"Byrne" v. 0.2.5 package for METAPOST and LaTeX

Sergey Slyusarev

February 25, 2026

Abstract

This document describes "Byrne" package for METAPOST and LaTeX.

1 Introduction

Oliver Byrne's 1847 edition on the first six books of Euclid's "Elements"[?] is an interesting example of tight interplay between text and graphics in information visualization. The main feature of this book is that instead of relying on letter designations to describe lines, angles, etc., all the diagrams in it are colored and parts of these diagrams are directly incorporated into text.

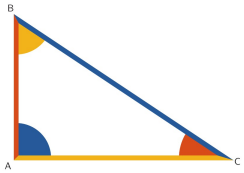
Recently this work met somewhat of a renaissance and in the span of a couple of years at least three independent attempts were made at reviving Byrne's work. In early 2017 the first six books remade in ConTeXt were published on GitHub, and later migrated to LaTeX[?], in late 2018 an interactive web-version was published by Nicholas Rougeux[?] and in late 2019 a printed version of all Euclid's 15 books was published by Kronecker Wallis. All three are very different in terms of approach and goals.

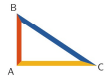
This package is a LaTeX (specifically LuaLaTeX) version of the toolkit devised for the TeXversion of the book. It consists of two parts: METAPOST code to describe and render geometric constructions and LaTeX code to incorporate these constructions in text.

2 Requirements

This package works with LuaLaTeX and requires `luamplib v2.23.0` or higher.

3 Example



In  the angle  is a right angle, therefore  is a right-angled triangle. In it and .

The construction above is described like this:

```
\usepackage{byrne}
...
\defineNewPicture{ % MetaPost code to describe the main diagram is placed inside
  textLabels := true; % This turns text labels on
  pair A, B, C;
  A := (0, 0);
  B := (0, 2cm);
  C := (3cm, 0);
  byAngleDefine(C, B, A, byyellow, 0); % This defines a yellow angle CBA
  byAngleDefine(B, A, C, byblue, 0);
  byAngleDefine(A, C, B, byred, 0);
  draw byNamedAngleResized(); % Draws angles.
  byLineDefine(A, B, byred, 0, 0);    % This defines a red line AB.
  byLineDefine(B, C, byblue, 0, 0);
  byLineDefine(C, A, byyellow, 0, 0);
  draw byNamedLineSeq(1)(AB,BC,CA); % Draws joint lines in sequence
  draw byLabelsOnPolygon(A, B, C)(0, 0); % Draws text labels
}
\drawCurrentPicture % Draws the diagram

In
\drawFromCurrentPicture[middle][triangleABC]{
  startAutoLabeling;
  draw byNamedLineSeq(0)(AB,BC,CA);
  stopAutoLabeling;
}
the angle \drawAngle{A} is a right angle, therefore \triangleABC\ is
a right-angled triangle. In it $\drawUnitLine{CB} > \drawUnitLine{AB}$
and $\drawUnitLine{BC} > \drawUnitLine{AC}$.
```

Several things to note here.

In the METAPOST part: All points are ordinary METAPOST pairs and can be defined and manipulated as such. Normally every part of the diagram is first defined (e.g. `byAngleDefine` or `byLineDefine`) and then is drawn (e.g. `draw byNamedAngleResized...` or `draw byNamedLineSeq...`). Also, although Byrne didn't use text labels in his book, they are implemented in this package **??**. Normally they are placed semi-manually in the main diagram (using functions like `byLabelsOnPolygon`) and mostly automatically in offspring pictures. By default they are turned off, and to turn them on `textLabels := true;` should be set.

In the LaTeX part: The most general way to reference main diagram is to use `\drawFromCurrentPicture`. It will process arbitrary METAPOST code based

on the main diagram's code and output an image. However, it would be too cumbersome to use this route every time, so a few shortcuts are in place. If you do need to process arbitrary code, you can give picture a name (in the example it's `triangleABC`). It will define a macro of the same name `\triangleABC` which will output the same image again. For most common cases there are special macros which don't require you to write any METAPOST code. For instance `\drawAngle` and `\drawUnitLine` allow you to call angles and lines simply by name. The names don't have to match the initial definition exactly. For instance, the angle in the example can be called `A`, `BAC` or `CAB` interchangeably, or the same line segment can be called either `BC` or `CB`.

For more examples you can look at the source code of the ConTEXt version of Byrne's book. This package shares syntax with it and its code can be transferred to LATEX with minimal changes.

The following reference guide is not complete and will be updated in the future.

4 METAPOST part reference

4.1 Global variables and settings

4.1.1 Global variables

There are several variables which control the looks and behaviour.

`lineWidth` — the width for regular lines, set to `2pt` by default. Normally referenced as `0`.

`lineWidthThin` — the width for thin lines, set to `1pt` by default. Normally referenced as `1`.

`lineWidthHair` — the width for superthin lines, set to `1/2pt` by default. Normally referenced as `2`.

`pointMarkSize` — size of point marks, set to `4pt` by default.

`pointLinesSize` — size for line segments used to depict points, set to `1/2cm` by default.

`defaultScaleFactor` — global default scale factor, set to `1` by default.

`angleSize` — angle arc radius, set to `1cm` by default.

`angleScale` — scale factor for angle arcs, set to `1` by default.

`globalRotation` — angle to rotate the whole picture.

`markLength` — size of marks on lines, set to `3lineWidth` by default.

`rayExtension` — length for a ray depiction, set to `1/3cm` by default.

`textLabels` — whether to render text labels, set to `false` by default.

`autoRightAngles` — whether to depict right angles using angles instead of arcs, set to `false` by default.

`textLabelShift` — how far away to put text labels from points, set to `lineWidth` by default.

`compensateLineLabels` — a boolean to determine if inline images of line segments should be centered regardless of whether line labels are turned on or not, set to `true` by default.

`lineLabelsOnTop` — a boolean to determine if inline images of line segments should have labels on top, if labels are turned on, set to `true` by default.

`angleClockwiseMode` — a boolean to determine if compound angles should be constructed clockwise, set to `false` by default.

`angleModeFallback` — a boolean to determine if compound angles should fallback to the opposite direction if current direction fails, set to `false` by default.

4.1.2 defineColor.ColorName(color)

Defines a color. Colors can be defined as regular METAPOST colors as well, this is just a shorthand version.

`ColorName` — mandatory color name of `suffix type`.

`color` — an actual calor, can be either of `color` type or of `cmykcolor` type.

Some colors are predefined. Colors similar to the ones Byrne used for his book are: `byblack`  , `byred`  , `byblue`  , `byyellow`  . One obvious issue of any color coding is whether the colors are discernible by colorblind people. Byrne's colors should be mostly fine, but there are only foru of them. In case more colors are necessary, eight colors from colorblind-friendly palette by Okabe and Ito [**?**] are defined: `oiBlack`  , `oiOrange`  , `oiSkyBlue`  , `oiGreen`  , `oiYellow` , `oiBlue`  , `oiVermillion`  , `oiPurple`  . Also text labels **??** are helpful as a secondary means of identification.

4.1.3 startTempScale(tmpScale); ... METAPOST code ... stopTempScale;

Used to temporarily set scale factor for a section of code;

`tmpScale` — scale factor for a section of code.

4.1.4 startTempAngleScale(tmpAngleScale); ... METAPOST code ... stopTempAngleScale;

Used to temporarily set angle scale factor for a section of code;

`tmpAngleScale` — angle scale factor for a section of code.

4.1.5 startGlobalRotation(rotationAngle); ... METAPOST code ... stopGlobalRotation;

Used to temporarily set global rotation for a section of code;

`rotationAngle` — global rotation for a section of code.

4.1.6 startAutoLabeling; ... METAPOST code ... stopAutoLabeling

Used to turn auto labeling for a section of code. Most of the drawing functions have auto labeling which is turned off by default. Sometimes it's handy to use it instead of labeling items by hand.

4.1.7 startAngleOppositeMode; ... METAPOST code ... stopAngleOppositeMode

Used to temporarily switch the default direction for compound angles.

4.2 Lines

4.2.1 byLineDefine.LineName(A, B, color, dashed, thick)

Defines a straight line. Does not return anything.

`LineName` — optional line name of `suffix type`. If no name is provided, it is being constructed from point names, e.g. `AB`.

`A, B` — point names, of `suffix` type (i.e., variable names should be provided).

`color` — line color, of `color` type.

`dashed` — is line dashed. `0` for solid line and `1` for dashed line.

`thick` — is line thick. `0` for regular line and `1` for thin line `??`.

For example, this line

is defined as `byLineDefine(A, B, byred, 0, 0);` and this line

is defined as `byLineDefine(C, D, byblue, 1, 1);`

4.2.2 byLine.LineName(A, B, color, dashed, thick)

Defines and draws a straight line. Returns a `picture` of the line.

Arguments are the same as in `byLineDefine` `??`.

4.2.3 byNamedLine(LINES_LIST)

Draws lines. Returns a picture of the lines.

`LINES_LIST` — a comma-separated list of line names.

4.2.4 byNamedLineSeq(lineShift)(LINES_LIST)

Draws lines joined in sequence. Line joints look like this:  . Returns a picture of the lines.

`lineShift` — how much the lines are shifted perpendicular lines' axis. `1` is for one regular line thickness to one side  , `-1` is for one regular line thickness to the other side  .

`LINES_LIST` — a comma-separated list of line names.

4.2.5 byNamedLineSeqByPoint(lineShift, isCyclic)(POINTS_LIST)

Draws lines joined in sequence, just as `byNamedLineSeq`, but built using a list of points.

`lineShift` — how much the lines are shifted perpendicular lines' axis. `1` is for one regular line thickness to one side

`isCyclic` — sould the sequence be closed, boolean. Another way to prevent closing of the sequence is to add `noPoint` as the last point

`POINTS_LIST` — a comma-separated list of point names.

4.2.6 byMarkLine(position, color)(LineName)

Defines and draws a mark on a line.

`position` — a position of the mark. `0` for one end of the line, `1` for the other end of the line.

`color` — mark color, of `color` type.

`LineName` — line name, of `suffix` type.

For example this line mark 

is drawn with `draw byMarkLine(1/4, byred, AB);`.

4.2.7 byNamedMarkLine(LineName)

Draws a defined mark on the given line.

`LineName` — line name, of `suffix` type.

4.3 Points

4.3.1 byPointLabelDefine(A, pointLabel)

Defines a point text label, returns nothing. By default point names are the same as the names of respective variables.

`A` — a name of a point of `suffix` type (i.e., variable name should be provided).

`pointLabel` — point label of `string` type.

4.3.2 byPointLabelRemove(POINTS_LIST)

Sets text labels of multiple points to blank. Useful if text labels are turned on, but you don't want labels on certain points.

`POINTS_LIST` — a comma-separated list of point names.

4.3.3 byPointMarkDefine(PointName)(color, style)

Defines a point mark.

`color` — mark color, of `color` type.

`style` — `0` for solid circle •, `1` for white circle with outline ○. Mark size is controlled by `pointMarkSize ??`.

4.4 Arcs and circles

4.4.1 byArcDefineBE.ArcName(O, begin, end, radius, color, dashed, thick, shift, endType)

Defines an arc based on its center, radius and two angles. Returns nothing.

`ArcName` — optional arc name of `suffix type`. If no name is provided, it is being constructed from center point name, e.g. `O`.

`O` — the center of the circle an arc is from, of `suffix` type (i.e., variable name should be provided).

`begin, end` — beginning and ending of the arc in eighths (in essence, `arctime` of `fullcircle`). `0` for 3 o'clock, `2` for 12 o'clock etc. For instance, this  is `0, 3` and this is `5, 7`.

`radius` — arc radius.

`color` — arc color, of `color` type.

`dashed` — is arc line dashed. `0` for solid line and `1` for dashed line.

`thick` — is arc line thick. `0` for regular line and `1` for thin line `??`.

`shift` — how much arc line is shifted. `1` is for one regular line thickness outwards  , `-1` is for one regular line thickness inwards  .

`endType` — type of arc line ends. `0` for ends cut by the radii  and `1` for ends cut by the chord  .

4.4.2 byArcBE.arcName(O, begin, end, radius, color, dashed, thick, shift, endType)

Defines and draws an arc based on its center, radius and two angles. Returns the picture of the arc.

Arguments are the same as in `byArcDefineBE` **??**.

4.4.3 byArcDefine.ArcName(O, B, E, radius, color, dashed, thick, shift, endType)

Defines an arc based on its center, radius and two points. Returns nothing.

Arguments are the same as in `byArcDefineBE` **??**, except for instead of angles in eights points are used.

`ArcName` — optional arc name of `suffix type`. If no name is provided, it is being constructed from point names, e.g. `BOE`.

`B, E` — points in the direction of the beginning and the ending of the arc respectively. Of `suffix` type (i.e., variable names should be provided).

4.4.4 byArc.ArcName(O, B, E, radius, color, dashed, thick, shift, endType))

Arguments are the same as in `byArcDefine` **??**.

4.4.5 byNamedArcExact(ARCS_LIST)

Draws a picture of arcs just as they were defined. Returns a picture of the arcs.

`ARCS_LIST` — a comma-separated list of arc names.

4.4.6 byNamedArcSeq(arcShift)(ARCS_LIST)

Draws arcs joined in sequence (only affects dash patterns). Returns a picture of the arcs.

`arcShift` — how much the arcs are shifted outwards. `1` is for one regular line thickness outwards.

`ARCS_LIST` — a comma-separated list of arc names.

4.4.7 byNamedArc(ARCS_LIST)

Draws a picture of arcs, except of it ignores `shift` and `endType`. Returns a picture of the arcs.

`ARCS_LIST` — a comma-separated list of arc names.

4.4.8 byCircleDefineFree.CircleName(o, radius, color, dashed, thick, shift)

Defines a circle based on its center and radius. Returns nothing.

`CircleName` — optional circle name of `suffix type`. If no name is provided, `aCircle` is used for a name.

`o` — the center of the circle, of `pair` type.

`radius` — circle radius.

`color` — circle color, of `color` type.

`dashed` — is circle line dashed. `0` for solid line and `1` for dashed line.

`thick` — is circle line thick. `0` for regular line and `1` for thin line **??**.

`shift` — how much circle line is shifted. `1` is for one regular line thickness outwards  , `-1` is for one regular line thickness inwards  .

4.4.9 byCircleDefineR.CircleName(O, radius, color, dashed, thick, shift)

Defines a circle based on its center and radius. Returns nothing.

Arguments are the same as in `byCircleDefineFree` **??**, except for the center is not defined directly by a `pair`.

`CircleName` — optional circle name of `suffix type`. If no name is provided, it is being constructed from center name, e.g. `O`.

`O` — the center of the circle. Of `suffix` type (i.e., variable name should be provided).

4.4.10 byCircleR.CircleName(O, radius, color, dashed, thick, shift)

Defines and draws circle based on its center and radius. Returns a picture of that circle.

Arguments are the same as in `byCircleDefineR` **??**.

4.4.11 byCircleDefine.CircleName(O, A, color, dashed, thick, shift)

Defines circle based on its center and a point. Returns nothing.

Arguments are the same as in `byCircleDefineR` **??**, except for instead of the radius, a point through which the circle is drawn is used.

`CircleName` — optional circle name of `suffix type`. If no name is provided, it is being constructed from point names, e.g. `OA`.

`A` — a point through which the circle is drawn. Of `suffix` type (i.e., variable name should be provided).

4.4.12 byCircle.CircleName(O, A, color, dashed, thick, shift)

Defines and draws circle based on its center and a point. Returns a picture of that circle.

Arguments are the same as in `byCircleDefine` **??**.

4.4.13 byCircleABC.CircleName(A, B, C, color, dashed, thick, shift)

Arguments are the same as in `byCircleDefineR` **??**, except for instead of the center and the radius, three points through which the circle is drawn are used.

`CircleName` — optional circle name of `suffix type`. If no name is provided, it is being constructed from point names, e.g. `ABC`.

`A, B, C` — points through which the circle is drawn. Of `suffix` type (i.e., variable names should be provided).

4.4.14 byNamedCircle(CIRCLES_LIST)

Draws circles. Returns a picture of the circles.

`CIRCLES_LIST` — a comma-separated list of circle names.

4.5 Arbitraty figures

4.5.1 byArbitraryFigureDefine.ArbitraryFigureName(figurePath, color, dashed, thick)

Defines an arbitrary figure (basically, any path). Returns nothing.

`ArbitraryFigureName` — optional figure name of `suffix type`. If no name is provided, `anArbitraryFigure` is used for a name.

`figurePath` — figure path, of `path` type.

`color` — figure color, of `color` type.

`dashed` — is figure line dashed. `0` for solid line and `1` for dashed line.

`thick` — is figure line thick. `0` for regular line and `1` for thin line **??**.

4.5.2 byArbitraryFigure.ArbitraryFigureName(figurePath, color, dashed, thick)

Defines an arbitrary figure. Returns nothing.

Arguments are the same as in `byArbitraryFigure` **??**.

4.5.3 byNamedArbitraryFigure(ARBITRARY_FIGURES_LIST)

Draws arbitrary figures. Returns a picture of the figures.

`ARBITRARY_FIGURES_LIST` — a comma-separated list of figure names.

4.6 Filled figures

4.6.1 byPolygonDefine.PolygonName(POINTS_LIST)(Color)

Defines a polygon based on its vertex points. Returns nothing.

`PolygonName` — optional polygon name `suffix`.

`POINTS_LIST` — a comma-separated list of point names.

`Color` — line color, of `suffix` type (i.e. a variable name should be provided).

4.6.2 byPolygon.polygonName(POINTS_LIST)(Color)

Defines and draws a polygon based on its vertex points. Returns a picture of the polygon.

Arguments are the same as in `byPolygonDefine` **??**.

4.6.3 byNamedPolygon(POLYGONS_LIST)

Draws polygons. Returns a picture of the polygons.

`POLYGONS_LIST` — a comma-separated list of polygon names.

4.7 Angles

4.7.1 byAngleDefine.AngleName(A, B, C, color, style)

Defines an angle based on three points. Returns nothing.

`AngleName` — optional angle name `suffix`.

`A, B, C` — angle points. Of `suffix` type (i.e., variable names should be provided).

`color` — angle color, of `color` type.

`style` — angle arc style. `0` for solid filled sector  , `1` for arc  , `2` for dashed arc  . Default arc radius is determined by `angleSize` **??**. It can be temporarily changed using `angleScale` **??**. You can also define custom angle styles **??**.

4.7.2 byConsiderAngleRight(AngleName)

Makes a defined angle look like an angle  instead of a circle arc  , as right angles are often depicted. Useful if an angle is not actually right in the diagram, but is meant to be right or assumed to be right. To make all the right angles look like this automatically, you can set `autoRightAngles := true;` **??**.

`AngleName` — name of the angle.

4.7.3 byAngle.AngleName(A, B, C, color, style)

Defines and draws an angle based on three points. Returns a picture of the angle.

Arguments are the same as in `byAngleDefine` **??**.

4.7.4 byAngleDefineExtended.AngleName(A, B, C, color, style)(OPTIONAL_COLORS_LIST)

Defines an angle based on three points. Returns nothing.

Arguments are the same as in `byAngleDefine` **??**, except for using this function it's possible to define optional colors for an angle.

`OPTIONAL_COLORS_LIST` — a comma-separated list of optional color names. See **??** for details.

4.7.5 byAngleExtended.AngleName(A, B, C, color, style)(OPTIONAL_COLORS_LIST)

Defines and draws an angle based on three points. Returns a picture of the angle.

Arguments are the same as in `byAngleDefineExtended` **??**.

4.7.6 byNamedAngle(ANGLES_LIST)

Draws angles. Returns a picture of the angles.

`ANGLES_LIST` — a comma-separated list of polygon names.

4.7.7 Angle styles

It's possible to add custom angle styles. To do this, you first need to define a function, with a particular set of arguments and then include this function's name into a special list. As an example here's the function which describes a regular solid angle sector:

```
vardef byAngleMSolid (expr angleArc, angleColor)(suffix AngleOptionalColors) =
    save p;
    path p;
    p := angleArc scaled (angleScale*angleSize);
    image(
        fill ((0, 0)--p--cycle) withcolor angleColor;
    )
enddef;
```

Note the argument `AngleOptionalColors`. It is not used for solid angles and is not defined with `byAngleDefine`, but several optional colors can be defined using `byAngleDefineExtended` ?? and can be accessed using `AngleOptionalColors[0]`, `AngleOptionalColors[1]` etc.

And here's how this angle style is included in the list:

```
byAngleMacroName[0] := "byAngleMSolid";
```

Here `0` is the value of `sty` argument used in `byAngleDefine` ?? and such to call this particular angle style.

As a more practical example, in Byrne's book, in a couple of places for some reason he chooses not to use ordinary arcs and sectors, but instead goes with unique angle designs. In order not to include these peculiarities into the main lib in the ConTEXt version they were defined *in situ*. Here's the definition from IV.III:

```
vardef byAngleMSectors (expr angleArc, angleColor)(suffix angleOptionalColors) =
    save p;
    path p[];
    p1 := (subpath(0, arctime (1/3arclength(angleArc)) of angleArc)
        of angleArc)
        scaled (angleScale*angleSize);
    p2 := (subpath(arctime (1/3arclength(angleArc))
        of angleArc, arctime(2/3arclength(angleArc)) of angleArc) of angleArc)
        scaled (angleScale*angleSize);
    p3 := (subpath(arctime (2/3arclength(angleArc))
        of angleArc, length(angleArc)) of angleArc)
        scaled (angleScale*angleSize);
    image(
        fill ((0, 0)--p1--cycle) withcolor angleOptionalColors[0];
        fill ((0, 0)--p2--cycle) withcolor angleColor;
        fill ((0, 0)--p3--cycle) withcolor angleOptionalColors[1];
    )
enddef;
byAngleMacroName[3] := "byAngleMSectors";
```

This angle style employs two optional colors and can be called by setting `style` argument ?? to `3`. This is what the result of `byAngleDefineExtended(A, B, C, byred, 3)(byblue, byyellow);` (??) looks like:  .

4.8 Text labels

There are no text labels in Byrne's book, but, as Edward Tufte points out in his 1990 book, adding small text labels can actually help speed recognition of geometric elements [?]. It's not clear whether it's indeed the case for the reader, but at the very least text labels provide the author with the means of linking source code with the rendered pages. Several functions listed below allow for adding text labels very similar to the ones suggested by Tufte.

Text labels are turned off by default. As a result, simply adding code for text labels won't make them visible. They only appear when `textLabels := true;` is set in the beginning of a main picture or globally by defining `\def\{...} mpPre` ??. Text labels don't always have to be added manually, most elements can do auto labeling ?? which is intended to work with standalone angles, polygons, etc.

4.8.1 byLabelPoint(A, labelAngle, distance)

Draws a text label for a point. Returns a picture of the label.

`A` — the point to label, of `suffix` type (i.e., point name should be provided).

`labelAngle` — angle at which the label should be drawn in degrees.

`distance` — distance at which the label should be drawn in `textLabelShift` units.

4.8.2 byLabelLine(distance)(LINES_LIST)

Draws text labels for lines. Returns a picture of the labels.

`distance` — distance at which the labels should be drawn in `textLabelShift` units.

`LINES_LIST` — a comma-separated list of line names.

4.8.3 byLabelPolygon(distance)(POLYGONS_LIST)

Draws text labels for polygons. Returns a picture of the labels.

`distance` — distance at which the labels should be drawn in `textLabelShift` units.

`POLYGONS_LIST` — a comma-separated list of polygon names.

4.8.4 byLabelsOnCircle(POINTS_LIST)(CircleName)

Draws text labels for points on a circle. Returns a picture of the labels.

`POINTS_LIST` — a comma-separated list of point names.

`CircleName` — circle name of `suffix type`.

4.8.5 byLabelsOnPolygon(POINTS_LIST)(style, shift)

Draws text labels for points on a polygon. Returns a picture of the labels.

`POINTS_LIST` — a comma-separated list of point names.

`style` — specifies which points to omit. `0` to draw labels for all listed points 

, `1` to draw labels for all listed points, except for the ones at straight angles 

, `2` omits first and last points 

, `3` omits the first point 

, `4` omits the last point 

.

`shift` — negative to move labels closer, positive to move labels farther.

4.8.6 byLabelLineEnd(A, B)(distance)

Draws a text label for a line end. Returns a picture of the label.

`A, B` — line ends, of `suffix` type (i.e., point name should be provided).

`distance` — negative to move labels closer, positive to move labels farther.

5 LaTeX part reference

5.1 General purpose

5.1.1 \defineNewPicture [offspringPictureScaleFactor] [mainPictureScaleFactor] {METAPOST code}

This macro is used to describe a main diagram.

`offspringPictureScaleFactor` — how much smaller offspring pictures should be. `1/3` by default.

`mainPictureScaleFactor` — scale factor for the main picture. `1` by default.

METAPOST `code` — actual METAPOST code to describe the main picture. To have some snippets of code always executed before and after the code for the main picture (e.g. to turn on text labels by default with `textLabels := true;`), define `\mpPre` and `\mpPost` macros respectively in the preamble of your document, like this:

```
\def\mpPre{...some \METAPOST\ code...}
\def\mpPost{...some \METAPOST\ code...}
```

5.1.2 \defineNewPictureBasedOnOld [offspringPictureScaleFactor] [mainPictureScaleFactor] {METAPOST code}

Works exacly like **??**, but instead of creating a new instance, it uses the same instance as previously defined picture, retaining all its variables and definitions.

5.1.3 \drawCurrentPicture

Draws the main diagram.

5.1.4 \defineFromCurrentPicture {verticalAlignment} {PictureName} {METAPOST code}

This macro is used to describe a diagram, based on the main diagram.

`verticalAlignment` — vertical alignment declaration. `middle` to align the picture with the middle of the line, anything else for bottom alignment.

`PictureName` — picture name.

`MP code` — actual METAPOST code to describe the picture.

Can be drawn using `\PictureName`.

5.1.5 \drawFromCurrentPicture [verticalAlignment] [PictureName] {METAPOST code}

This macro is used to describe and draw a diagram, based on the main diagram.

`verticalAlignment` — optional vertical alignment declaration. `middle` to align the picture with the middle of the line, anything else for bottom alignment.

`PictureName` — optional picture name.

METAPOST `code` — actual METAPOST code to describe the picture.

Can be drawn again later using `\PictureName`.

5.2 Specialized

5.2.1 \drawUnitLine[LineSecondName]{LineName}

Draws a line of unit length with colors and labels based on the line of the same name in the main diagram.

`LineSecondName` — optional line name.

`LineName` — name of the line to reference in the main diagram.

If a name is given, can be called later using `\ulineLineNameLineSecondName`, otherwise — `\ulineLineName`.

5.2.2 \drawProportionalLine{LineName}

Draws a line with colors and labels based on and length linearly proportional to the line of the same name in the main diagram.

`LineName` — name of the line to reference in the main diagram.

Can be called later using `\plineLineName`.

5.2.3 \drawSizedLine[LineSecondName]{LineName}

Draws a line with colors and labels based on and length proportional to the line of the same name in the main diagram.

`LineSecondName` — optional line name.

`LineName` — name of the line to reference in the main diagram.

If a name is given, can be called later using `\slineLineNameLineSecondName`, otherwise — `\slineLineName`.

5.2.4 \drawUnitRay[RayName]{LineName}

Draws a ray of unit length with colors and labels based on the line of the same name in the main diagram.

`RayName` — optional ray name.

`LineName` — name of the line to reference in the main diagram.

If a name is given, can be called later using `\urayLineNameRayName`, otherwise — `\urayLineName`.

5.2.5 \drawRightAngle, \drawTwoRightAngles

Draws a right angle  and two right angles  respectively.

More broadly, following generic right angles are defined: `rightAngleNE` , `rightAngleES`  , `rightAngleSW`  , `rightAngleWN`  .

If you choose to go with angles instead of arcs for right angles (`??`, `??`), adding `autoRightAngles := true;` to `\mpPre ??` won't change the appearance of these angles, since they are defined before `\mpPre` commands are executed. Instead, these angles should be redefined. There's a special command for this purpose: `byDefineGenericRightAngles(true);` where `true` tells this command to run `byConsiderAngleRight ??` for each of the generic right angles.

5.2.6 \drawAngle{AngleName}

Draws an angle.

`AngleName` — name of the angle to reference in the main diagram. When just one letter is used, all the angles defined at this point will be drawn, e.g.  . When used to reference one single defined angle, the direction doesn't matter. E.g. `AOB` and `BOA` will return the same picture 

. When used to reference an angle composed of several defined angles, the name sould be given counterclockwise. E.g. `\drawAngle{AOD}` is 

and `\drawAngle{DOA}` is 

.

Can be called later using `\angleAngleName`.

5.2.7 \drawAngleWithSides {AngleName}

Draws an angle and sections of its adjacent sides placed if angle portion style is not `0`. E.g., this angle  is drawn with `\drawAngle{B}`, and this angle  is drawn with `\drawAngleWithSides{B}`.

`AngleName` — the same as in **??**.

Can be called later using `\anglewithsidesAngleName`.

5.2.8 \drawPolygon[verticalAlignment][PolygonNewName] {PolygonName}

Draws a polygon.

`verticalAlignment` — optional vertical alignment declaration. `middle` to align the picture with the middle of the line, anything else for bottom alignment.

`PolygonNewName` — optional polygon name.

`PolygonName` — name of the polygon to reference in the main diagram.

If a name is given, can be called later using `\PolygonNewName`, otherwise — `\polygonPolygonName`.

5.2.9 \drawCircle[verticalAlignment][scaleFactor]{CircleName}

Draws a circle.

`verticalAlignment` — optional vertical alignment declaration. `middle` to align the picture with the middle of the line, anything else for bottom alignment.

`scaleFactor` — optional scale factor for the circle to replace the default offspring picture scale factor.

`CircleName` — name of the circle to reference in the main diagram.

Can be called later using `\circleCircleName`.

5.2.10 \drawArc[verticalAlignment][scaleFactor]{ArcName}

Draws an arc.

`verticalAlignment` — optional vertical alignment declaration. `middle` to align the picture with the middle of the line, anything else for bottom alignment.

`scaleFactor` — optional scale factor for the arc to replace the default offspring picture scale factor.

`ArcName` — name of the arc to reference in the main diagram.

Can be called later using `\arcCircleName`.

5.2.11 \drawLine[verticalAlignment][LineNewName]{LineName}

Draws a line.

`verticalAlignment` — optional vertical alignment declaration. `middle` to align the picture with the middle of the line, anything else for bottom alignment.

`LineNewName` — optional line name.

`LineName` — name of the line to reference in the main diagram.

If a name is given, can be called later using `\LineNewName`, otherwise — `\lineLineName`.

5.2.12 \drawLineByPoints[verticalAlignment][LineNewName]{PointNames}

Draws a line, built, using a list of points.

`verticalAlignment` — optional vertical alignment declaration. `middle` to align the picture with the middle of the line, anything else for bottom alignment.

`LineNewName` — optional line name.

`PointNames` — names of the points, constituting the line.

If a name is given, can be called later using `\LineNewName`, otherwise — `\lineLineName`.

5.2.13 \drawPointM{PointName}

Draws a point, depicted as a small circle  .

`PointName` — name of the point to reference in the main diagram.

Can be called later using `\pointmPointName`.

5.2.14 \drawPointL[verticalAlignment][LinesToOmit]{PointName}

Draws a point, depicted as line segments starting/ending in it  . Only the lines which were actually drawn, not just defined, are used. Line segments' length is controlled by `pointLinesSize` **??**.

`verticalAlignment` — optional vertical alignment declaration. `middle` to align the picture with the middle of the line, anything else for bottom alignment.

`LinesToOmit` — a comma-separated list lines, segments of which not to draw.

`PointName` — name of the point to reference in the main diagram.

If a list of lines to omit is given, can be called later using `\pointlPointName-MinusLinesToOmit`, otherwise — `\pointlPointName`.

5.2.15 \drawPoint[verticalAlignment][LinesToOmit]{PointName}

Draws a point, depicted as line segments starting/ending in it with a point mark if defined  . Only the lines which were actually drawn, not just defined, are used.

Arguments are the same as in `drawPointL` **??**.

If a list of lines to omit is given, can be called later using `\pointPointName-MinusLinesToOmit`, otherwise — `\pointPointName`.