KoDi

KOMMUTATIVE DIAGRAMME FÜR TEX

ENCHIRIDION

UNRELEASED

v1.0.0

25TH FEBRUARY 2017

KODI is a TIKZ library. Its purpose is drawing commutative diagrams. It is designed precisely to overcome the shortcomings of traditional ones. The syntax is minimalistic and intelligible.

Paolo Brasolin

Preliminaries

TikZ is the only dependency of koDi. This ensures compatibility with most TEX flavours. Furthermore, it can be invoked as a standalone as well as a TikZ library. Below are minimal working examples for the main dialects.

TEX package

```
\input kodi

\kodi
  % diagram here
\endkodi
\bye
```

ConTEXt module

```
\usemodule[kodi]
\starttext
\startkodi
  % diagram here
\stopkodi
\stoptext
```

LATEX package

```
\documentclass{article}
\usepackage{kodi}
\begin{document}
\begin{kodi}
  % diagram here
\end{kodi}
\end{document}
```

TEX (TikZ library)

```
\input tikz
\usetikzlibrary kodi

\tikzpicture[kodi]
  % diagram here
\endtikzpicture
\bye
```

ConTEXt (TikZ library)

```
\usemodule[tikz]
\usetikzlibrary[kodi]
\starttext
\starttikzpicture[kodi]
  % diagram here
\stoptikzpicture
\stoptext
```

LATEX (TikZ library)

```
\documentclass{article}
\usepackage{tikz}
\usetikzlibrary{kodi}
\begin{document}
\begin{tikzpicture}[kodi]
  % diagram here
\end{tikzpicture}
\end{document}
```

A useful TikZ feature exclusive to LATEX is externalization. A small expedient is necessary to use it with koDi.

TikZ externalization

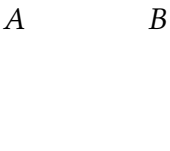
```
\documentclass{article}
\usepackage{tikz}
\usetikzlibrary{kodi}
\usetikzlibrary{external}
\tikzexternalize
  [prefix=tikzpicfolder/]
\begin{document}
\begin{tikzpicture}[kodi]
  % diagram here
\end{tikzpicture}
\end{document}
```

Quick tour

Objects are typeset using the **\obj** macro.

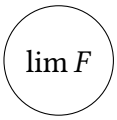

```
\obj {X};
```

Almost every diagram is laid along a regular grid, so the customary tabular syntax of TEX is recognized.



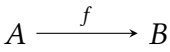

```
\obj {
  A & B \\
  C & D \\
};
```

KODI objects are self-aware and clever enough to name themselves so you can comfortably refer to them.



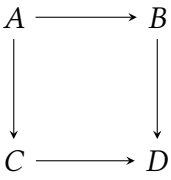

```
\obj {\lim F};
\draw (lim F) circle (4ex);
```

Morphisms are typeset using the **\mor** macro.




```
\obj { A & B \\ };
\mor A f:-> B;
```

Commutative diagrams exist to illustrate composition and commutation, so KODI allows arrow chaining and chain gluing.




```
\obj { A & B \\ C & D \\ };
\mor A -> B -> D;
\mor * -> C -> *;
```

These are the only two macros defined by KODI.

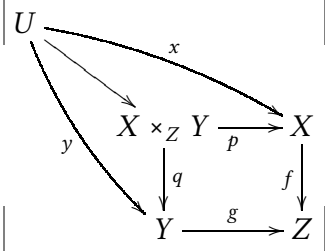
There are more features: read on if this caught your attention.

Alternatives

It is only fair to mutely offer a comparison with mainstream packages, showing idiomatic code to draw the same diagram.

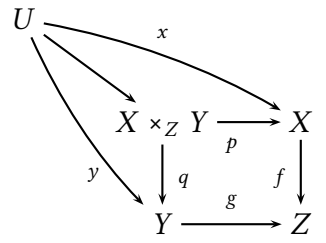
Let Xy-pic set the bar with a *verbatim* extract from its manual.

```
\xymatrix{
 U \ar@/_/[ddr]_y \ar[dr] \ar@/^/[drr]^x \\
  & X \times_Z Y \ar[d]^q \ar[r]_p
                  & X \ar[d]_f            \\
  & Y \ar[r]^g & Z                        }
```



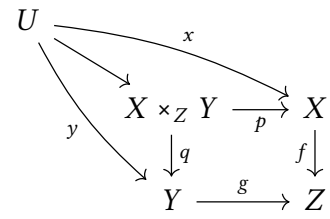
Here is an example adapted from pst-node's documentation.

```
$ \psset{colsep=2.5em, rowsep=2em}
 \begin{psmatrix}
  U \\
 & X\times_Z Y & X \\
 & Y & Z
 \psset{arrows=->, nodesep=3pt}
 \everypsbox{\scriptstyle}
 \ncline{1,1}{2,2}
 \ncarc[arcangle=-10]{1,1}{3,2}_{y}
 \ncarc[arcangle=10]{1,1}{2,3}^{x}
 \ncline{2,2}{3,2}>{q}
 \ncline{2,2}{2,3}_{p}
 \ncline{2,3}{3,3}<{f}
 \ncline{3,2}{3,3}^{g}
 \end{psmatrix}$
```



Next one is refitted from the guide to {tikz-cd}.

```
\begin{tikzcd}[column sep=scriptsize, row sep=scriptsize]
  U
  \arrow[drr, bend left=10, "x"]
  \arrow[ddr, bend right=10, swap, "y"]
  \arrow[dr] & & \\
    & X \times_Z Y \arrow[r, swap, "p"] \arrow[d, "q"]
      & X \arrow[d, swap, "f"] \\
    & Y \arrow[r, "g"]
      & Z
\end{tikzcd}
```

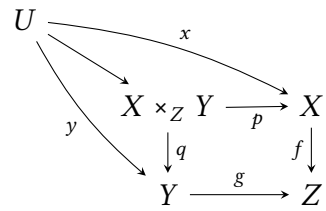


Finally, KODI.

```
\begin{kodi}[golden]
  \obj { |(pb)| X \times_Z Y & X \\
                              Y & Z \\ };
  \obj [above left=of pb] {U};

  \mor[swap] pb p:-> X f:-> Z;
  \mor        * q:-> Y g:-> *;

  \mor                          U   -> pb;
  \mor      :[bend left=10]   * x:-> X;
  \mor[swap]:[bend right=10]  * y:-> Y;
\end{kodi}
```



Syntax: objects

The first of the two macros that KoDi offers is **\obj**. It is polymorphic and can draw both single objects and layouts.

Orange denotes optional fragments.

```
\obj ‹object options› {‹math›};
\obj ‹layout options› {‹layout›};
```

Layouts are described using the customary TEX tabular syntax.

Underlined fragments are repeated one or more times.

```
‹layout›           = ‹row› ‹row separator›
‹row›              = ‹cell› ‹cell separator› ‹cell›
‹row separator›    = \\ [‹length›]
‹cell›             = |‹object options›| ‹math›
‹cell separator›   = & [‹length›]
```

The discretionary options syntax is analogous to standard TikZ nodes and matrices, respectively.

```
‹object options› = [object keylist] (‹name›) at (‹coordinate›)
‹layout options› = [layout keylist] (‹name›) at (‹coordinate›)
```

∴

Very little of the given syntax is specific to KoDi. TikZ options are easy to pick up on the way, blah blah blah.

Here is a kitchen sink for tabular syntax:

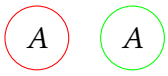
A B C
D E F
G H I

```
\obj { A & B &[1em] C \\
       D & E &      F \\[-1em]
       G & H &      I \\ };
```

Objects are automagically named; the latest homonymous prevails.


```
\obj { A & A \\ };
\draw (A) circle (1em);
```

Naming a specific object avoids its automatic labeling.




```
\obj { A & |(A')| A \\ };
\draw [red]   (A)  circle (1em);
\draw [green] (A') circle (1em);
```

Naming a layout lets you refer to its objects by row and column.


```
\obj (M) { A & A \\ A & A \\ };
\draw [red]   (M-1-2) circle (1em);
\draw [green] (M-2-1) circle (1em);
```

Syntax: morphisms

The second and last macro that KoDi offers is **\mor**. It can draw single or chained morphisms.

```
\mor ‹chain options› ‹object›␣‹morphism›␣‹object›;
```

Whitespace marked as ␣ is mandatory.

Source and target objects are referred to by their name.

```
‹object›    ≡ (‹name›)
```

Blue fragments can be either enclosed in the shown delimiters, or a TEX group (not idiomatic), or simply devoid of whitespace.

Morphisms consist of one or more optional labels and an arrow.

```
‹morphism› ≡ ‹labels› : ‹arrow›
‹labels›   ≡ "‹math›" | ["‹math›", ‹label keylist›]
‹arrow›    ≡ [‹arrow keylist›]
```

Alternatives are separated by |s.

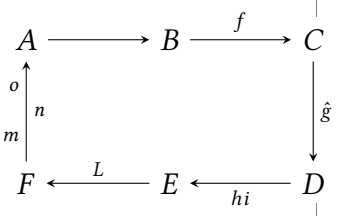
Global options can be given to both labels and arrows.

```
‹chain options› ≡ [‹label keylist›] : [‹arrow keylist›]
```

∴

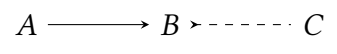
These rules allow for a label syntax that sprouts gracefully from the simplest to the most complex arrow.

```
\mor A -> B;
\mor B f:-> C;
\mor C \hat g:-> D;
\mor D "h i":-> E;
\mor E ["L", above]:-> F;
\mor F ["m", near start]["n", swap]["o", near end]:-> A;
```



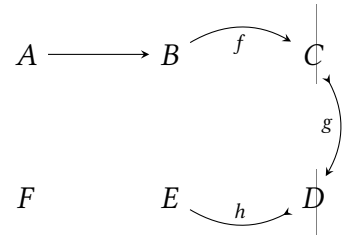
The same holds for arrow syntax.

```
\mor A -> B;
\mor B [>-, dashed] C;
```



Blah.

```
\mor A -> B;
\mor [swap]:[bend left] B f:-> C g:>-> D h:>- E;
```



Names

As you'll have guessed by now, objects name themselves.

The process happens in three steps:

- expand tokens;
- replace characters;
- apply name, overwriting if necessary.

Each one can be configured in any KODI scope with the keys.

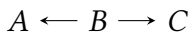
You can control the degree of expansion; the default behaviour (no expansion) shields you from problems with unsafe macros.

Shortcuts

Two special labels exist: $*$ and $+$.

As a source, $*$ evaluates to the head of the previous chain.

```
\mor B -> C;
\mor * -> A;
```

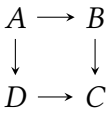


As a target, $*$ evaluates to the tail of the previous chain.

```
\mor B -> C;
\mor D -> *;
```

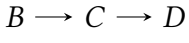
The natural use case for $*$ is chain gluing.

```
\mor A -> B -> C;
\mor * -> D -> *;
```



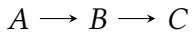
As a source, $+$ evaluates to the tail of the previous chain.

```
\mor B -> C;
\mor + -> D;
```



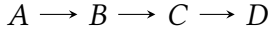
As a target, $+$ evaluates to the head of the previous chain.

```
\mor B -> C;
\mor A -> +;
```



The natural use case for $+$ is chain extension.

```
\mor B -> C;
\mor A -> + -> D;
```



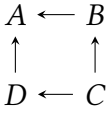
The meanings of $*$ and $+$ swap on opposite chains.

Chain extension can be obtained using $*$.

```
\mor B <- C;
\mor D -> * -> A;
```

Chain gluing can be obtained using $+$.

```
\mor A <- B <- C;
\mor + -> D -> +;
```

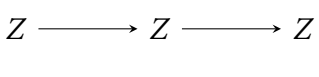


Expansion

The expansion behaviour of the naming routine can be configured inside any KODI scope using the expand key.

```
/kD/expand = none | once | full
```

The three available settings correspond to different degrees of expansion. A side by side comparison completely illustrates their meanings.

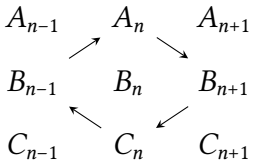


```
\def\B{Z}
\def\A{\B}
\obj{ |[expand=none]| \A &     % name: A (default)
      |[expand=once]| \A &     % name: B
      |[expand=full]| \A \\ }; % name: Z
\mor A -> B -> Z;
```

$\therefore$

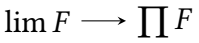
The default behaviour is to avoid expansion in compliance with the principle that *names should be predictable from the* literal *code*. Furthermore, it is seldom wise to liberally expand tokens.

There are circumstances in which it is useful to perform token expansion, though. A useful application is procedural drawing.



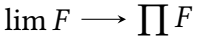
```
\foreach [count=\r] \l in {A,B,C}
  \foreach [count=\c] \n in {n-1,n,n+1}
    \obj [expand=full] at (3em*\c,-2em*\r) {\l_{\n}};
\mor (A_{n}) -> (B_{n+1}) -> (C_{n}) -> (B_{n-1}) -> (A_{n});
```

In some cases finer control is needed. For instance, full expansion yields unpractical results when parametrizing macros.



```
\foreach [count=\c] \m in {\lim,\prod}
  \obj [expand=full] at (4em*\c,0) {\m F};
\mor (protect mathop {relax kern z@ mathgroup
                      symoperators lim}nmlimits@ F)
  -> (DOTSI prodop slimits@ F);
```

This explains why a setting to force a single expansion exists.



```
\foreach [count=\c] \m in {\lim,\prod}
  \obj [expand=once] at (4em*\c,0) {\m F};
\mor (lim F) -> (prod F);
```

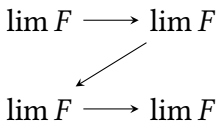
Replacement

The character replacement behaviour of the naming routine can be configured inside any KODI scope using various keys.

```
/kD/replace character = ‹character› with ‹character›
/kD/replace charcode = ‹charcode› with ‹character›
/kD/remove characters = ‹characters›
/kD/remove character = ‹character›
/kD/remove charcode = ‹charcode›
```

You can set up a replacement for any character, using the character code for the hardest to type, like ␣ or \.

```
\obj{ |[replace character=F with G]| \lim F &     % name: lim G
      |[remove character=F]|         \lim F \\    % name: lim
      |[replace charcode=92 with /]| \lim F &     % name: /lim F
      |[remove charcode=32]|         \lim F \\ }; % name: limF
\mor (lim G) -> (lim) -> (/lim F) -> (limF);
```



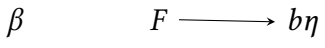
∴

The default behaviour is removal of the minimal set of universally annoying[1] characters: ( ) , . : have special meanings to TIKZ while \ is impossible to type by ordinary means, so they're *kaput*.

Each one can be restored by replacing it with itself. Don't.

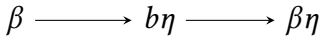
Another egregiously bad idea is replacing characters with spaces. It's tempting because it solves a somewhat common edge case.

```
\obj{ \beta & F & b\eta \\ };
\mor F -> beta;
```



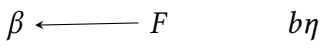
Since characters in names are literal, this causes whitespace duplication and names become inaccessible by ordinary means.

```
\obj [replace charcode=92 with \space]
  { \beta & b\eta & \beta \eta \\ };
\mor beta -> (b eta) -> (beta \space eta);
```



The wise solution is writing better code.

```
\obj{ \beta & F & b \eta \\ };
\mor F -> beta;
```



[1] The difficult part is not creating the names but having to type them.

Overwriting

The name overwriting behaviour of the naming routine can be configured inside any KoDi scope using the `overwrite` key.

```
/kD/overwrite = false | alias | true
```

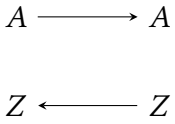
The three available settings correspond to different naming priorities. A side by side comparison completely illustrates their meanings.

```
\obj{ |[overwrite=false] (A')| A &     % names: A'    (default)
      |[overwrite=alias] (B')| B &     % names: B', B
      |[overwrite=true]  (C')| C \\ }; % names:     C
\mor A' -> B';
\mor B  -> C;
```

$\therefore$

The default behaviour is the ideal for manually solving automatic names conflicts.

TODO: Why is false useful? conflict solving.



```
\obj {        A & |(A')| A \\
       |(Z')| Z &        Z \\ };
\mor A -> A';
\mor Z -> Z';
```

TODO: Why is alias useful? semantic aliasing

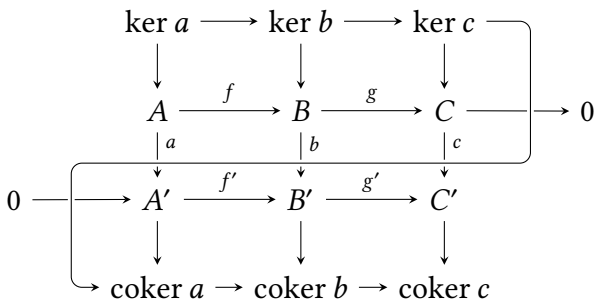
```
\obj [overwrite=alias] { A & |(center)| B & |(right)| C \\ };
\mor A -> B;
\mor center -> right;
```

TODO: Why is true useful? ...completeness?

Gallery

The remainder of the text is just commented examples.

Snake



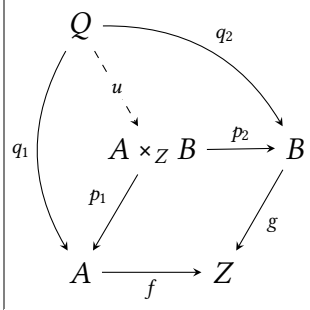

```
\begin{kodi}[golden]
  \obj{          & \ker a   & \ker b   & \ker c   &   \\
                 & A        & B        & C        & 0 \\
        |(0')| 0 & A'       & B'       & C'       &   \\
                 & \coker a & \coker b & \coker c &   \\ };

  \mor   (ker a) ->   (ker b) ->   (ker c);
  \mor (coker a) -> (coker b) -> (coker c);
  \mor       A  f :-> B  g :-> C -> 0;
  \mor 0' -> A' f':-> B' g':-> C';

  \mor[near start] (ker a) -> A a:-> A' -> (coker a);
  \mor[near start] (ker b) -> B b:-> B' -> (coker b);
  \mor[near start] (ker c) -> C c:-> C' -> (coker c);

  \draw[/kD/arrows/crossing over, ->, rounded corners, >=stealth]
    (ker c) -- ++( 0.6,0) -- ++(0,-1.6)
            -- ++(-3.2,0) -- ++(0,-1.4) -- (coker a);
\end{kodi}
```

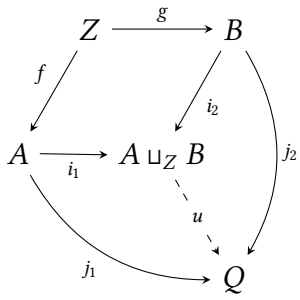
Pullback & pushout




```
\begin{kodi}[comb]
  \obj{ |(pb)| A \times_Z B & B \\
              A             & Z \\ };
  \obj[above left=of pb] {Q};

  \mor[swap] pb p_1:-> A f:-> Z;
  \mor        * p_2:-> B g:-> *;

  \mor[swap]:[bend right] Q q_1:-> A;
  \mor      :[bend left]  * q_2:-> B;
  \mor [mid]:[dashed]     *   u:-> pb;
\end{kodi}
```

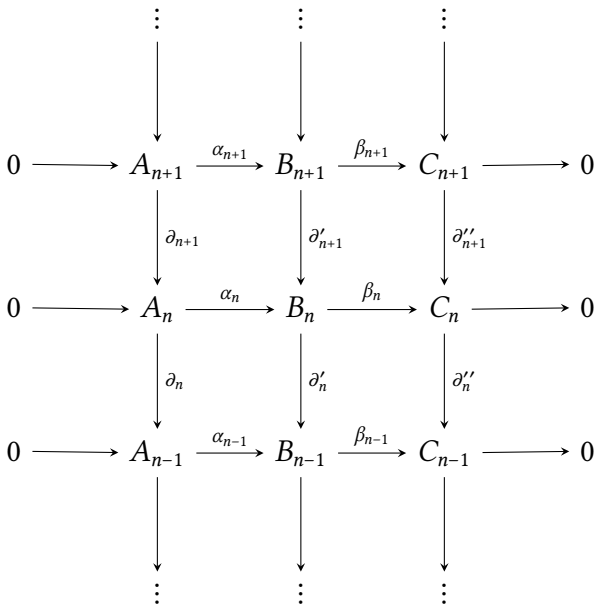



```
\begin{kodi}[comb]
  \obj{ Z &                   B \\
        A & |(po)| A \sqcup_Z B \\ };
  \obj[below right=of po] {Q};

  \mor[swap] Z f:-> A i_1:-> po;
  \mor       * g:-> B i_2:-> *;

  \mor[swap]:[bend right]  A j_1:-> Q;
  \mor      :[bend left]   B j_2:-> *;
  \mor [mid]:[dashed]     po   u:-> *;
\end{kodi}
```

Complexes sequence

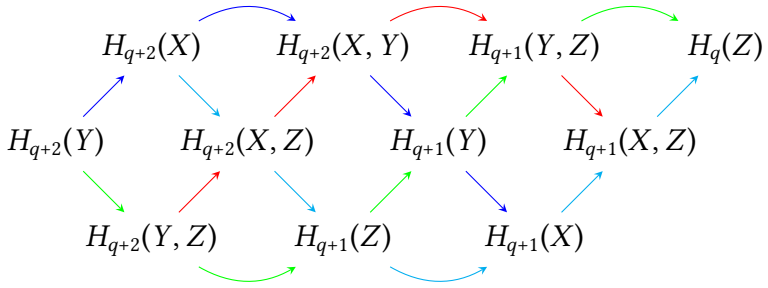



```
\begin{kodi}
  \obj (M) {   & \vdots  & \vdots  & \vdots  &   \\
             0 & A_{n+1} & B_{n+1} & C_{n+1} & 0 \\
             0 & A_{n}   & B_{n}   & C_{n}   & 0 \\
             0 & A_{n-1} & B_{n-1} & C_{n-1} & 0 \\
               & \vdots  & \vdots  & \vdots  &   \\ };

  \foreach \n/\row in {n+1/2, n/3, n-1/4}
    \mor (M-\row-1) -> (A_{\n}) "\alpha_{\n}":-> (B_{\n})
                                "\beta_{\n}":-> (C_{\n}) -> (M-\row-5);

  \foreach \l/\col/\q in {A/2/, B/3/', C/4/''}
    \mor (M-1-\col) -> (\l_{n+1}) "\partial \q_{n+1}":-> (\l_{n})
                                  "\partial \q_{n}"  :-> (\l_{n-1}) -> (M-5-\col);
\end{kodi}
```

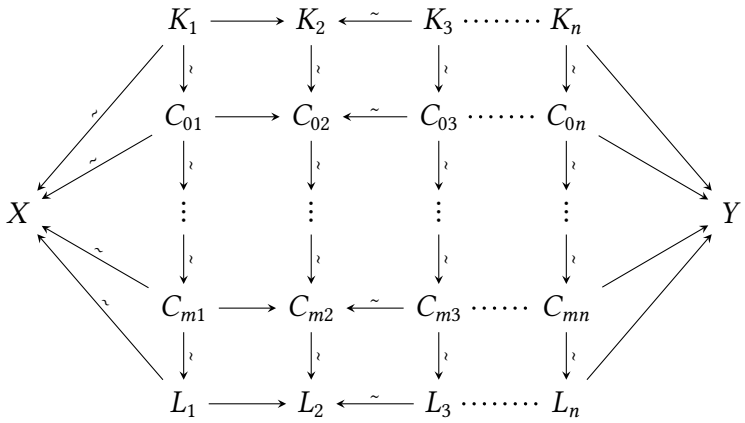
Braid




```
\begin{kodi}[ l/.style={bend left}, r/.style={bend right} ]
  \obj [ comb=step 6em angle 45, remove characters=H_\{q+\} ] {
    H_{q+2}(X)   & H_{q+2}(X,Y) & H_{q+1}(Y,Z) & H_{q}(Z)     \\
    H_{q+2}(Y)   & H_{q+2}(X,Z) & H_{q+1}(Y)   & H_{q+1}(X,Z) \\
    H_{q+2}(Y,Z) & H_{q+1}(Z)   & H_{q+1}(X)   &              \\
  };

  \mor :[blue]  2Y  -> 2X  l,-> 2XY   -> 1Y  -> 1X;
  \mor :[green] 2Y  -> 2YZ r,-> 1Z    -> 1Y  -> 1YZ l,-> Z;
  \mor :[cyan]  2X  -> 2XZ   -> 1Z  r,-> 1X  -> 1XZ   -> Z;
  \mor :[red]   2YZ -> 2XZ   -> 2XY l,-> 1YZ -> 1XZ;
\end{kodi}
```

Hammock




```
\begin{kodi}[x=4em, y=-3em, node distance=1 and 1,
    sim/.style={sloped, auto,
      edge node={node[every edge quotes][/velos/install quote
        handler,"\sim", anchor=south, outer sep=-.15em]}
    },
    ␣>/.style={->, sim},
    <␣/.style={<-, sim},
    ../.style={line width=.25ex, dash pattern=on 0sp off .75ex, line cap=round},
    remove characters=_\{\},
    expand=full,
  ]

  \foreach [count=\c] \col in {1, 2, 3, n}
  \foreach [count=\r] \row in {K_{\col}, C_{0\col}, \vdots, C_{m\col}, L_{\col}}
    \obj [name/.expanded={\ifnum\r=3 vdots\col\fi}] at (\c,\r) {\row};

  \obj  [left=of vdots1] {X};
  \obj [right=of vdotsn] {Y};

  \foreach \col in {1, 2, 3, n}
    \mor (K\col) ␣> (C0\col) ␣> (vdots\col) ␣>  (Cm\col) ␣> (L\col);

  \foreach \row in {K, C0, Cm, L} {
    \mor (\row1) -> (\row2) <␣ (\row3) .. (\row n);
    \mor X <␣ + -> Y;
  }
\end{kodi}
```