CoDi

Commutative Diagrams for TeX

ENCHIRIDION

1.0.0

6th June 2020

CoDi is a TikZ library. Its aim is making commutative diagrams easy to design, parse and tweak.

Preliminaries

TikZ is the only dependency of CoDi. This ensures compatibility with most[1] TeX flavours. Furthermore, it can be invoked both as a standalone and as a TikZ library. Below are minimal working examples for the main dialects.

TeX package

```
\input
  {commutative-diagrams}

\codi
  % diagram here
\endcodi
\bye
```

ConTeXt module

```
\usemodule
  [commutative-diagrams]
\starttext
\startcodi
  % diagram here
\stopcodi
\stoptext
```

LaTeX package

```
\documentclass{article}
\usepackage
  {commutative-diagrams}
\begin{document}
\begin{codi}
  % diagram here
\end{codi}
\end{document}
```

TeX (TikZ library)

```
\input{tikz}
\usetikzlibrary
  [commutative-diagrams]

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

ConTeXt (TikZ library)

```
\usemodule[tikz]
\usetikzlibrary
  [commutative-diagrams]
\starttext
\starttikzpicture[codi]
  % diagram here
\stoptikzpicture
\stoptext
```

LaTeX (TikZ library)

```
\documentclass{article}
\usepackage{tikz}
\usetikzlibrary
  {commutative-diagrams}
\begin{document}
\begin{tikzpicture}[codi]
  % diagram here
\end{tikzpicture}
\end{document}
```

A useful TikZ feature exclusive to LaTeX is externalization. It is an effective way to boost processing times by (re)compiling figures as external files only when strictly necessary.

A small expedient is necessary to use it with CoDi: diagrams must be wrapped in `tikzpicture` environments endowed with the `/tikz/codi` key.

On the side is an example saving the pictures in the `./tikzpics/` folder to keep things tidy.

```
\documentclass{article}

\usepackage
  {commutative-diagrams}
% Or, equivalently:
%\usepackage{tikz}
%\usetikzlibrary
%  {commutative-diagrams}

\usetikzlibrary{external}
\tikzexternalize
  [prefix=tikzpics/]

\begin{document}
\begin{tikzpicture}[codi]
  % diagram here
\end{tikzpicture}
\end{document}
```

∴

Basic knowledge of TikZ is assumed. A plethora of excellent resources exist, so no crash course on the matter will be improvised here. Higher proficiency is not necessary, though recommended: it will make CoDi a pliable framework instead of a black box.

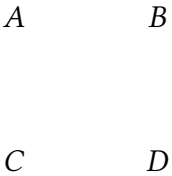
[1] CoDi builds upon TikZ, which builds upon `pgf`, which after version 3.1 requires at least ε-TeX version 2. This is inconsequential except in the unlikely event you're using Knuth's original `tex` format.

Quick tour

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

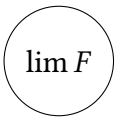

```
\obj {X};
```

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



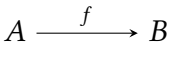

```
\obj {
  A & B \\
  C & D \\
};
```

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



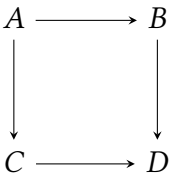

```
\obj {\lim F};
\draw (lim F) circle (4ex);
```

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




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

Commutative diagrams exist to illustrate composition and commutation, so CoDi 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 CoDi.

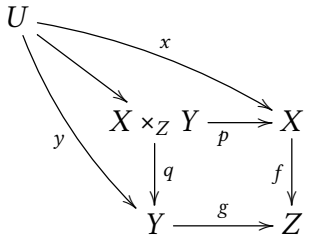
There are more features, though.
Read on if this caught your attention.

Alternatives

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

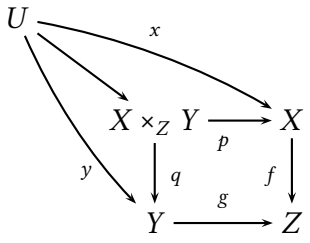
Let Xy-pic set the bar with a *verbatim* extract from its manual.

```
\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                       }
```



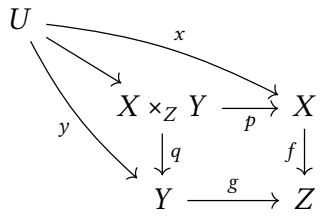
Here is an example adapted from pst-node's documentation.

```
$ \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}
```

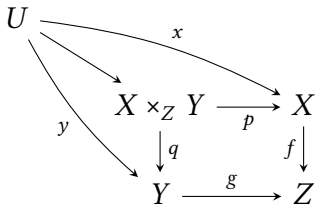


Finally, **CoDi**.

```
\begin{codi}[tetragonal]
  \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{codi}
```



Syntax: objects

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

Orange fragments are optional.

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

Layouts are described using the customary TEX tabular syntax.

Underlined fragments can repeat 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.

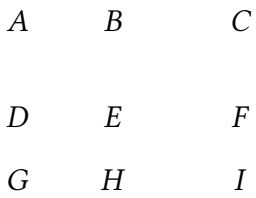
```
‹object options› = [object keylist] (‹name›) at (‹coordinate›)
‹layout options› = [layout keylist] (‹name›) at (‹coordinate›)
```

∴

Nothing of the given syntax is specific to CoDi. In fact, **\obj** can draw both single objects and layouts by behaving like the standard TikZ macros \node and **\matrix** respectively.

Furthermore, layouts content is specified using the common TEX tabular syntax. The only catch is that row and column separators are always mandatory.

Here is a kitchen sink that includes custom spacing:

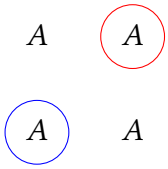



```
\obj {
  A & B &[1em] C \\
  D & E &      F \\[-1em]
  G & H &      I \\
};
```

Here is another one that includes custom options:


```
\obj [red] {
  A & |[blue]| B & C \\
};
```

A standard feature inherited from TikZ worth a mention is the ability to name a layout and refer to cells by their row/column index pairs.




```
\obj (M) { A & A \\ A & A \\ };
\node [draw=red,  shape=circle, minimum size=2em] at (M-1-2) {};
\node [draw=blue, shape=circle, minimum size=2em] at (M-2-1) {};
```

Syntax: morphisms

The second and last macro that CoDi 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›)
```

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

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

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

Alternatives are separated by |s.

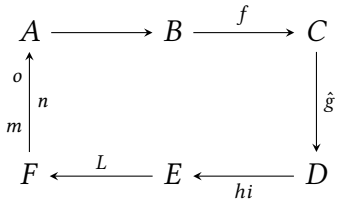
Global options can be given to both labels and arrows.

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

∴

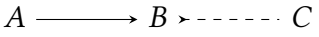
These rules allow for a label syntax that sprouts gracefully from the simplest to the most complex case.

```
\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;
```



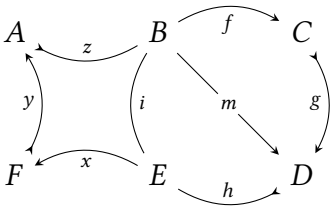
The same holds for arrow syntax.

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



Global options can be used to minimize local ones and keep the code terse and readable.

```
\mor [swap]:[bend left] B f:-> C g:>-> D h:>- E i:- B;
\mor :[bend right] E x:-> F y:>-> A z:>- B;
\mor [mid] B m:-> D;
```



Names

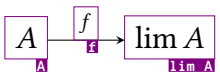
As you'll have guessed by now, objects name themselves.

The process happens in three steps:

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

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

While you're getting acquainted with the process you can use the `/codi/prompter` key to display labels with generated names.



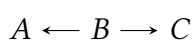

```
\begin{codi}[prompter]
  \obj{ A & \lim A \\ };
  \mor A f:-> (lim A);
\end{codi}
```

Names: shortcuts

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

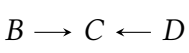
As a source, $*$ evaluates to the head of the previous chain.

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



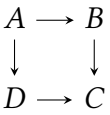
As a target, $*$ evaluates to the tail of the previous chain.

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



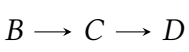
The natural use case for $*$ is chain gluing.

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



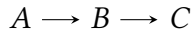
As a source, $+$ evaluates to the tail of the previous chain.

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



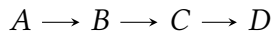
As a target, $+$ evaluates to the head of the previous chain.

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



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

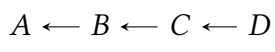
```
\mor B -> C;
\mor A -> + -> D;
```



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

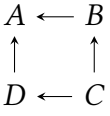
Chain extension can be obtained using $*$.

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



Chain gluing can be obtained using $+$.

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



Names: expansion

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

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

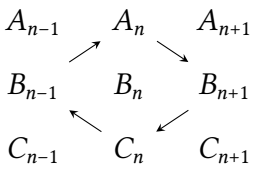
The three available settings correspond to different degrees of expansion. A side by side comparison completely illustrates their meanings.

```
\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;
```

$\therefore$

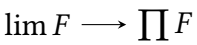
The default behaviour is to avoid expansion in compliance with the principle that *names should be predictable from the* literal *code*. Furthermore, it is seldom wise to liberally expand tokens.

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);
```

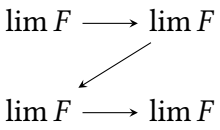
Names: replacement

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

```
/codi/replace character = ‹character› with ‹character›
/codi/replace charcode = ‹charcode› with ‹character›
/codi/remove characters = ‹characters›
/codi/remove character = ‹character›
/codi/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);
```



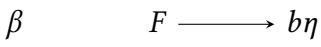
∴

The default behaviour is removal of the minimal set of universally annoying[2] 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.

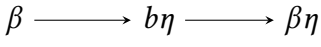
Another egregiously bad idea is replacing characters with spaces. It's tempting because it solves a somewhat common edge case.

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



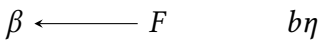
Since characters in names are literal, this causes whitespace duplication and names become inaccessible by ordinary means.

```
\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;
```



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

Names: overwriting

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

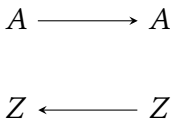
```
/codi/overwrite = false | alias | true
```

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 avoids overwriting explicit labels in order to give you a simple means of naming conflict resolution.



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

Sometimes you might want an object to have both a literal and a semantic alias.

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

The hard overwriting behaviour ignores any label except generated ones; it exists for completeness and debugging purposes.

Styles: scopes

CoDi structures diagrams into five layers implemented with TikZ.

CoDi's	represents an	using TikZ's
`diagram`	(commutative) diagram	`tikzpicture`
`layout`	arrangement of vertices	`matrix`
`object`	vertex	`node`
`arrow`	edge between vertices	`edge`
`label`	label of an edge	`node`

Each layer can be styled using TikZ keys.

Each layer possesses a default style:

```
/codi/every diagram
/codi/every layout
/codi/every object
/codi/every arrow
/codi/every label
```

You can customize them using TikZ key handlers, e. g.

```
/codi/every label/.append style={red}
```

Each layer possesses a library of commonplace styles:

```
/codi/diagrams/
/codi/layouts/
/codi/objects/
/codi/arrows/
/codi/labels/
```

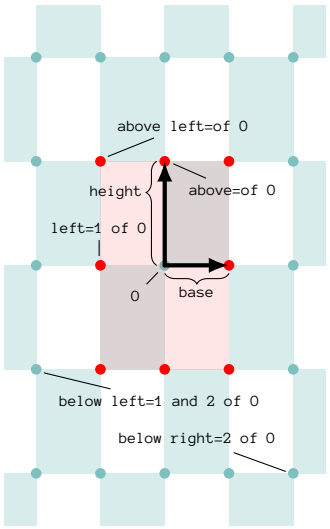
They are the proper place to find styles and define you own:

```
/codi/arrows/fat/.style={ultra thick}
```

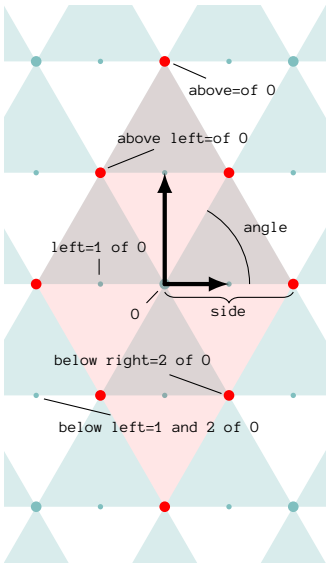
Fully scoping keys is usually unnecessary, as CoDi searches for keys in the library of the layer it's in before falling back to TikZ default search algorithm. Here's some meta code demonstrating this:

```
\begin{codi}[‹diagram keylist›]
  \obj [‹layout keylist›] { |[‹object keylist›]| a & b \\ };
  \obj [‹object keylist›] {x};
  \mor [‹label keylist›]:[‹arrow keylist›]
     a [‹label keylist›]:[‹arrow keylist›] b;
\end{codi}
```

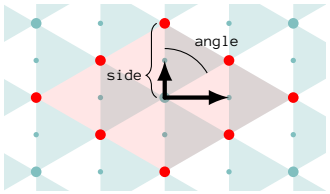
Styles: diagrams



Tetragonal



Hexagonal (horizontal)



Hexagonal (vertical)

Diagrams can be laid over regular grids:

```
/codi/diagrams/tetragonal=base ‹length› height ‹length›
                                (default: base 4.5em height 2.8em)
```

```
/codi/diagrams/hexagonal=‹direction› side ‹length› angle ‹angle›
                          (default: horizontal side 4.5em angle 60)
```

When one of these keys is used

- the versors of the coordinate system are changed,
- the node positioning is set up to lay them on grid,
- and the corresponding key will be applied to all layouts.

The pictures show the key parameters, versors, and a unitary grid.

This setup allows you to mix coordinates and relative positioning keys to arrange objects.

As usual, relative positioning keys can accept two components, a radius, or nothing at all (which defaults to a certain radius).

When using a radius (or defaulting to 1) the tetragonal grid uses Manhattan distance to lay objects along concentric rectangles.

When using a radius (or defaulting to 2) the hexagonal grid[3] uses Chebyshev distance to lay objects along concentric rhombi.

To clarify, a few relative positioning keys are drawn along with red zones displaying the default radii around the origins.

[3] which in truth is built upon a tetragonal grid

Styles: layouts

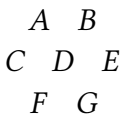
Layouts can be laid over regular grids:

```
/codi/layouts/tetragonal=base ‹length› height ‹length›
                               (default: base 4.5em height 2.8em)
```

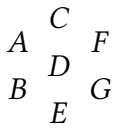
```
/codi/layouts/hexagonal=‹direction› side ‹length› angle ‹angle›
                         (default: horizontal side 4.5em angle 60)
```

When one of these keys is used the layout columns and rows will be spaced and offset in order to reproduce the grids given by diagram styles.

```
\obj [hexagonal=horizontal side 1.5em angle 60] {
  A & B &   \\
  C & D & E \\
  F & G &   \\
};
```




```
\obj [hexagonal=vertical side 1.5em angle 60] {
  A & C & F \\
  B & D & G \\
    & E &   \\
};
```



Note that *each row must have the same number of cells*[4] or the spacing will be incorrect.

[4] this is different from the behaviour of, say, tables

Styles: objects

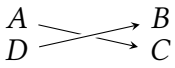
No styles are available at the moment.

Styles: arrows

```
/codi/arrows/crossing over
/codi/arrows/crossing over/clearance=‹length›    (default: 0.5ex)
/codi/arrows/crossing over/color=‹color›         (default: white)
```

This key a provides the configurable illusion of an arrow passing over a *previously drawn* one.

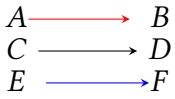
```
\mor A -> C;
\mor :[crossing over] D -> B;
```




```
/codi/arrows/slide=‹length›
```

This key slides an arrow backward (negative) and forward (positive) along its direction of the given length.

```
\mor :[slide=-.3em, red] A -> B;
\mor C -> D;
\mor :[slide=+.3em, blue] E -> F;
```




```
/codi/arrows/shove=‹length›
```

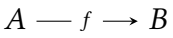
This key shoves an arrow to the left (negative) and to the right (positive) with respect to its direction of the given length.

```
\mor :[shove=-.3em, red] A -> B;
\mor A -> B;
\mor :[shove=+.3em, blue] A -> B;
```

Styles: labels

`/codi/labels/mid`

This key places a label in the middle of an arrow.

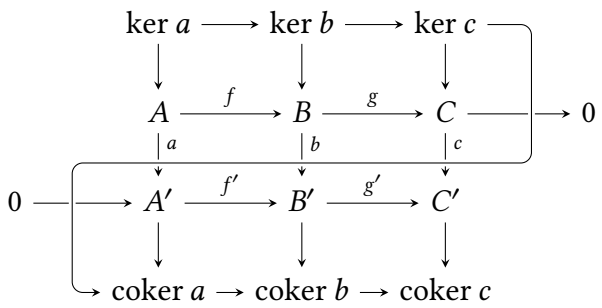


```
\mor [mid] A f:-> B;
```

Gallery

The remainder of the text is just commented examples.

Snake



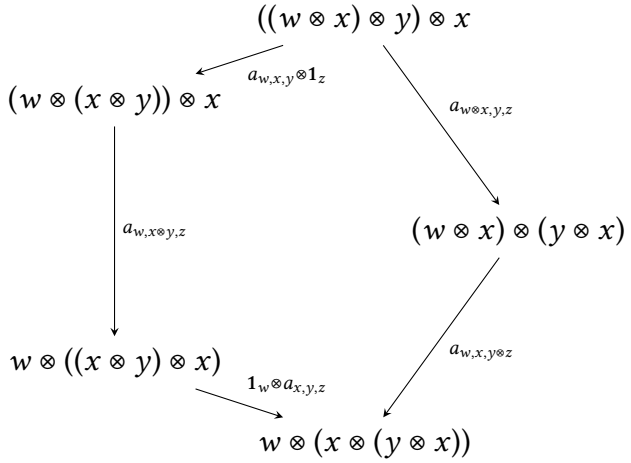

```
\begin{codi}[tetragonal]
  \obj{          & \ker a   & \ker b   & \ker c   &   \\
                 & A        & B        & C        & 0 \\
        |(0')| 0 & A'       & B'       & C'       &   \\
                 & \coker a & \coker b & \coker c &   \\ };

  \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[/codi/arrows/crossing over, ->, rounded corners, >=stealth]
    (ker c) -- ++( 0.6,0) -- ++(0,-1.55)
            -- ++(-3.2,0) -- ++(0,-1.45) -- (coker a);
\end{codi}
```

The fourth associahedron

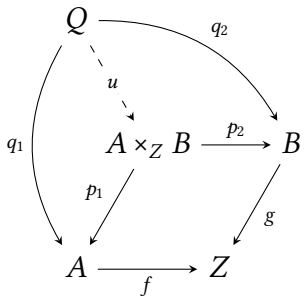



```
\begin{codi}
% From the LaTeX preamble:
%   \usepackage{newunicodechar}
%   \newunicodechar{1}{\mathbf 1}
%   \newunicodechar{×}{\otimes}

  \foreach [count=\n] \o in {
      ((w×x)×y)×x,
      (w×(x×y))×x,
      w×((x×y)×x),
      w×(x×(y×x)),
      (w×x)×(y×x)
    } \obj (\n) at (72*\n:7em) {\o};

  \mor 1 "a_{w,x,y}×1_z": -> 2
           "a_{w,x×y,z}": -> 3
         "1_w×a_{x,y,z}": -> 4;
  \mor *   "a_{w×x,y,z}": -> 5
           "a_{w,x,y×z}": -> *;
\end{codi}
```

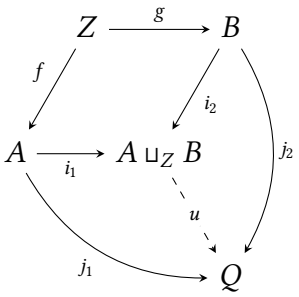
Pullback & pushout




```
\begin{codi}[hexagonal]
  \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{codi}
```

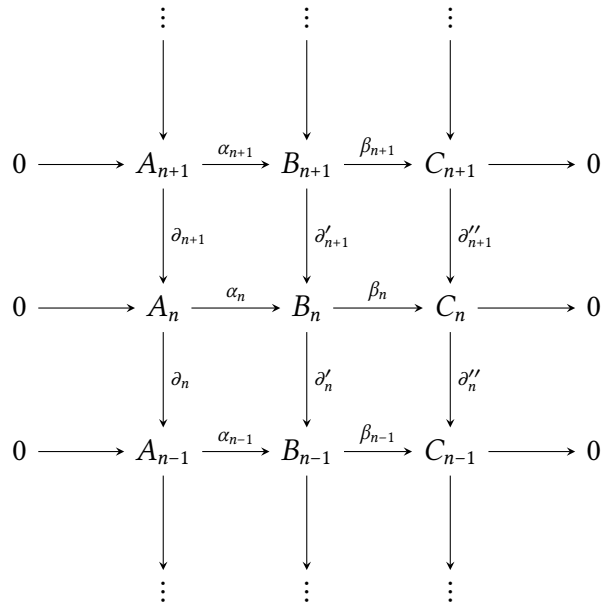



```
\begin{codi}[hexagonal]
  \obj{ Z &                   B \\
        A & |(po)| A \sqcup_Z B \\ };
  \obj[below right=of po] {Q};

  \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{codi}
```

Complexes sequence

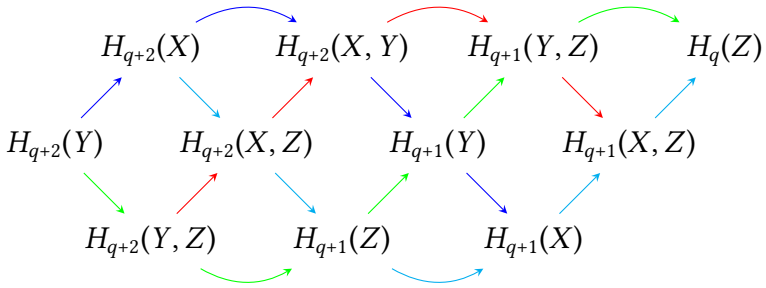



```
\begin{codi}
  \obj (M) {   & \vdots  & \vdots  & \vdots  &   \\
             0 & A_{n+1} & B_{n+1} & C_{n+1} & 0 \\
             0 & A_{n}   & B_{n}   & C_{n}   & 0 \\
             0 & A_{n-1} & B_{n-1} & C_{n-1} & 0 \\
               & \vdots  & \vdots  & \vdots  &   \\ };

  \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{codi}
```

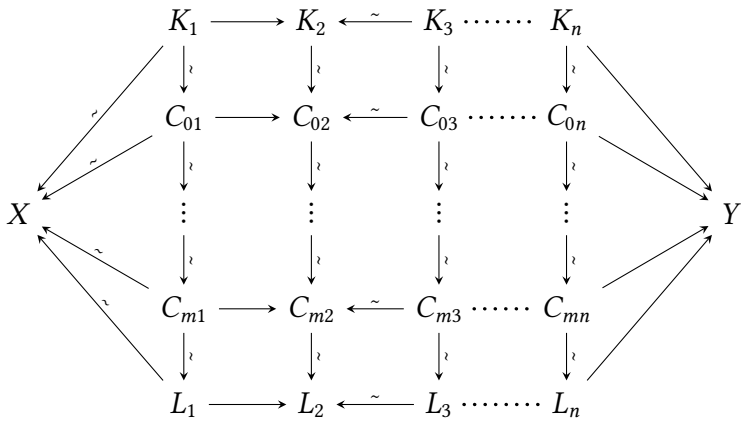
Braid




```
\begin{codi}[l/.style={bend left}, r/.style={bend right} ]
  \obj [ hexagonal=horizontal side 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{codi}
```

Hammock




```
\begin{codi}[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{codi}
```