever a counter already exists, we need to undefine it first. (Here we’re relying on the internal L^AT_EX convention of using `c@` as a prefix to counter names.)

```

\cs_new:Nn \@@_drop_ctr:n {
  \cs_undefine:c { c@ #1 }
}

\cs_new:Nn \@@_copy_ctr:nn {
  \@@_drop_ctr:n { #1 }
  \newaliascnt { #1 } { #2 }
}

\cs_new:Nn \@@_star_setup: {
  \cs_if_exist:cT { c@ * } {
    \@@_copy_ctr:nn { @@_orig_* } { * }
  }
  \@@_copy_ctr:nn { * } { pseudoline }

  \group_insert_after:N \@@_star_reset:
}

\cs_new:Nn \@@_star_reset: {
  \cs_if_exist:cT { c@ @@_orig_* } {
    \@@_copy_ctr:nn { * } { @@_orig_* }
    \cs_undefine:c { c@ @@_orig_* }
  }
}

```

Label saving. In the body of each line, we make sure to save the counter, so it’s available for the `\label` command. We’ve already incremented `pseudoline` with `\stepcounter` in the label, so we first need to decrement it before we again increment it, this time with `\refstepcounter`. However, we only do so if the counter actually *was* incremented, i.e., if it’s different from the last one we saved.

```

\cs_new:Nn \@@_save_label: {

  \int_set:Nn \l_tmpa_int {\arabic{pseudoline}}

  \int_compare:nF {\l_tmpa_int = \g_@@_last_saved_line_int} {
    \addtocounter{pseudoline}{-1}
    \refstepcounter{pseudoline}
    \int_gset_eq:NN \g_@@_last_saved_line_int \l_tmpa_int
  }

}

\DeclareDocumentCommand \pseudosavelabel { } {
  \@@_save_label:
}

```

Saving and restoring. In general, we could just use local variables and trust the scope mechanism, but if we use global assignments inside the scope (e.g., because of where in a tabular we must assign things and use them), the original meaning *won't* be restored. Of course, this should *not* be used if assignments are local, as it will globally set the original name to the meaning it had when we entered the scope.

In saving a macro, we also supply a name for the original, which may then be used to refer to it until it's restored.

```

\cs_new:Nn \@@_gsave_as:NN {
  \cs_gset_eq:NN #2 #1
  \group_insert_after:N \cs_gset_eq:NN
  \group_insert_after:N #1
  \group_insert_after:N #2
}

```

6.3 Styles

The first text styling commands are only straight-up shortcuts for normal font commands:

```

\@@_new_cmd:Nnn \nf { } { \normalfont }
\@@_new_cmd:Nnn \tn { m } { \textnormal { #1 } }
\@@_new_cmd:Nnn \kw { m } { \textnormal {\kwfont { #1 } } }
\@@_new_cmd:Nnn \cn { m } { \textnormal {\cnfont { #1 } } }
\@@_new_cmd:Nnn \id { m } { \textnormal {\idfont { #1 } } }

```

(As a side-effect, we've now also defined `\pseudonf` and `\pseudotn`, which we don't really need.) While we're at it, we'll define the initial value for `\kwfont`, which is generally non-extended bold, if that's available, but extended bold otherwise:

```

\cs_new:Nn \@@_b_or_bx: {

  % Note: We're relying on the warning text in \@defaultsubs

```

```

% being defined by \selectfont if the desired font isn't
% found. This won't happen, however, if the same
% \curr@fontshape combination has been attempted before
% (cf. source2e.pdf page 179).

\group_begin:

\cs_if_exist:NT \@defaultsubs {
  \@_gsave_as:NN \@defaultsubs \@_defaultsubs
  \cs_gset_eq:NN \@defaultsubs \relax
}

% This is what we'd like:
\cs_gset:Nn \@_b_or_bx: { \fontseries{b}\selectfont }

% Try it:
\@_b_or_bx:

% Fallback, if that failed:
\cs_if_exist:NT \@defaultsubs {
  \cs_gset_eq:NN \@_b_or_bx: \bfseries
}

\group_end:

% Make sure the new version is used:
\@_b_or_bx:

}

```

Note that the command redefines itself after the first use, so as not to execute the check every time.

The `\pr` command is also a font shortcut, but in addition takes optional parenthesis-delimited arguments, which are set in math mode. To avoid erroneously slurping up following parentheticals, there should be no space separating the command and its optional argument. With current versions of `xparse`, this can be achieved with the `!` argument type, but for compatibility with older `TeX` distributions, I'll deal with it “manually.”

```

\cs_new:Nn \@_fmt_pr:n {
  \textnormal{\prfont{ #1 }}
}

\NewDocumentCommand \@_parse_paren_args { +d() } {
  \IfNoValueF { #1 } {
    \ensuremath{ ( #1 ) }
  }
}

\NewDocumentCommand \@_parse_bracket_or_paren_args { +o } {
  \IfNoValueTF { #1 } {
    \@_parse_paren_args
  } {

```

```

        \ensuremath{ [ #1 ] }
    }
}

\@@_new_cmd:Nnn \pr { m } {
    \@@_fmt_pr:n { #1 }
    \peek_catcode:NTF { ~ } { } {
        \@@_parse_paren_args
    }
}

```

The `\fn` command is similar, but alternatively permits arguments in square brackets.

```

\cs_new:Nn \@@_fmt_fn:n {
    \textnormal{\fnfont{ #1 }}
}

\@@_new_cmd:Nnn \fn { m } {
    \@@_fmt_fn:n { #1 }
    \peek_catcode:NTF { ~ } { } {
        \@@_parse_bracket_or_paren_args
    }
}

```

The `\hd` command is similar to `\pr` command, except that it spans two columns (effectively ignoring the labeling column). Because it needs to be expandable in order to insert the multicolumn, the final, parenthesis-enclosed argument can not be optional (unlike for `\pr`).

```

% \@@_new_ecmd:Nnn \hd { m +r() } {
% Now uses \def syntax:
\@@_new_ecmd:Nnn \hd { #1 ( #2 ) } {
    \multicolumn{2}
    {\pseudohdpreamble}
    {\@@_fmt_pr:n{#1}\ensuremath{(#2)}}
}

```

Finally, `\st` and `\ct` add quotes and comment delimiters, respectively, to the typeset string, keeping it all in `\textnormal`:

```

\@@_new_cmd:Nnn \st { +m } {
    \textnormal {
        \l_@@_st_left_tl {\stfont{#1}} \l_@@_st_right_tl }
    }
\@@_new_cmd:Nnn \ct { +m } {
    \textnormal {
        \l_@@_ct_left_tl {\ctfont{#1}} \l_@@_ct_right_tl }
    }
}

```

Beyond text styling, we also have styling for entire rows, i.e., highlighting:


```

% \NewExpandableDocumentCommand \pseudohl { } {
% For backward compatibility:
\def \pseudohl {
    \rowcolor{\pseudohlcolor}
}

```

Declarations. To declare shortcuts using the various styles, commands à la `DeclareMathOperator` and `DeclareDocumentCommand` are provided:

```

\@@_new_dec:nn { Comment      } { ct }
\@@_new_dec:nn { Constant    } { cn }
\@@_new_dec:nn { Function     } { fn }
\@@_new_dec:nn { Identifier   } { id }
\@@_new_dec:nn { Keyword      } { kw }
\@@_new_dec:nn { Normal       } { tn }
\@@_new_dec:nn { Procedure    } { pr }
\@@_new_dec:nn { String       } { st }

```

6.4 Notation

Here we'll define a couple of symbols that are useful for pseudocode but that are not necessarily entirely standard mathematical notation. First, the double equals sign, ubiquitous in modern programming languages, and useful if `=` is used for assignment. The horizontal scaling of the equals signs, as well as the space between them and the padding on both sides may be adjusted by using the keys `eqs-scale`, `eqs-sep` and `eqs-pad`. Initially, these are set to emulate the `\eqeq` symbol from `stix` when used with Computer Modern, Latin Modern or the like (though the command works just fine with other fonts as well).

```

\NewDocumentCommand \eqs { } {
  \group_begin:
    \muskip_set:Nn \l_tmpa_muskip \l_@@_eqs_pad_tl
    \muskip_set:Nn \l_tmpb_muskip \l_@@_eqs_sep_tl
    \hbox_set:Nn \l_tmpa_box {\!(=\!)}
    \box_scale:Nnn \l_tmpa_box {\l_@@_eqs_scale_fp}{1}
    \mathrel{
      \tex_mskip:D \l_tmpa_muskip
      \box_use:N \l_tmpa_box
      \tex_mskip:D \l_tmpb_muskip
      % \box_use_drop:N \l_tmpa_box
      % Replaced for compatibility
      \box \l_tmpa_box
      \tex_mskip:D \l_tmpa_muskip
    }
  \group_end:
}

```

For convenience and source-code clarity, the following shortcut (i.e., `\==`) is defined (hijacking the `\=` accent command):

```

\cs_gset_eq:NN \c_@@_orig_eq_cs \=

\DeclareDocumentCommand \= { m } {
  \tl_if_eq:nnTF { #1 } { = } {
    \eqs
  } {
    \c_@@_orig_eq_cs{#1}
  }
}

\cs_gset_eq:NN \@@_eq: \= % Stored for \pseudoeq

```

Similarly, there's the Pascal two-dot interval notation, whose implementation mirrors Knuth's `\dts` command from Concrete Mathematics (see `gkpmac.tex`).

```

\cs_new:Nn \@@_dts: {
  \mathinner {
    \ldotp
    \ldotp
  }
}

\NewDocumentCommand \dts { } { \@@_dts: }

```

There's a shortcut (`\.`) defined for this as well (this time hijacking `\.`):

```

\cs_gset_eq:NN \c_@@_dot_cs \.

\DeclareDocumentCommand \. { m } {
  \tl_if_eq:nnTF { #1 } { . } {
    \dts
  } {
    \c_@@_dot_cs{#1}
  }
}

```

6.5 Options

Much of the behavior of `pseudo` may be configured through various options, and these are defined below. You provide these either through `\pseudoset` or (where applicable) as optional arguments to `\` or the `pseudo` environment itself.

The `\usepackage` options (handled by `l3keys2e`) are subject to full expansion, and so many options simply won't work. In order to make the `kw` option as easily available as possible, however, we permit it here, by way of a `bool` that triggers the *actual* key later on:

```

\keys_define:nn { pseudo/package } {
  kw .bool_gset:N = \g_@@_kw_bool,
  kw .default:n = true
}
\ProcessKeysOptions{ pseudo/package }

```

We now define the actual keys used by `\pseudoset`. Note that `hpad` and `hsep` do *not* use `.dim_set:N`. This is because the `dim` would then be interpreted at the point where it's *set*, and not where it's *used*. If we use units like `em` and `ex`, which depend on the font and font size, the spacing would not be updated if we change these things between setting `hpad` and `hsep` and actually typesetting the pseudocode.

```
\keys_define:nn { pseudo } {

    font          .tl_set:N      = \pseudofont,
    font          .initial:n     = \normalfont,

    hpad          .tl_set:N      = \l_@@_hpad_tl,
    hpad          .initial:n     = 0.0em,
    hpad          .default:n     = 0.3em,

    hsep          .tl_set:N      = \l_@@_hsep_tl,
    hsep          .initial:n     = .75em,

    left-margin   .tl_set:N      = \l_@@_left_margin_tl,
    left-margin   .initial:n     = 0pt,

    label         .tl_set:N      = \l_@@_label_tl,
    label         .initial:n     = \arabic*,

    label-align   .code:n        =
        \@@_def_col:nn{ \pseudolabelalign }{#1},
    label-align   .initial:n     = r,

    ref           .tl_set:N      = \thepseudoline,
    ref           .default:n     = \l_@@_label_tl,

    indent-length .tl_set:N      = \l_@@_indent_length_tl,
    indent-length .initial:V     = \q_no_value,

    indent-text   .tl_set:N      = \l_@@_indent_text_tl,
    indent-text   .initial:n     = { \pseudofont\kw{else}\ },

    indent-mark   .tl_set:N      = \l_@@_indent_mark_tl,
    indent-mark   .initial:x     = \c_novalue_tl,
```

The default `indent-mark` is a vertical with width set by `indent-mark-width`. The height and depth are normally `.7\baselineskip` and `.3\baselineskip`, though these are scaled by `line-height`. This is simply handled by measuring the array strut (as given by `arstrutbox`),* which will only work for tabular environments (as set by `begin-tabular` and `end-tabular`) that use `@arstrutbox`. If there is trouble with this, a simple fix would be to set `indent-mark` to `\vrule height .7\baselineskip depth .3\baselineskip width .6pt`,† changing or scaling the values as appropriate.

* Ideally, we should be using `\rule:nnn`, here, but that is a fairly recent command, and not necessarily available.

† Or `\rule[-.3\baselineskip]{.6pt}{\baselineskip}`.

```

indent-mark      .default:n      =
    \tex_vrule:D height \box_ht:N \@arstrutbox
                  depth  \box_dp:N \@arstrutbox
                  width  \l_@@_indent_mark_wd_tl,

indent-mark-width .tl_set:N      = \l_@@_indent_mark_wd_tl,
indent-mark-width .initial:n     = .6pt, % TikZ semithick

indent-mark-color .tl_set:N      = \l_@@_indent_color_tl,
indent-mark-color .initial:n     = lightgray,

indent-level     .int_set:N      =
    \l_@@_initial_indent_level_int,

kwfont           .tl_set:N      = \kwfont,
kwfont           .initial:n     = \@_b_or_bx:,

kw               .meta:n        = { font = \kwfont },
kw               .value_forbidden:n = true,

hl               .meta:n        = { bol-prepend = \pseudohl },
hl               .value_forbidden:n = true,

bol              .tl_set:N      = \l_@@_bol_tl,
bol-append       .code:n        = {
    \tl_put_right:Nn \l_@@_bol_tl {#1}
},
bol-prepend      .code:n        = {
    \tl_put_left:Nn \l_@@_bol_tl {#1}
},

eol              .tl_set:N      = \l_@@_eol_tl,
eol-append       .code:n        = {
    \tl_put_right:Nn \l_@@_eol_tl {#1}
},
eol-prepend      .code:n        = {
    \tl_put_left:Nn \l_@@_eol_tl {#1}
},

% Defined differently in beamer -- see below
pause            .meta:n        = ,
pause            .value_forbidden:n = true,

cnfont           .tl_set:N      = \cnfont,
cnfont           .initial:n     = \textsc,

idfont           .tl_set:N      = \idfont,
idfont           .initial:n     = \textit,

stfont           .tl_set:N      = \stfont,
stfont           .initial:n     = \textnormal,

st-left          .tl_set:N      = \l_@@_st_left_tl,
st-left          .initial:n     = ‘,

```

```

st-right      .tl_set:N      = \l_@@_st_right_tl,
st-right      .initial:n     = '',

prfont        .tl_set:N      = \prfont,
prfont        .initial:n     = \cnfont,

fnfont        .tl_set:N      = \fnfont,
fnfont        .initial:n     = \idfont,

ctfont        .tl_set:N      = \ctfont,
ctfont        .initial:n     = \textit,

ct-left       .tl_set:N      = \l_@@_ct_left_tl,
ct-left       .initial:n     = (,

ct-right      .tl_set:N      = \l_@@_ct_right_tl,
ct-right      .initial:n     = ),

hl-color      .tl_set:N      = \pseudohlcolor,
hl-color      .initial:n     = black!12,

dim-color     .tl_set:N      = \pseudodimcolor,
dim-color     .initial:n     = \pseudohlcolor,

dim           .meta:n        = {
  bol-append   = \color{\pseudodimcolor},
  setup-append = \color{\pseudodimcolor}
},

line-height   .tl_set:N      = \l_@@_line_height_tl,
line-height   .initial:n     = 1,

start         .tl_set:N      = \l_@@_start_tl,
start         .initial:n     = 1,

```

Line structure. The preamble for the internal `tabular` is defined as a single column type, to make it easier to apply it despite the `array` protections against expansion.

```

preamble      .code:n        =
  \@@_def_col:nn{ \pseudopreamble }{#1},

```

The preamble is laid out as described in section 4:

```

preamble      .initial:n     = {
  >{ \pseudohpad }
  \pseudolabelalign
  >{ \pseudosetup }
  1
  <{ \pseudohpad }
},
setup         .tl_set:N      = \l_@@_setup_tl,

```

```

setup          .initial:n      = {
    \pseudoindent \pseudofont \pseudosavelabel
},

setup-append   .code:n         = {
    \tl_put_right:Nn \l_@@_setup_tl {#1}
},
setup-prepend  .code:n         = {
    \tl_put_left:Nn \l_@@_setup_tl {#1}
},

```

The preamble used for multicolumns is treated similarly:

```

hd-preamble    .code:n         =
    \@@_def_col:nn{ \pseudohdpreamble }{#1},
hd-preamble    .initial:n      = {
    >{\pseudohpad} 1 <{\pseudohpad}
},

```

The prefix is inserted by the row separator command.

```

prefix         .tl_set:N       = \pseudoprefix,
prefix         .initial:n      = {
    \pseudobol \stepcounter* \pseudolabel &
},

```

Tabular setup. The beginning and end of the tabular environment, as well as some positioning and spacing.

```

pos            .tl_set:N       = \pseudopos,
pos            .initial:n      = t,

prevdepth      .tl_set:N       = \l_@@_prevdepth_tl,
prevdepth      .initial:n      = .3\baselineskip,

begin-tabular  .tl_set:N       = \l_@@_begin_tabular_tl,
begin-tabular  .initial:n      =
    \begin{tabular}[\pseudopos]{\pseudopreamble},

end-tabular    .tl_set:N       = \l_@@_end_tabular_tl,
end-tabular    .initial:n      = \end{tabular},

```

List-like spacing. Space above and below is handled similarly to in the built-in L^AT_EX lists, with the option of locally overriding `\topsep` and `\partopsep`, with `compact` used to control the presence of this spacing (overriding the ordinary automatic choice based on the current mode).

```

topsep         .tl_set:N       = \l_@@_topsep_tl,
topsep         .initial:n      = { \topsep },

partopsep      .tl_set:N       = \l_@@_partopsep_tl,
partopsep      .initial:n      = { \partopsep },

```

```

compact      .meta:n      = {
  compact-val = { #1 },
  compact-def = true,
  compact-code = { #1 },
},
compact      .default:n    = true,

% For internal use:
compact-val  .bool_set:N   = \l_@@_compact_bool,
compact-def  .bool_set:N   = \l_@@_compact_def_bool,
compact-code .code:n       = {
  \bool_if:nT { \l_@@_compact_bool } {
    \tl_clear:N \pseudopos
  }
},

```

Details. Finally, some tweakable parameters.

```

eqs-scale    .fp_set:N     = \l_@@_eqs_scale_fp,
eqs-scale    .initial:n    = 0.6785,

eqs-sep      .tl_set:N     = \l_@@_eqs_sep_tl,
eqs-sep      .initial:n    = 0.63mu,

eqs-pad      .tl_set:N     = \l_@@_eqs_pad_tl,
eqs-pad      .initial:n    = 0.28mu,

}

```

Now that we’ve defined the real `kw` key, we reexamine the placeholder handled by `l3keys2e`:

```

\bool_if:NT \g_@@_kw_bool {
  \keys_set:nn { pseudo } { kw }
}

```

Beamer overlays. We redefine the `pause` key if we’re using `beamer`:

```

\bool_if:NT \c_@@_beamer_bool {
  \keys_define:nn { pseudo } {
    pause .meta:n = { eol-append = \pause }
  }
}

```

There’s also the mechanism for handling overlay specifications on keys. Here we handle unknown keys by checking if they end with an overlay specification, and if they do, and we’re in `beamer`, we extract it. Outside `beamer`, keys with overlays are simply ignored.

Note that because unknown keys currently can’t have a default (which we could, in this case, use for some kind of marker, indicating no value was supplied), the only solution is to treat an empty value the same way as no value, in

this case. This means that `foo<1>` and `foo<1>={}` are equivalent, and both will trigger the default of `foo`, even though the latter of the two really shouldn't.*

```
\cs_new:Nn \@@_keys_set_overlay:nnn {
  \bool_if:NT \c_@@_beamer_bool {
    \only<#1>{ \keys_set:nn { #2 } { #3 } }
  }
}
\cs_generate_variant:Nn \@@_keys_set_overlay:nnn { VnV }
\msg_new:nnn { pseudo } { unknown-key } {
  Unknown-key~'#1'~ignored.
}

\tl_new:N \l_@@_overlay_tl

\keys_define:nn { pseudo } {
  unknown .code:n = {

    \group_begin:

    \tl_map_inline:Nn \l_keys_key_tl {

      \tl_if_eq:nnTF { ##1 } { < } {

        \int_incr:N \l_tmpa_int
        \int_compare:nF { \l_tmpb_int == 0 } {
          % We already found '>'!
          % Increment again to prevent match:
          \int_incr:N \l_tmpa_int
        }

        \tl_set_eq:NN \l_tmpb_tl \l_tmpa_tl
        \tl_clear:N \l_tmpa_tl

      } {

        \tl_if_eq:nnTF { ##1 } { > } {

          \int_incr:N \l_tmpb_int

          \tl_set_eq:NN \l_@@_overlay_tl \l_tmpa_tl
          \tl_clear:N \l_tmpa_tl

        } {

          \tl_put_right:Nn \l_tmpa_tl { ##1 }

        } }

      }

    % A single '<' and a final, single '>'?
```

* See <https://github.com/latex3/latex3/issues/67>.


```

\bool_if:nTF {
  \int_compare_p:n { \l_tmpa_int == \l_tmpb_int == 1 }
  &&
  \tl_if_empty_p:N \l_tmpa_tl
}{
  % Match
  \tl_if_blank:nF{#1} {
    \tl_put_right:Nn \l_tmpb_tl {= #1}
  }
  \@@_keys_set_overlay:VnV
  \l_@@_overlay_tl { pseudo } \l_tmpb_tl
} {
  % No match
  \msg_error:nxx
  { pseudo } { unknown-key } { \l_keys_path_tl }
}

\group_end:
}
}

```

Option processing. To let the user work with the options (other than when they’re available as optional arguments to other commands), we supply a command for setting them.

```

\cs_new:Nn \@@_set:n { \keys_set:nn { pseudo } { #1 } }

```

6.6 The row separator

Much of the work of the `pseudo` environment is performed by the row separator, that is, the `\\` command; whatever part of the line structure (see section 4) that’s not in the `preamble` must be handled by `\\`. For example, this is where the `prefix` gets inserted. One reason for this is that there is no straightforward way to insert the column separator (`&`) from the `preamble` itself; and if you want to prevent the column separator insertion because you need to do some custom work in the first column, you’ll probably want to suppress other parts of the `prefix` as well, so they might as well be collected in one place.

Beyond inserting material such as `\tabularnewlines` and `prefix` contents, `\\` is also an entrypoint for local customization, i.e., modifying the indentation level and setting any locally meaningful keys.

Indentation utilities. First we have some functions for modifying the indentation level—essentially just incrementing, decrementing and setting it to zero.

```

\cs_new:Nn \@@_inc_indent: {
  \int_gincr:N \g_@@_indent_level_int
}

\cs_new:Nn \@@_dec_indent: {

```

If the user happens to dedent too much, we might as well be a bit forgiving, and clamp the indent level to non-negative values:

```
% Not using \c_zero_int for compatibility
\int_compare:nNnT \g_@@_indent_level_int > 0 {
  \int_gdecr:N \g_@@_indent_level_int
}
}
```

The actual row separator. The command consists of a few interacting macros. The implementation of `\` is `@@_eol:`, but that is just a thin wrapper that counts pluses and minuses, before handing the control over to `@@_eol_tail`. This is where the remaining argument parsing takes place, and the `\tabularnewline` is inserted, after which control is passed to `@@_bol:` in order to begin a new line—unless we’re at the end of the environment.

```
\cs_new:Nn \@@_eol_handle_args:nnn {
  \@@_keys_set_overlay:nnn { #2 } { pseudo } { h1 }
  \keys_set:nn { pseudo } { #3 }
```

The variables underlying the keys (`\l_@@_label_tl`, etc.) are kept local, so they’ll be restored after the environment, but in order to carry over to the next line and its preamble, we need to perform some global assignments here.

```
\tl_gset_eq:NN \pseudolabel \l_@@_label_tl
\tl_gset_eq:NN \pseudobol \l_@@_bol_tl
\tl_gset_eq:NN \pseudoeol \l_@@_eol_tl
\tl_gset_eq:NN \pseudosetup \l_@@_setup_tl
```

If starred, clear out the prefix:

```
\IfBooleanTF { #1 } {
  \tl_gclear:N \g_@@_cur_prefix_tl
} {
  \tl_gset_eq:NN \g_@@_cur_prefix_tl \pseudoprefix
}
}
\NewDocumentCommand \@@_eol_tail { s d<> +0{ } } {
  \@@_eol_handle_args:nnn{#1}{#2}{#3}
```

A new line is begun only if we’re not at the end of the (or, at least of *some*) environment. (We could have put the `\tabularnewline` outside, but then we’d have a conditional at the beginning of the next line, which would mess up `\bottomrule` or the like. We need to keep `@@_bol:` alone at the start of the line.)

```
\peek_meaning_ignore_spaces:NF \end {
  \pseudoeol
  \tabularnewline
  \@@_bol:
}
```

```
}
```

And here is the actual `\@@_eol:` command:

```
\cs_new:Nn \@@_eol: {
    \@@_per_char:nnn { + } {
        \@@_inc_indent:
    } {
    \@@_per_char:nnn { - } {
        \@@_dec_indent:
    } {
        \@@_eol_tail
    } }
}
```

The `\@@_bol:` command (currently) just inserts the `prefix:`

```
\cs_new:Nn \@@_bol: {
    \g_@@_cur_prefix_tl
}
```

6.7 Various user commands

A few user-level wrappers around internal commands. First, a couple primarily for use in the `preamble`, together with `\pseudosavelabel` and `\pseudofont:`

```
\NewDocumentCommand \pseudohpad { } {
    \skip_horizontal:n { \l_@@_hpad_tl - \tabcolsep }
}
\NewDocumentCommand \pseudoindent { } {
    \@@_indent:N { \g_@@_indent_level_int }
}
```

The `\pseudoslash` command simply redefines the row separator, and is used at the start of the `pseudo` environment. It may be useful for the user if some other construct redefines `\` as well. (This is similar to the `\arraycr` command of the `array` package.)

```
\NewDocumentCommand \pseudoslash { } {
    \cs_gset_eq:NN \ \ \@@_eol:
}
```

We also have a command for restoring our definition of `\=` if it has been overwritten:

```
\NewDocumentCommand \pseudoeq { } {
    \cs_gset_eq:NN \= \@@_eq:
}
```

Finally, two utilities for working with options. The first (`\pseudoset`) directly sets a collection of keys, while the second (`\pseudodefinesstyle`) defines a new key which can be used as a shortcut for setting multiple keys at some later point:

```
\NewDocumentCommand \pseudoset { +m }
  { \@@_set:n { #1 } }

\NewDocumentCommand \pseudodefinesstyle { m +m } {
  \keys_define:nn { pseudo } {
    #1 .meta:n = {
      #2
    }
  }
}
```

6.8 The pseudo environment

While this is the main attraction, it's essentially just an augmented `tabular` environment, which does a bit of setup initially, using the various macros already described.

```
\NewDocumentEnvironment { pseudo } { +o s d<> +0{ } } {

  \group_begin:

  \@@_gsave_as:NN \ \ \c_@@_saved_cr_cs
  \@@_gsave_as:NN \ = \c_@@_saved_eq_cs

  % \pseudoslash is inside the tabular
  \pseudoeq

  \int_set:Nn \g_@@_last_saved_line_int {\arabic{pseudoline}}
  \@@_star_setup:

  \IfNoValueF { #1 } {
    \pseudoset { #1 }
  }
  \@@_set_indent_length:

  % If not manually set as compact/noncompact,
  % set automatically:
  \bool_if:NF \l_@@_compact_def_bool {
    \bool_set:Nn \l_@@_compact_bool {
      \mode_if_horizontal_p: && \mode_if_inner_p:
    }
  }

  \bool_if:NF { \l_@@_compact_bool } {

    \skip_set:Nn \l_tmpa_skip {
      \l_@@_topsep_tl
    }
  }
```

```

\mode_if_vertical:TF {
  \skip_add:Nn \l_tmpa_skip { \l_@@_partopsep_tl }
} {
  \unskip \par
}

\addvspace { \l_tmpa_skip }

\noindent
\skip_horizontal:n{ \dim_eval:n { \l_@@_left_margin_tl } }

}

\dim_set:Nn \tabcolsep { \l_@@_hsep_tl / 2 }
\tl_set_eq:NN \arraystretch \l_@@_line_height_tl

\stepcounter{pseudoenv}
\setcounter{pseudoline}{\l_@@_start_tl}
\addtocounter{pseudoline}{-1}

\tl_use:N \l_@@_begin_tabular_tl

```

We use `\noalign` to be able to place these definitions inside the `tabular`, without messing up `\multicolumn` or `\hline` or the like. It's not really supposed to be used in `expl3`; the alternative would be to create an extra dummy line, like:

```

[
  \skip_vertical:n{ -\dim_eval:n{ \box_ht:N \@arstrutbox +
                                   \box_dp:N \@arstrutbox } }
  \tabularnewline
]

```

This would give us a fresh start, without moving vertically. It's probably more hacky than just using `\noalign` here, though, so...

```

\tex_noalign:D {

```

We keep the `\\`-definition inside the `tabular`, to override the redefinition placed there by `array`, without patching any internals:

```

\pseudoslash

```

In a `tabularx`, for example, the body is executed multiple times, so we must make sure that any resets that are performed—such as setting the initial indentation level—are performed each time:

```

\int_gset_eq:NN \g_@@_indent_level_int
               \l_@@_initial_indent_level_int

```

Finally, we handle the line arguments, just like with the row separator:

```

\@@_eol_handle_args:nnn{#2}{#3}{#4}
}

```

Definitions and setup are done, we've left the `\noalign`, and we can start the line:

```

\@@_bol:

} {

\tl_use:N \l_@@_end_tabular_tl

```

We'll only adjust spacing here if we're not `compact`. Otherwise, we'll just end the current group:

```

\bool_if:nTF { \l_@@_compact_bool } {

\group_end:

} {

\mode_if_vertical:F {
\unskip \par
\group_insert_after:N \endparenv
}

\addvspace{ \l_tmpa_skip }

```

To ensure any local changes to `prevdepth` are used, we expand its local value before setting `csprevdepth` *outside* the group.*

```

\exp_args:NNNV
\group_end:
\dim_set:Nn \prevdepth \l_@@_prevdepth_tl

}

}

```

The starred version of the environment is just a wrapper that uses the custom (and overridable) `starred` style:

```

\pseudodefinesstyle{starred}{
  preamble={
    >\pseudohpad\pseudoindent\pseudofont}
    1
    <\pseudohpad}
  },
  prefix={\pseudobol},
}

\NewDocumentEnvironment { pseudo* } { +0{} } {
  \begin{pseudo}[starred, #1]
  % \begin{pseudo} will "eat" any remaining arguments to pseudo*

```

* See, e.g., <https://tex.stackexchange.com/questions/56294>.

```

} {
  \end{pseudo}
}

```

6.9 Boxes and floats

Some spacing and width values are taken from `booktabs`, to partly emulate its table appearance. If `booktabs` is not loaded, we'll just define these constants ourselves; if `booktabs` is loaded later, it will blithely overwrite these.

```

\@ifpackageloaded { booktabs } { } {
  \dim_const:Nn \aboverulesep { .40ex }
  \dim_const:Nn \belowrulesep { .65ex }
  \dim_const:Nn \heavyrulewidth { .08em }
  \dim_const:Nn \lightrulewidth { .05em }
}

```

We'll be adjusting the spacing after the contents based on the value of `\prevdepth`. If `\prevdepth` is negative, this is suppressed. Otherwise, we add vertical space to the `\prevdepth`, to take us to (at least) `.3\baselineskip`. Since the mechanism is the same for the title and the body, we define a macro:

```

\cs_new:Npn \@@_prevdepth_adjustment: {
  \par % Ensure vertical mode
  \dim_compare:nNnF \prevdepth < \c_zero_dim {
    \dim_compare:nNnT \prevdepth < { .3\baselineskip } {
      \skip_vertical:n { .3\baselineskip - \prevdepth }
      \skip_vertical:N \c_zero_dim % Hide previous skip
    }
  }
}

```

We now define some `tcolorbox` box styles. Rather than importing `tcolorbox`, we just use its key management mechanism, `pgfkeys`, with the appropriate namespace

```

\pgfkeys {tcb/pseudo} {

```

The first style (`pseudo/boxruled`) is the basis for the others.

```

boxruled/.style = {

```

By default, our boxes aren't floats, but if the `float` style is used, we'll want to have the placement configured. The `tcolorbox` default is `htb`, but we're emulating normal floats, so we'll use the normal L^AT_EX default:

```

floatplacement = tbp,

```

Before the contents (which uses the upper part of the box), we adjust some distances, and set `\prevdepth`, for consistent vertical spacing of the first line.

These settings may be overridden using `before upper app`, which appends code to `before upper`. For example, one could change the `\topsep` by using `before upper app = {\topsep10pt}.*`

```
before-upper = {
  \dim_set:Nn \parskip    { .3\baselineskip }
  \dim_set:Nn \topsep     { .2\baselineskip }
  \dim_set:Nn \partopsep  { 0pt             }
  \dim_set:Nn \prevdepth  { .3\baselineskip }
},
```

At the end of the contents, we add some spacing, again for consistent vertical alignment.

```
after-upper = \@_prevdepth_adjustment:,
```

Spacing before and after the box, when not used with `float`:[†]

```
beforeafter-skip-balanced = 0.5\baselineskip plus 2pt,
```

Now we set up basic spacing for the contents. The spacing above and below the title is the same as for the top row of a `booktabs` tabular. For the “body” of the box, we add some extra space.

```
boxsep          = 0pt,
toptitle        = \belowrulesep,
bottomtitle     = \aboverulesep,
top             = 1.75 \belowrulesep,
bottom          = 1.75 \aboverulesep,
left            = 1.50 \belowrulesep,
right           = 1.50 \belowrulesep,
```

The title has a separate part called the *description* (in `tcolorbox` theorem terms). We give the title one font (bold), and then reset that to `\normalfont` when we get to the description.

```
fonttitle       = \bfseries,
description-font = \normalfont,
```

The spacing above and below the title is adjusted as for the body. We want to separate the initial part of the title (e.g., “Algorithm 3.2”) from the description by an `\enskip` (.5em horizontal space). However, a single normal space is hard-coded into `tcolorbox`, so we’ll subtract the width of that. (We make sure to do this with `\normalfont`, to not get the units warped by an extended bold, for example.)

* While you might want to modify `\parskip`, `\topsep` or `\partopsep`, there’s probably no need to change `\prevdepth`.

† This setting is supposed to be the `tcolorbox` default; however, though in some versions, this default does not work (cf. <https://github.com/T-F-S/tcolorbox/issues/182>).


```

before-title      =
  \dim_set:Nn \prevdepth { .3\baselineskip },
after-title       = \@@_prevdepth_adjustment:,
separator~sign    = {
  \normalfont
  \skip_horizontal:n{ .5em - \fontdimen2\font\space }
},

```

Finally, some basic styling. The values `.4pt` and `.2pt` correspond to the `tikz` line thicknesses `thin` and `very thin`.

```

sharp~corners,
colback          = white,
colbacktitle     = white,
coltitle         = black,
colframe         = black,
boxrule          = .4pt,
titlerule        = .2pt,
},

```

The remaining box styles are based on `boxruled`, but rely on other skins (`\empty` and `\tile`), which remove the default frame drawing, as some of the frame are removed. (It would be possible to simply set the relevant widths to zero, but this tends to leave perceptible hairlines in most PDF viewers—probably because the frame is drawn by *filling* rather than *drawing*.)

```

ruled/.style = {
  pseudo/boxruled,
  empty,

```

Even though we’ve removed the default frame, we do want some rules. First the various rule thicknesses (and some horizontal spacing) are set. The ones that are missing have their thicknesses set to zero, for spacing/positioning purposes. The `titlerule` is drawn normally, but for the top and bottom rules, we need to use the `borderline` mechanism.

```

boxrule          = 0pt,
toprule          = \heavyrulewidth,
titlerule        = \lightrulewidth,
bottomrule       = \heavyrulewidth,
left             = 0pt,
right            = 0pt,
titlerule~style  = draw,
borderline~horizontal = {\heavyrulewidth}{0pt}{black},
},

```

The `boxed` style is similar, with the `titlerule` removed, and with the `borderline` drawn on all sides. Finally, we want to “simulate” the title just being the first paragraph of the contents, so we zero out the spacing after the title, and make the top spacing equal to the the normal `\parskip` (which we set to `.3\baselineskip` in `before upper`).

```

boxed/.style = {
  pseudo/boxruled,
  empty,
  titlerule          = 0pt,
  borderline          = {.4pt}{0pt}{black},
  bottomtitle        = 0pt,
  top                 = .3\baselineskip,
},

```

The `tworuled` style is based on `boxed` (including the spacing adjustment between title and body), but removes the previously drawn borderlines, adjusts the thicknesses, and draws new horizontal lines.

```

tworuled/.style = {
  pseudo/boxed,
  no-borderline,
  left           = 0pt,
  right          = 0pt,
  boxrule        = 0pt,
  toprule        = \heavyrulewidth,
  bottomrule     = \heavyrulewidth,
  borderline-horizontal = {\heavyrulewidth}{0pt}{black},
},

```

Finally, the `filled` style uses the `tile` skin, which has no frame, and is designed for filling. In addition to the colors, there's a slight spacing adjustment.*

```

filled/.style = {
  pseudo/boxruled,
  tile,
  colback       = black!11,
  colbacktitle  = black!22,
  left          = 2 \belowrulesep,
  right         = 2 \belowrulesep,
}

```

And that ends the `tcolorbox` definitions:

```

} % \pgfqkeys

```

References

- [1] T. H. Cormen et al. *Introduction to Algorithms*. 3rd ed. MIT Press, 2009.
- [2] T. H. Cormen et al. *Introduction to Algorithms*. 4th ed. MIT Press, 2022.
- [3] R. L. Graham, D. E. Knuth, and O. Patashnik. *Concrete Mathematics: A Foundation for Computer Science*. Addison-Wesley Professional, 1994.

* The `tile` skin also sets things like `sharp corners` and `fonttitle`, so some of what we inherit from `boxruled` is a bit redundant, here.

- [4] D. E. Knuth. “Random Matroids”. *Discrete Mathematics* 12.4 (1975), pp. 341–358.
- [5] D. P. Williamson and D. B. Shmoys. *The Design of Approximation Algorithms*. Cambridge University Press, 2011.