

TikZ

Manual for Version 3.1.5b

```

\begin{tikzpicture}
  \coordinate (front) at (0,0);
  \coordinate (horizon) at (0,.31\paperheight);
  \coordinate (bottom) at (0,-.6\paperheight);
  \coordinate (sky) at (0,.57\paperheight);
  \coordinate (left) at (-.51\paperwidth,0);
  \coordinate (right) at (.51\paperwidth,0);

  \shade [bottom color=white,
    top color=blue!30!black!50]
      ([yshift=-5mm]horizon -| left)
  rectangle (sky -| right);

  \shade [bottom color=black!70!green!25,
    top color=black!70!green!10]
  (front -| left) -- (horizon -| left)
  decorate [decoration=random steps] {
    -- (horizon -| right)
    -- (front -| right) -- cycle;

  \shade [top color=black!70!green!25,
    bottom color=black!25]
      ([yshift=-5mm-1pt]front -| left)
  rectangle ([yshift=1pt]front -| right);

  \fill [black!25]
      (bottom -| left)
  rectangle ([yshift=-5mm]front -| right);

\def\nodesshadowed[#1]{%
  \node[scale=2,above,#1]{%
    \global\setbox\mybox=\hbox{\#2}%
    \copy\mybox};%
  \node[scale=2,above,#1,yscale=-1,
    scope fading=south,opacity=0.4]{\box\mybox};%
}

```

```

\nodeshadowed [at={(-5,8 )},yslant=0.05]
  {\Huge Ti\textcolor{orange}{\emph{k}}\textcolor{black}{Z}};
\nodeshadowed [at={( 0,8.3)}]
  {\huge \textcolor{green!50!black!50}{\&}\textcolor{black}{));
\nodeshadowed [at={( 5,8 )},yslant=-0.05]
  {\Huge \textsc{PGF}\textcolor{black}{)};
\nodeshadowed [at={( 0,5 )}]
  {Manual for Version \pgfversion};

\foreach \where in {-9cm,9cm} {
  \nodeshadowed [at={(\where,5cm)}] {\tikz
    \draw [green!20!black, rotate=90,
      l-system={rule set={F -> FF-[-F+F]+[+F-F] },
      axiom=F, order=4,step=2pt,
      randomize step percent=50, angle=30,
      randomize angle percent=5}] l-system; }

\foreach \i in {0.5,0.6,...,2}
  \fill
    [white,opacity=\i/2,
     decoration=Koch snowflake,
     shift=(horizon),shift={(rand*11,rnd*7)},
     scale=\i,double copy shadow={
       opacity=0.2,shadow xshift=0pt,
       shadow yshift=3*\i pt,fill=white,draw=none}]
   decorate {
    decorate {
      decorate {
        (0,0)- ++(60:1) -- +(-60:1) -- cycle
      } } };

\node (left text) ...
\node (right text) ...

\fill [decorate,decoration={footprints,foot of=gnome},
  opacity=.5,brown] (rand*8,-rnd*10)
  to [out=rand*180,in=rand*180] (rand*8,-rnd*10);
\end{tikzpicture}

```

Für meinen Vater, damit er noch viele schöne TEX-Graphiken erschaffen kann.

Till

Copyright 2007 to 2013 by Till Tantau

Permission is granted to copy, distribute and/or modify *the documentation* under the terms of the GNU Free Documentation License, Version 1.2 or any later version published by the Free Software Foundation; with no Invariant Sections, no Front-Cover Texts, and no Back-Cover Texts. A copy of the license is included in the section entitled GNU Free Documentation License.

Permission is granted to copy, distribute and/or modify *the code of the package* under the terms of the GNU Public License, Version 2 or any later version published by the Free Software Foundation. A copy of the license is included in the section entitled GNU Public License.

Permission is also granted to distribute and/or modify *both the documentation and the code* under the conditions of the LaTeX Project Public License, either version 1.3 of this license or (at your option) any later version. A copy of the license is included in the section entitled LaTeX Project Public License.

The TikZ and PGF Packages

Manual for version 3.1.5b

<https://github.com/pgf-tikz/pgf>

Till Tantau*

Institut für Theoretische Informatik
Universität zu Lübeck

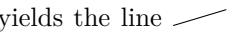
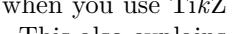
June 25, 2020

Contents

*Editor of this documentation. Parts of this documentation have been written by other authors as indicated in these parts or chapters and in Section 1.5.

1 Introduction

Welcome to the documentation of TikZ and the underlying PGF system. What began as a small L^AT_EX style for creating the graphics in my (Till Tantau's) PhD thesis directly with pdfL^AT_EX has now grown to become a full-blown graphics language with a manual of over a thousand pages. The wealth of options offered by TikZ is often daunting to beginners; but fortunately this documentation comes with a number slowly-paced tutorials that will teach you almost all you should know about TikZ without your having to read the rest.

I wish to start with the questions “What is TikZ?” Basically, it just defines a number of T_EX commands that draw graphics. For example, the code `\tikz \draw (0pt,0pt) --(20pt,6pt);` yields the line  and the code `\tikz \fill[orange] (1ex,1ex) circle (1ex);` yields . In a sense, when you use TikZ you “program” your graphics, just as you “program” your document when you use T_EX. This also explains the name: TikZ is a recursive acronym in the tradition of “GNU's Not Unix” and means “TikZ ist *kein* Zeichenprogramm”, which translates to “TikZ is not a drawing program”, cautioning the reader as to what to expect. With TikZ you get all the advantages of the “T_EX-approach to typesetting” for your graphics: quick creation of simple graphics, precise positioning, the use of macros, often superior typography. You also inherit all the disadvantages: steep learning curve, no WYSIWYG, small changes require a long recompilation time, and the code does not really “show” how things will look like.

Now that we know what TikZ is, what about “PGF”? As mentioned earlier, TikZ started out as a project to implement T_EX graphics macros that can be used both with pdfL^AT_EX and also with the classical (PostScript-based) L^AT_EX. In other words, I wanted to implement a “portable graphics format” for T_EX – hence the name PGF. These early macros are still around and they form the “basic layer” of the system described in this manual, but most of the interaction an author has these days is with TikZ – which has become a whole language of its own.

1.1 The Layers Below TikZ

It turns out that there are actually *two* layers below TikZ:

System layer: This layer provides a complete abstraction of what is going on “in the driver”. The driver is a program like dvips or dvipdfm that takes a .dvi file as input and generates a .ps or a .pdf file. (The pdftex program also counts as a driver, even though it does not take a .dvi file as input. Never mind.) Each driver has its own syntax for the generation of graphics, causing headaches to everyone who wants to create graphics in a portable way. PGF's system layer “abstracts away” these differences. For example, the system command `\pgf{sys@lineto}{10pt}{10pt}` extends the current path to the coordinate (10pt,10pt) of the current `{pgfpicture}`. Depending on whether dvips, dvipdfm, or pdftex is used to process the document, the system command will be converted to different `\special` commands. The system layer is as “minimalistic” as possible since each additional command makes it more work to port PGF to a new driver.

As a user, you will not use the system layer directly.

Basic layer: The basic layer provides a set of basic commands that allow you to produce complex graphics in a much easier manner than by using the system layer directly. For example, the system layer provides no commands for creating circles since circles can be composed from the more basic Bézier curves (well, almost). However, as a user you will want to have a simple command to create circles (at least I do) instead of having to write down half a page of Bézier curve support coordinates. Thus, the basic layer provides a command `\pgfpathcircle` that generates the necessary curve coordinates for you.

The basic layer consists of a *core*, which consists of several interdependent packages that can only be loaded *en bloc*, and additional *modules* that extend the core by more special-purpose commands like node management or a plotting interface. For instance, the BEAMER package uses only the core and not, say, the `shapes` modules.

In theory, TikZ itself is just one of several possible “frontends”, which are sets of commands or a special syntax that makes using the basic layer easier. A problem with directly using the basic layer is that code written for this layer is often too “verbose”. For example, to draw a simple triangle, you may need as many as five commands when using the basic layer: One for beginning a path at the first corner of the triangle, one for extending the path to the second corner, one for going to the third, one for closing the path, and one for actually painting the triangle (as opposed to filling it). With the TikZ frontend all this boils down to a single simple METAFONT-like command:

```
\draw (0,0) -- (1,0) -- (1,1) -- cycle;
```

In practice, *TikZ* is the only “serious” frontend for PGF. It gives you access to all features of PGF, but it is intended to be easy to use. The syntax is a mixture of METAFONT and PSTricks and some ideas of myself. There are other frontends besides *TikZ*, but they are intended more as “technology studies” and less as serious alternatives to *TikZ*. In particular, the *pgfpict2e* frontend reimplements the standard L^AT_EX {picture} environment and commands like \line or \vector using the PGF basic layer. This layer is not really “necessary” since the *pict2e.sty* package does at least as good a job at reimplementing the {picture} environment. Rather, the idea behind this package is to have a simple demonstration of how a frontend can be implemented.

Since most users will only use *TikZ* and almost no one will use the system layer directly, this manual is mainly about *TikZ* in the first parts; the basic layer and the system layer are explained at the end.

1.2 Comparison with Other Graphics Packages

TikZ is not the only graphics package for T_EX. In the following, I try to give a reasonably fair comparison of *TikZ* and other packages.

1. The standard L^AT_EX {picture} environment allows you to create simple graphics, but little more. This is certainly not due to a lack of knowledge or imagination on the part of L^AT_EX’s designer(s). Rather, this is the price paid for the {picture} environment’s portability: It works together with all backend drivers.
2. The *pstricks* package is certainly powerful enough to create any conceivable kind of graphic, but it is not really portable. Most importantly, it does not work with *pdftex* nor with any other driver that produces anything but PostScript code.

Compared to *TikZ*, *pstricks* has a similar support base. There are many nice extra packages for special purpose situations that have been contributed by users over the last decade. The *TikZ* syntax is more consistent than the *pstricks* syntax as *TikZ* was developed “in a more centralized manner” and also “with the shortcomings on *pstricks* in mind”.

3. The *xypic* package is an older package for creating graphics. However, it is more difficult to use and to learn because the syntax and the documentation are a bit cryptic.
4. The *drtex* package is a small graphic package for creating a graphics. Compared to the other package, including *TikZ*, it is very small, which may or may not be an advantage.
5. The *metapost* program is a powerful alternative to *TikZ*. It used to be an external program, which entailed a bunch of problems, but in L^AT_EX it is now built in. An obstacle with *metapost* is the inclusion of labels. This is *much* easier to achieve using PGF.
6. The *xfig* program is an important alternative to *TikZ* for users who do not wish to “program” their graphics as is necessary with *TikZ* and the other packages above. There is a conversion program that will convert *xfig* graphics to *TikZ*.

1.3 Utility Packages

The PGF package comes along with a number of utility package that are not really about creating graphics and which can be used independently of PGF. However, they are bundled with PGF, partly out of convenience, partly because their functionality is closely intertwined with PGF. These utility packages are:

1. The *pgfkeys* package defines a powerful key management facility. It can be used completely independently of PGF.
2. The *pgffor* package defines a useful \foreach statement.
3. The *pgfcalendar* package defines macros for creating calendars. Typically, these calendars will be rendered using PGF’s graphic engine, but you can use *pgfcalendar* also typeset calendars using normal text. The package also defines commands for “working” with dates.

- The `pgfpages` package is used to assemble several pages into a single page. It provides commands for assembling several “virtual pages” into a single “physical page”. The idea is that whenever `TeX` has a page ready for “shipout”, `pgfpages` interrupts this shipout and instead stores the page to be shipped out in a special box. When enough “virtual pages” have been accumulated in this way, they are scaled down and arranged on a “physical page”, which then *really* shipped out. This mechanism allows you to create “two page on one page” versions of a document directly inside `LATeX` without the use of any external programs. However, `pgfpages` can do quite a lot more than that. You can use it to put logos and watermark on pages, print up to 16 pages on one page, add borders to pages, and more.

1.4 How to Read This Manual

This manual describes both the design of `TikZ` and its usage. The organization is very roughly according to “user-friendliness”. The commands and subpackages that are easiest and most frequently used are described first, more low-level and esoteric features are discussed later.

If you have not yet installed `TikZ`, please read the installation first. Second, it might be a good idea to read the tutorial. Finally, you might wish to skim through the description of `TikZ`. Typically, you will not need to read the sections on the basic layer. You will only need to read the part on the system layer if you intend to write your own frontend or if you wish to port PGF to a new driver.

The “public” commands and environments provided by the system are described throughout the text. In each such description, the described command, environment or option is printed in red. Text shown in green is optional and can be left out.

1.5 Authors and Acknowledgements

The bulk of the PGF system and its documentation was written by Till Tantau. A further member of the main team is Mark Wibrow, who is responsible, for example, for the PGF mathematical engine, many shapes, the decoration engine, and matrices. The third member is Christian Feuersänger who contributed the floating point library, image externalization, extended key processing, and automatic hyperlinks in the manual.

Furthermore, occasional contributions have been made by Christophe Jorssen, Jin-Hwan Cho, Olivier Binda, Matthias Schulz, Renée Ahrens, Stephan Schuster, and Thomas Neumann.

Additionally, numerous people have contributed to the PGF system by writing emails, spotting bugs, or sending libraries and patches. Many thanks to all these people, who are too numerous to name them all!

1.6 Getting Help

When you need help with PGF and `TikZ`, please do the following:

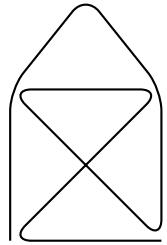
- Read the manual, at least the part that has to do with your problem.
- If that does not solve the problem, try having a look at the GitHub development page for PGF and `TikZ` (see the title of this document). Perhaps someone has already reported a similar problem and someone has found a solution.
- On the website you will find numerous forums for getting help. There, you can write to help forums, file bug reports, join mailing lists, and so on.
- Before you file a bug report, especially a bug report concerning the installation, make sure that this is really a bug. In particular, have a look at the `.log` file that results when you `TeX` your files. This `.log` file should show that all the right files are loaded from the right directories. Nearly all installation problems can be resolved by looking at the `.log` file.
- As a last resort* you can try to email me (Till Tantau) or, if the problem concerns the mathematical engine, Mark Wibrow. I do not mind getting emails, I simply get way too many of them. Because of this, I cannot guarantee that your emails will be answered in a timely fashion or even at all. Your chances that your problem will be fixed are somewhat higher if you mail to the PGF mailing list (naturally, I read this list and answer questions when I have the time).

Part I

Tutorials and Guidelines

by Till Tantau

To help you get started with TikZ, instead of a long installation and configuration section, this manual starts with tutorials. They explain all the basic and some of the more advanced features of the system, without going into all the details. This part also contains some guidelines on how you should proceed when creating graphics using TikZ.



```
\tikz \draw[thick,rounded corners=8pt]
  (0,0) -- (0,2) -- (1,3.25) -- (2,2) -- (2,0) -- (0,2) -- (2,2) -- (0,0) -- (2,0);
```

2 Tutorial: A Picture for Karl's Students

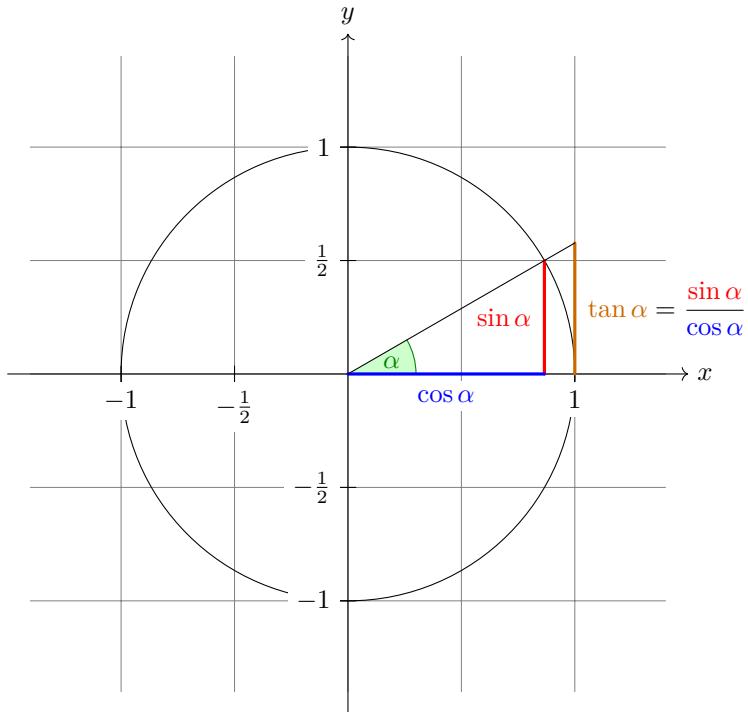
This tutorial is intended for new users of `TikZ`. It does not give an exhaustive account of all the features of `TikZ`, just of those that you are likely to use right away.

Karl is a math and chemistry high-school teacher. He used to create the graphics in his worksheets and exams using `LATeX`'s `{picture}` environment. While the results were acceptable, creating the graphics often turned out to be a lengthy process. Also, there tended to be problems with lines having slightly wrong angles and circles also seemed to be hard to get right. Naturally, his students could not care less whether the lines had the exact right angles and they find Karl's exams too difficult no matter how nicely they were drawn. But Karl was never entirely satisfied with the result.

Karl's son, who was even less satisfied with the results (he did not have to take the exams, after all), told Karl that he might wish to try out a new package for creating graphics. A bit confusingly, this package seems to have two names: First, Karl had to download and install a package called PGF. Then it turns out that inside this package there is another package called `TikZ`, which is supposed to stand for "TikZ ist *kein* Zeichenprogramm". Karl finds this all a bit strange and `TikZ` seems to indicate that the package does not do what he needs. However, having used GNU software for quite some time and "GNU not being Unix", there seems to be hope yet. His son assures him that `TikZ`'s name is intended to warn people that `TikZ` is not a program that you can use to draw graphics with your mouse or tablet. Rather, it is more like a "graphics language".

2.1 Problem Statement

Karl wants to put a graphic on the next worksheet for his students. He is currently teaching his students about sine and cosine. What he would like to have is something that looks like this (ideally):



The angle α is 30° in the example ($\pi/6$ in radians). The sine of α , which is the height of the red line, is

$$\sin \alpha = 1/2.$$

By the Theorem of Pythagoras we have $\cos^2 \alpha + \sin^2 \alpha = 1$. Thus the length of the blue line, which is the cosine of α , must be

$$\cos \alpha = \sqrt{1 - 1/4} = \frac{1}{2}\sqrt{3}.$$

This shows that $\tan \alpha$, which is the height of the orange line, is

$$\tan \alpha = \frac{\sin \alpha}{\cos \alpha} = 1/\sqrt{3}.$$

2.2 Setting up the Environment

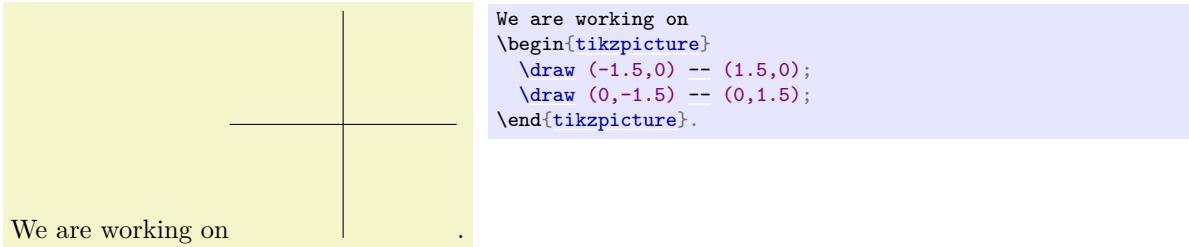
In `TikZ`, to draw a picture, at the start of the picture you need to tell `TeX` or `LATeX` that you want to start a picture. In `LATeX` this is done using the environment `{tikzpicture}`, in plain `TeX` you just use `\tikzpicture` to start the picture and `\endtikzpicture` to end it.

2.2.1 Setting up the Environment in `LATeX`

Karl, being a `LATeX` user, thus sets up his file as follows:

```
\documentclass{article} % say
\usepackage{tikz}
\begin{document}
We are working on
\begin{tikzpicture}
  \draw (-1.5,0) -- (1.5,0);
  \draw (0,-1.5) -- (0,1.5);
\end{tikzpicture}.
\end{document}
```

When executed, that is, run via `pdflatex` or via `latex` followed by `dvips`, the resulting will contain something that looks like this:



Admittedly, not quite the whole picture, yet, but we do have the axes established. Well, not quite, but we have the lines that make up the axes drawn. Karl suddenly has a sinking feeling that the picture is still some way off.

Let's have a more detailed look at the code. First, the package `tikz` is loaded. This package is a so-called "frontend" to the basic PGF system. The basic layer, which is also described in this manual, is somewhat more, well, basic and thus harder to use. The frontend makes things easier by providing a simpler syntax.

Inside the environment there are two `\draw` commands. They mean: "The path, which is specified following the command up to the semicolon, should be drawn." The first path is specified as `(-1.5,0) --(0,1.5)`, which means "a straight line from the point at position $(-1.5, 0)$ to the point at position $(0, 1.5)$ ". Here, the positions are specified within a special coordinate system in which, initially, one unit is 1cm.

Karl is quite pleased to note that the environment automatically reserves enough space to encompass the picture.

2.2.2 Setting up the Environment in Plain TeX

Karl's wife Gerda, who also happens to be a math teacher, is not a L^AT_EX user, but uses plain TeX since she prefers to do things "the old way". She can also use TikZ. Instead of `\usepackage{tikz}` she has to write `\input tikz.tex` and instead of `\begin{tikzpicture}` she writes `\tikzpicture` and instead of `\end{tikzpicture}` she writes `\endtikzpicture`.

Thus, she would use:

```
%% Plain TeX file
\input tikz.tex
\baselineskip=12pt
\hsize=6.3truein
\vsize=8.7truein
We are working on
\tikzpicture
  \draw (-1.5,0) -- (1.5,0);
  \draw (0,-1.5) -- (0,1.5);
\endtikzpicture.
\bye
```

Gerda can typeset this file using either `pdftex` or `tex` together with `dvips`. TikZ will automatically discern which driver she is using. If she wishes to use `dvipdfm` together with `tex`, she either needs to modify the file `pgf.cfg` or can write `\def\pgfsysdriver{pgfsys-dvipdfm.def}` somewhere *before* she inputs `tikz.tex` or `pgf.tex`.

2.2.3 Setting up the Environment in ConTeXt

Karl's uncle Hans uses ConTeXt. Like Gerda, Hans can also use TikZ. Instead of `\usepackage{tikz}` he says `\usemodule[tikz]`. Instead of `\begin{tikzpicture}` he writes `\starttikzpicture` and instead of

```
\end{tikzpicture} he writes \stoptikzpicture.
```

His version of the example looks like this:

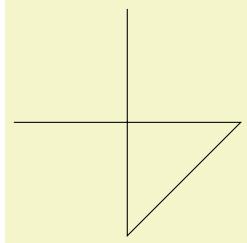
```
%% ConTeXt file
\usemodule[tikz]

\starttext
We are working on
\starttikzpicture
\draw (-1.5,0) -- (1.5,0);
\draw (0,-1.5) -- (0,1.5);
\stoptikzpicture.
\stoptext
```

Hans will now typeset this file in the usual way using `texexec` or `context`.

2.3 Straight Path Construction

The basic building block of all pictures in TikZ is the path. A *path* is a series of straight lines and curves that are connected (that is not the whole picture, but let us ignore the complications for the moment). You start a path by specifying the coordinates of the start position as a point in round brackets, as in $(0,0)$. This is followed by a series of “path extension operations”. The simplest is `--`, which we used already. It must be followed by another coordinate and it extends the path in a straight line to this new position. For example, if we were to turn the two paths of the axes into one path, the following would result:



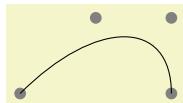
```
\tikz \draw (-1.5,0) -- (1.5,0) -- (0,-1.5) -- (0,1.5);
```

Karl is a bit confused by the fact that there is no `{tikzpicture}` environment, here. Instead, the little command `\tikz` is used. This command either takes one argument (starting with an opening brace as in `\tikz{\draw (0,0) --(1.5,0)}`, which yields _____) or collects everything up to the next semicolon and puts it inside a `{tikzpicture}` environment. As a rule of thumb, all TikZ graphic drawing commands must occur as an argument of `\tikz` or inside a `{tikzpicture}` environment. Fortunately, the command `\draw` will only be defined inside this environment, so there is little chance that you will accidentally do something wrong here.

2.4 Curved Path Construction

The next thing Karl wants to do is to draw the circle. For this, straight lines obviously will not do. Instead, we need some way to draw curves. For this, TikZ provides a special syntax. One or two “control points” are needed. The math behind them is not quite trivial, but here is the basic idea: Suppose you are at point x and the first control point is y . Then the curve will start “going in the direction of y at x ”, that is, the tangent of the curve at x will point toward y . Next, suppose the curve should end at z and the second support point is w . Then the curve will, indeed, end at z and the tangent of the curve at point z will go through w .

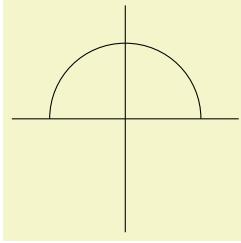
Here is an example (the control points have been added for clarity):



```
\begin{tikzpicture}
\filldraw [gray] (0,0) circle [radius=2pt]
(1,1) circle [radius=2pt]
(2,1) circle [radius=2pt]
(2,0) circle [radius=2pt];
\draw (0,0) .. controls (1,1) and (2,1) .. (2,0);
\end{tikzpicture}
```

The general syntax for extending a path in a “curved” way is `... controls <first control point> and <second control point> ... <end point>`. You can leave out the `and <second control point>`, which causes the first one to be used twice.

So, Karl can now add the first half circle to the picture:



```
\begin{tikzpicture}
  \draw (-1.5,0) -- (1.5,0);
  \draw (0,-1.5) -- (0,1.5);
  \draw (-1,0) .. controls (-1,0.555) and (-0.555,1) .. (0,1)
    .. controls (0.555,1) and (1,0.555) .. (1,0);
\end{tikzpicture}
```

Karl is happy with the result, but finds specifying circles in this way to be extremely awkward. Fortunately, there is a much simpler way.

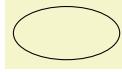
2.5 Circle Path Construction

In order to draw a circle, the path construction operation `circle` can be used. This operation is followed by a radius in brackets as in the following example: (Note that the previous position is used as the *center* of the circle.)



```
\tikz \draw (0,0) circle [radius=10pt];
```

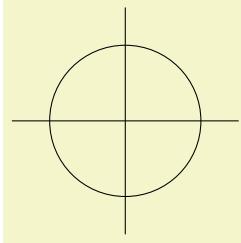
You can also append an ellipse to the path using the `ellipse` operation. Instead of a single radius you can specify two of them:



```
\tikz \draw (0,0) ellipse [x radius=20pt, y radius=10pt];
```

To draw an ellipse whose axes are not horizontal and vertical, but point in an arbitrary direction (a “turned ellipse” like \textcirclearrowright) you can use transformations, which are explained later. The code for the little ellipse is `\tikz \draw[rotate=30] (0,0) ellipse [x radius=6pt, y radius=3pt];`, by the way.

So, returning to Karl’s problem, he can write `\draw (0,0) circle [radius=1cm];` to draw the circle:

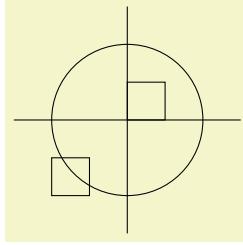


```
\begin{tikzpicture}
  \draw (-1.5,0) -- (1.5,0);
  \draw (0,-1.5) -- (0,1.5);
  \draw (0,0) circle [radius=1cm];
\end{tikzpicture}
```

At this point, Karl is a bit alarmed that the circle is so small when he wants the final picture to be much bigger. He is pleased to learn that TikZ has powerful transformation options and scaling everything by a factor of three is very easy. But let us leave the size as it is for the moment to save some space.

2.6 Rectangle Path Construction

The next things we would like to have is the grid in the background. There are several ways to produce it. For example, one might draw lots of rectangles. Since rectangles are so common, there is a special syntax for them: To add a rectangle to the current path, use the `rectangle` path construction operation. This operation should be followed by another coordinate and will append a rectangle to the path such that the previous coordinate and the next coordinates are corners of the rectangle. So, let us add two rectangles to the picture:



```
\begin{tikzpicture}
\draw (-1.5,0) -- (1.5,0);
\draw (0,-1.5) -- (0,1.5);
\draw (0,0) circle [radius=1cm];
\draw (0,0) rectangle (0.5,0.5);
\draw (-0.5,-0.5) rectangle (-1,-1);
\end{tikzpicture}
```

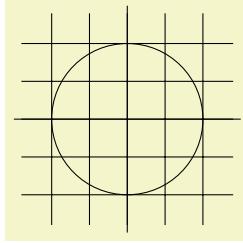
While this may be nice in other situations, this is not really leading anywhere with Karl's problem: First, we would need an awful lot of these rectangles and then there is the border that is not "closed".

So, Karl is about to resort to simply drawing four vertical and four horizontal lines using the nice `\draw` command, when he learns that there is a `grid` path construction operation.

2.7 Grid Path Construction

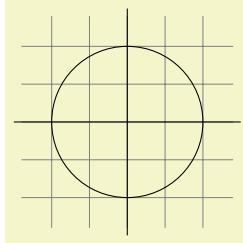
The `grid` path operation adds a grid to the current path. It will add lines making up a grid that fills the rectangle whose one corner is the current point and whose other corner is the point following the `grid` operation. For example, the code `\tikz \draw[step=2pt] (0,0) grid (10pt,10pt);` produces . Note how the optional argument for `\draw` can be used to specify a grid width (there are also `xstep` and `ystep` to define the steppings independently). As Karl will learn soon, there are *lots* of things that can be influenced using such options.

For Karl, the following code could be used:



```
\begin{tikzpicture}
\draw (-1.5,0) -- (1.5,0);
\draw (0,-1.5) -- (0,1.5);
\draw (0,0) circle [radius=1cm];
\draw[step=.5cm] (-1.4,-1.4) grid (1.4,1.4);
\end{tikzpicture}
```

Having another look at the desired picture, Karl notices that it would be nice for the grid to be more subdued. (His son told him that grids tend to be distracting if they are not subdued.) To subdue the grid, Karl adds two more options to the `\draw` command that draws the grid. First, he uses the color `gray` for the grid lines. Second, he reduces the line width to `very thin`. Finally, he swaps the ordering of the commands so that the grid is drawn first and everything else on top.



```
\begin{tikzpicture}
\draw[step=.5cm,gray,very thin] (-1.4,-1.4) grid (1.4,1.4);
\draw (-1.5,0) -- (1.5,0);
\draw (0,-1.5) -- (0,1.5);
\draw (0,0) circle [radius=1cm];
\end{tikzpicture}
```

2.8 Adding a Touch of Style

Instead of the options `gray`, `very thin` Karl could also have said `help lines`. *Styles* are predefined sets of options that can be used to organize how a graphic is drawn. By saying `help lines` you say "use the style that I (or someone else) has set for drawing help lines". If Karl decides, at some later point, that grids should be drawn, say, using the color `blue!50` instead of `gray`, he could provide the following option somewhere:

```
help lines/.style={color=blue!50,very thin}
```

The effect of this "style setter" is that in the current scope or environment the `help lines` option has the same effect as `color=blue!50,very thin`.

Using styles makes your graphics code more flexible. You can change the way things look easily in a consistent manner. Normally, styles are defined at the beginning of a picture. However, you may sometimes wish to define a style globally, so that all pictures of your document can use this style. Then you can easily change the way all graphics look by changing this one style. In this situation you can use the `\tikzset` command at the beginning of the document as in

```
\tikzset{help lines/.style=very thin}
```

To build a hierarchy of styles you can have one style use another. So in order to define a style `Karl's grid` that is based on the `grid` style Karl could say

```
\tikzset{Karl's grid/.style={help lines,color=blue!50}}
...
\draw[Karl's grid] (0,0) grid (5,5);
```

Styles are made even more powerful by parametrization. This means that, like other options, styles can also be used with a parameter. For instance, Karl could parameterize his grid so that, by default, it is blue, but he could also use another color.

```
\begin{tikzpicture}
[Karl's grid/.style ={help lines,color=#1!50},
 Karl's grid/.default=blue]

\draw[Karl's grid] (0,0) grid (1.5,2);
\draw[Karl's grid=red] (2,0) grid (3.5,2);
\end{tikzpicture}
```

In this example, the definition of the style `Karl's grid` is given as an optional argument to the `{tikzpicture}` environment. Additional styles for other elements would follow after a comma. With many styles in effect, the optional argument of the environment may easily happen to be longer than the actual contents.

2.9 Drawing Options

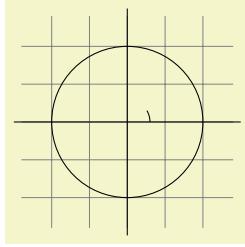
Karl wonders what other options there are that influence how a path is drawn. He saw already that the `color=(color)` option can be used to set the line's color. The option `draw=(color)` does nearly the same, only it sets the color for the lines only and a different color can be used for filling (Karl will need this when he fills the arc for the angle).

He saw that the style `very thin` yields very thin lines. Karl is not really surprised by this and neither is he surprised to learn that `thin` yields thin lines, `thick` yields thick lines, `very thick` yields very thick lines, `ultra thick` yields really, really thick lines and `ultra thin` yields lines that are so thin that low-resolution printers and displays will have trouble showing them. He wonders what gives lines of “normal” thickness. It turns out that `thin` is the correct choice, since it gives the same thickness as TeX's `\hrule` command. Nevertheless, Karl would like to know whether there is anything “in the middle” between `thin` and `thick`. There is: `semithick`.

Another useful thing one can do with lines is to dash or dot them. For this, the two styles `dashed` and `dotted` can be used, yielding `----` and `.....`. Both options also exist in a loose and a dense version, called `loosely dashed`, `densely dashed`, `loosely dotted`, and `densely dotted`. If he really, really needs to, Karl can also define much more complex dashing patterns with the `dash pattern` option, but his son insists that dashing is to be used with utmost care and mostly distracts. Karl's son claims that complicated dashing patterns are evil. Karl's students do not care about dashing patterns.

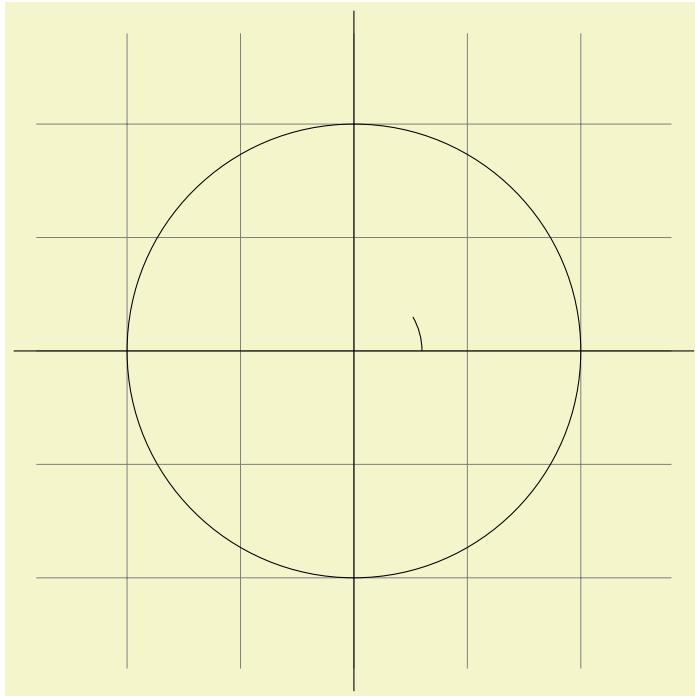
2.10 Arc Path Construction

Our next obstacle is to draw the arc for the angle. For this, the `arc` path construction operation is useful, which draws part of a circle or ellipse. This `arc` operation is followed by options in brackets that specify the arc. An example would be `arc [start angle=10, end angle=80, radius=10pt]`, which means exactly what it says. Karl obviously needs an arc from 0° to 30° . The radius should be something relatively small, perhaps around one third of the circle's radius. When one uses the arc path construction operation, the specified arc will be added with its starting point at the current position. So, we first have to “get there”.



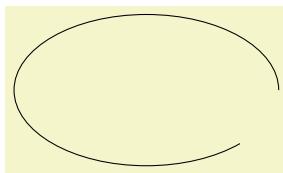
```
\begin{tikzpicture}
\draw[step=.5cm,gray,very thin] (-1.4,-1.4) grid (1.4,1.4);
\draw (-1.5,0) -- (1.5,0);
\draw (0,-1.5) -- (0,1.5);
\draw (0,0) circle [radius=1cm];
\draw (3mm,0mm) arc [start angle=0, end angle=30, radius=3mm];
\end{tikzpicture}
```

Karl thinks this is really a bit small and he cannot continue unless he learns how to do scaling. For this, he can add the `[scale=3]` option. He could add this option to each `\draw` command, but that would be awkward. Instead, he adds it to the whole environment, which causes this option to apply to everything within.



```
\begin{tikzpicture}[scale=3]
\draw[step=.5cm,gray,very thin] (-1.4,-1.4) grid (1.4,1.4);
\draw (-1.5,0) -- (1.5,0);
\draw (0,-1.5) -- (0,1.5);
\draw (0,0) circle [radius=1cm];
\draw (3mm,0mm) arc [start angle=0, end angle=30, radius=3mm];
\end{tikzpicture}
```

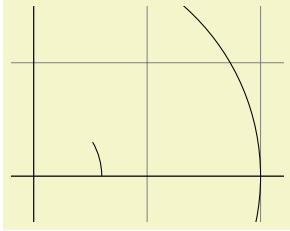
As for circles, you can specify “two” radii in order to get an elliptical arc.



```
\tikz \draw (0,0)
arc [start angle=0, end angle=315,
x radius=1.75cm, y radius=1cm];
```

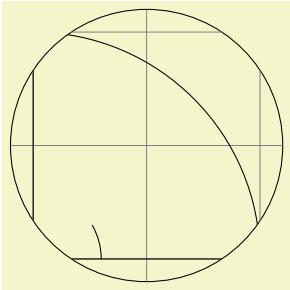
2.11 Clipping a Path

In order to save space in this manual, it would be nice to clip Karl’s graphics a bit so that we can focus on the “interesting” parts. Clipping is pretty easy in TikZ. You can use the `\clip` command to clip all subsequent drawing. It works like `\draw`, only it does not draw anything, but uses the given path to clip everything subsequently.



```
\begin{tikzpicture}[scale=3]
\clip (-0.1,-0.2) rectangle (1.1,0.75);
\draw [step=.5cm,gray,very thin] (-1.4,-1.4) grid (1.4,1.4);
\draw (-1.5,0) -- (1.5,0);
\draw (0,-1.5) -- (0,1.5);
\draw (0,0) circle [radius=1cm];
\draw (3mm,0mm) arc [start angle=0, end angle=30, radius=3mm];
\end{tikzpicture}
```

You can also do both at the same time: Draw *and* clip a path. For this, use the `\draw` command and add the `clip` option. (This is not the whole picture: You can also use the `\clip` command and add the `draw` option. Well, that is also not the whole picture: In reality, `\draw` is just a shorthand for `\path[draw]` and `\clip` is a shorthand for `\path[clip]` and you could also say `\path[draw,clip]`.) Here is an example:



```
\begin{tikzpicture}[scale=3]
\clip[draw] (0.5,0.5) circle (.6cm);
\draw [step=.5cm,gray,very thin] (-1.4,-1.4) grid (1.4,1.4);
\draw (-1.5,0) -- (1.5,0);
\draw (0,-1.5) -- (0,1.5);
\draw (0,0) circle [radius=1cm];
\draw (3mm,0mm) arc [start angle=0, end angle=30, radius=3mm];
\end{tikzpicture}
```

2.12 Parabola and Sine Path Construction

Although Karl does not need them for his picture, he is pleased to learn that there are `parabola` and `sin` and `cos` path operations for adding parabolas and sine and cosine curves to the current path. For the `parabola` operation, the current point will lie on the parabola as well as the point given after the `parabola` operation. Consider the following example:



```
\tikz \draw (0,0) rectangle (1,1) (0,0) parabola (1,1);
```

It is also possible to place the bend somewhere else:



```
\tikz \draw[x=1pt,y=1pt] (0,0) parabola bend (4,16) (6,12);
```

The operations `sin` and `cos` add a sine or cosine curve in the interval $[0, \pi/2]$ such that the previous current point is at the start of the curve and the curve ends at the given end point. Here are two examples:

A sine ↗ curve.

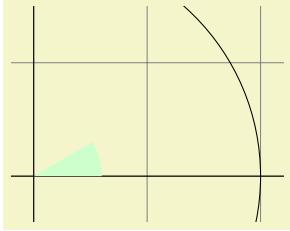
A sine `\tikz \draw[x=1ex,y=1ex] (0,0) sin (1.57,1);` curve.



```
\tikz \draw[x=1.57ex,y=1ex] (0,0) sin (1,1) cos (2,0) sin (3,-1) cos (4,0)
(0,1) cos (1,0) sin (2,-1) cos (3,0) sin (4,1);
```

2.13 Filling and Drawing

Returning to the picture, Karl now wants the angle to be “filled” with a very light green. For this he uses `\fill` instead of `\draw`. Here is what Karl does:



```
\begin{tikzpicture}[scale=3]
\clip (-0.1,-0.2) rectangle (1.1,0.75);
\draw[step=.5cm,gray,very thin] (-1.4,-1.4) grid (1.4,1.4);
\draw (-1.5,0) -- (1.5,0);
\draw (0,-1.5) -- (0,1.5);
\draw (0,0) circle [radius=1cm];
\fill[green!20!white] (0,0) -- (3mm,0mm)
arc [start angle=0, end angle=30, radius=3mm] -- (0,0);
\end{tikzpicture}
```

The color `green!20!white` means 20% green and 80% white mixed together. Such color expression are possible since TikZ uses Uwe Kern's `xcolor` package, see the documentation of that package for details on color expressions.

What would have happened, if Karl had not “closed” the path using `--(0,0)` at the end? In this case, the path is closed automatically, so this could have been omitted. Indeed, it would even have been better to write the following, instead:

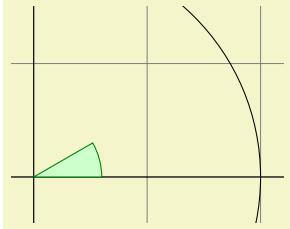
```
\fill[green!20!white] (0,0) -- (3mm,0mm)
arc [start angle=0, end angle=30, radius=3mm] -- cycle;
```

The `--cycle` causes the current path to be closed (actually the current part of the current path) by smoothly joining the first and last point. To appreciate the difference, consider the following example:



```
\begin{tikzpicture}[line width=5pt]
\draw (0,0) -- (1,0) -- (1,1) -- (0,0);
\draw (2,0) -- (3,0) -- (3,1) -- cycle;
\useasboundingbox (0,1.5); % make bounding box higher
\end{tikzpicture}
```

You can also fill and draw a path at the same time using the `\filldraw` command. This will first draw the path, then fill it. This may not seem too useful, but you can specify different colors to be used for filling and for stroking. These are specified as optional arguments like this:



```
\begin{tikzpicture}[scale=3]
\clip (-0.1,-0.2) rectangle (1.1,0.75);
\draw[step=.5cm,gray,very thin] (-1.4,-1.4) grid (1.4,1.4);
\draw (-1.5,0) -- (1.5,0);
\draw (0,-1.5) -- (0,1.5);
\draw (0,0) circle [radius=1cm];
\filldraw[fill=green!20!white, draw=green!50!black] (0,0) -- (3mm,0mm)
arc [start angle=0, end angle=30, radius=3mm] -- cycle;
\end{tikzpicture}
```

2.14 Shading

Karl briefly considers the possibility of making the angle “more fancy” by *shading* it. Instead of filling the area with a uniform color, a smooth transition between different colors is used. For this, `\shade` and `\shadedraw`, for shading and drawing at the same time, can be used:



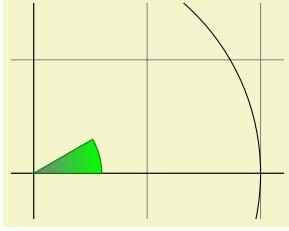
```
\tikz \shade (0,0) rectangle (2,1) (3,0.5) circle (.5cm);
```

The default shading is a smooth transition from gray to white. To specify different colors, you can use options:



```
\begin{tikzpicture}[rounded corners, ultra thick]
\shade[top color=yellow, bottom color=black] (0,0) rectangle +(2,1);
\shade[left color=yellow, right color=black] (3,0) rectangle +(2,1);
\shadedraw[inner color=yellow, outer color=black, draw=yellow] (6,0) rectangle +(2,1);
\shade[ball color=green] (9,.5) circle (.5cm);
\end{tikzpicture}
```

For Karl, the following might be appropriate:



```
\begin{tikzpicture}[scale=3]
\clip (-0.1,-0.2) rectangle (1.1,0.75);
\draw[step=.5cm,gray,very thin] (-1.4,-1.4) grid (1.4,1.4);
\draw (-1.5,0) -- (1.5,0);
\draw (0,-1.5) -- (0,1.5);
\draw (0,0) circle [radius=1cm];
\shadedraw[left color=gray, right color=green, draw=green!50!black]
(0,0) -- (3mm,0mm)
arc [start angle=0, end angle=30, radius=3mm] -- cycle;
\end{tikzpicture}
```

However, he wisely decides that shadings usually only distract without adding anything to the picture.

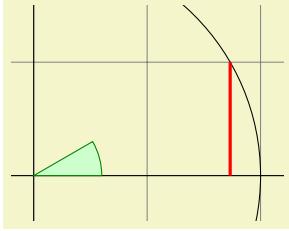
2.15 Specifying Coordinates

Karl now wants to add the sine and cosine lines. He knows already that he can use the `color=` option to set the lines' colors. So, what is the best way to specify the coordinates?

There are different ways of specifying coordinates. The easiest way is to say something like `(10pt, 2cm)`. This means 10pt in x -direction and 2cm in y -directions. Alternatively, you can also leave out the units as in `(1, 2)`, which means “one times the current x -vector plus twice the current y -vector”. These vectors default to 1cm in the x -direction and 1cm in the y -direction, respectively.

In order to specify points in polar coordinates, use the notation `(30 : 1cm)`, which means 1cm in direction 30 degree. This is obviously quite useful to “get to the point $(\cos 30^\circ, \sin 30^\circ)$ on the circle”.

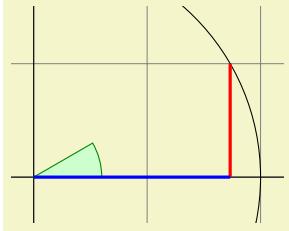
You can add a single + sign in front of a coordinate or two of them as in `+(0cm, 1cm)` or `++(2cm, 0cm)`. Such coordinates are interpreted differently: The first form means “1cm upwards from the previous specified position” and the second means “2cm to the right of the previous specified position, making this the new specified position”. For example, we can draw the sine line as follows:



```
\begin{tikzpicture}[scale=3]
\clip (-0.1,-0.2) rectangle (1.1,0.75);
\draw[step=.5cm,gray,very thin] (-1.4,-1.4) grid (1.4,1.4);
\draw (-1.5,0) -- (1.5,0);
\draw (0,-1.5) -- (0,1.5);
\draw (0,0) circle [radius=1cm];
\filldraw[fill=green!20, draw=green!50!black] (0,0) -- (3mm,0mm)
arc [start angle=0, end angle=30, radius=3mm] -- cycle;
\draw[red, very thick] (30:1cm) -- +(0,-0.5);
\draw[blue, very thick] (30:1cm) ++(0,-0.5) -- (0,0);
\end{tikzpicture}
```

Karl used the fact $\sin 30^\circ = 1/2$. However, he very much doubts that his students know this, so it would be nice to have a way of specifying “the point straight down from $(30:1\text{cm})$ that lies on the x -axis”. This is, indeed, possible using a special syntax: Karl can write `(30:1cm |- 0,0)`. In general, the meaning of `((p) |- (q))` is “the intersection of a vertical line through p and a horizontal line through q ”.

Next, let us draw the cosine line. One way would be to say `(30:1cm |- 0,0) -- (0,0)`. Another way is the following: we “continue” from where the sine ends:

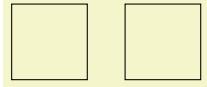


```
\begin{tikzpicture}[scale=3]
\clip (-0.1,-0.2) rectangle (1.1,0.75);
\draw[step=.5cm,gray,very thin] (-1.4,-1.4) grid (1.4,1.4);
\draw (-1.5,0) -- (1.5,0);
\draw (0,-1.5) -- (0,1.5);
\draw (0,0) circle [radius=1cm];
\filldraw[fill=green!20, draw=green!50!black] (0,0) -- (3mm,0mm)
arc [start angle=0, end angle=30, radius=3mm] -- cycle;
\draw[red, very thick] (30:1cm) -- +(0,-0.5);
\draw[blue, very thick] (30:1cm) ++(0,-0.5) -- (0,0);
\end{tikzpicture}
```

Note that there is no `--` between `(30:1cm)` and `++(0,-0.5)`. In detail, this path is interpreted as follows: “First, the `(30:1cm)` tells me to move by pen to $(\cos 30^\circ, 1/2)$. Next, there comes another coordinate

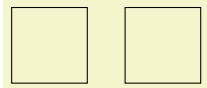
specification, so I move my pen there without drawing anything. This new point is half a unit down from the last position, thus it is at $(\cos 30^\circ, 0)$. Finally, I move the pen to the origin, but this time drawing something (because of the `--`)."

To appreciate the difference between `+` and `++` consider the following example:



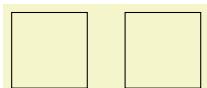
```
\begin{tikzpicture}
\def\rectanglepath{-- ++(1cm,0cm) -- +(0cm,1cm) -- ++(-1cm,0cm) -- cycle}
\draw (0,0) \rectanglepath;
\draw (1.5,0) \rectanglepath;
\end{tikzpicture}
```

By comparison, when using a single `+`, the coordinates are different:



```
\begin{tikzpicture}
\def\rectanglepath{-- +(1cm,0cm) -- +(1cm,1cm) -- +(0cm,1cm) -- cycle}
\draw (0,0) \rectanglepath;
\draw (1.5,0) \rectanglepath;
\end{tikzpicture}
```

Naturally, all of this could have been written more clearly and more economically like this (either with a single or a double `+`):



```
\tikz \draw (0,0) rectangle +(1,1) (1.5,0) rectangle +(1,1);
```

2.16 Intersecting Paths

Karl is left with the line for $\tan \alpha$, which seems difficult to specify using transformations and polar coordinates. The first – and easiest – thing he can do is so simply use the coordinate `(1,{tan(30)})` since TikZ's math engine knows how to compute things like `tan(30)`. Note the added braces since, otherwise, TikZ's parser would think that the first closing parenthesis ends the coordinate (in general, you need to add braces around components of coordinates when these components contain parentheses).

Karl can, however, also use a more elaborate, but also more “geometric” way of computing the length of the orange line: He can specify intersections of paths as coordinates. The line for $\tan \alpha$ starts at `(1,0)` and goes upward to a point that is at the intersection of a line going “up” and a line going from the origin through `(30:1cm)`. Such computations are made available by the `intersections` library.

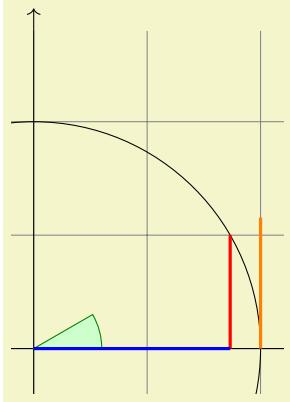
What Karl must do is to create two “invisible” paths that intersect at the position of interest. Creating paths that are not otherwise seen can be done using the `\path` command without any options like `draw` or `fill`. Then, Karl can add the `name` path option to the path for later reference. Once the paths have been constructed, Karl can use the `name intersections` to assign names to the coordinate for later reference.

```
\path [name path=upward line] (1,0) -- (1,1);
\path [name path=sloped line] (0,0) -- (30:1.5cm); % a bit longer, so that there is an intersection
% (add '\usetikzlibrary{intersections}' after loading tikz in the preamble)
\draw [name intersections={of=upward line and sloped line, by=x}]
[very thick,orange] (1,0) -- (x);
```

2.17 Adding Arrow Tips

Karl now wants to add the little arrow tips at the end of the axes. He has noticed that in many plots, even in scientific journals, these arrow tips seem to be missing, presumably because the generating programs cannot produce them. Karl thinks arrow tips belong at the end of axes. His son agrees. His students do not care about arrow tips.

It turns out that adding arrow tips is pretty easy: Karl adds the option `->` to the drawing commands for the axes:

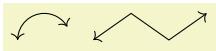


```
\usetikzlibrary {intersections}
\begin{tikzpicture}[scale=3]
\clip (-0.1,-0.2) rectangle (1.1,1.51);
\draw [step=.5cm,gray,very thin] (-1.4,-1.4) grid (1.4,1.4);
\draw[->] (-1.5,0) -- (1.5,0);
\draw[->] (0,-1.5) -- (0,1.5);
\draw (0,0) circle [radius=1cm];
\filldraw[fill=green!20,draw=green!50!black] (0,0) -- (3mm,0mm)
    arc [start angle=0, end angle=30, radius=3mm] -- cycle;
\draw[red,very thick] (30:1cm) -- +(0,-0.5);
\draw[blue,very thick] (30:1cm) ++(0,-0.5) -- (0,0);

\path [name path=upward line] (1,0) -- (1,1);
\path [name path=sloped line] (0,0) -- (30:1.5cm);
\draw [name intersections={of=upward line and sloped line, by=x}]
[very thick,orange] (1,0) -- (x);
\end{tikzpicture}
```

If Karl had used the option `<-` instead of `->`, arrow tips would have been put at the beginning of the path. The option `<->` puts arrow tips at both ends of the path.

There are certain restrictions to the kind of paths to which arrow tips can be added. As a rule of thumb, you can add arrow tips only to a single open “line”. For example, you cannot add tips to, say, a rectangle or a circle. However, you can add arrow tips to curved paths and to paths that have several segments, as in the following examples:



```
\begin{tikzpicture}
\draw [<->] (0,0) arc [start angle=180, end angle=30, radius=10pt];
\draw [<->] (1,0) -- (1.5cm,10pt) -- (2cm,0pt) -- (2.5cm,10pt);
\end{tikzpicture}
```

Karl has a more detailed look at the arrow that TikZ puts at the end. It looks like this when he zooms it: \rightarrow . The shape seems vaguely familiar and, indeed, this is exactly the end of TeX’s standard arrow used in something like $f: A \rightarrow B$.

Karl likes the arrow, especially since it is not “as thick” as the arrows offered by many other packages. However, he expects that, sometimes, he might need to use some other kinds of arrow. To do so, Karl can say `>=<kind of end arrow tip>`, where `<kind of end arrow tip>` is a special arrow tip specification. For example, if Karl says `>=Stealth`, then he tells TikZ that he would like “stealth-fighter-like” arrow tips:



```
\usetikzlibrary {arrows.meta}
\begin{tikzpicture}[>=Stealth]
\draw [->] (0,0) arc [start angle=180, end angle=30, radius=10pt];
\draw [->,very thick] (1,0) -- (1.5cm,10pt) -- (2cm,0pt) -- (2.5cm,10pt);
\end{tikzpicture}
```

Karl wonders whether such a military name for the arrow type is really necessary. He is not really mollified when his son tells him that Microsoft’s PowerPoint uses the same name. He decides to have his students discuss this at some point.

In addition to `Stealth` there are several other predefined kinds of arrow tips Karl can choose from, see Section ???. Furthermore, he can define arrows types himself, if he needs new ones.

2.18 Scoping

Karl saw already that there are numerous graphic options that affect how paths are rendered. Often, he would like to apply certain options to a whole set of graphic commands. For example, Karl might wish to draw three paths using a `thick` pen, but would like everything else to be drawn “normally”.

If Karl wishes to set a certain graphic option for the whole picture, he can simply pass this option to the `\tikz` command or to the `{tikzpicture}` environment (Gerda would pass the options to `\tikzpicture` and Hans passes them to `\starttikzpicture`). However, if Karl wants to apply graphic options to a local group, he put these commands inside a `{scope}` environment (Gerda uses `\scope` and `\endscope`, Hans uses `\startscope` and `\stopscope`). This environment takes graphic options as an optional argument and these options apply to everything inside the scope, but not to anything outside.

Here is an example:



```
\begin{tikzpicture}[ultra thick]
\draw (0,0) -- (0,1);
\begin{scope}[thin]
\draw (1,0) -- (1,1);
\draw (2,0) -- (2,1);
\end{scope}
\draw (3,0) -- (3,1);
\end{tikzpicture}
```

Scoping has another interesting effect: Any changes to the clipping area are local to the scope. Thus, if you say `\clip` somewhere inside a scope, the effect of the `\clip` command ends at the end of the scope. This is useful since there is no other way of “enlarging” the clipping area.

Karl has also already seen that giving options to commands like `\draw` apply only to that command. It turns out that the situation is slightly more complex. First, options to a command like `\draw` are not really options to the command, but they are “path options” and can be given anywhere on the path. So, instead of `\draw[thin] (0,0) --(1,0);` one can also write `\draw (0,0) [thin] --(1,0);` or `\draw (0,0) --(1,0) [thin];`; all of these have the same effect. This might seem strange since in the last case, it would appear that the `thin` should take effect only “after” the line from $(0,0)$ to $(1,0)$ has been drawn. However, most graphic options only apply to the whole path. Indeed, if you say both `thin` and `thick` on the same path, the last option given will “win”.

When reading the above, Karl notices that only “most” graphic options apply to the whole path. Indeed, all transformation options do *not* apply to the whole path, but only to “everything following them on the path”. We will have a more detailed look at this in a moment. Nevertheless, all options given during a path construction apply only to this path.

2.19 Transformations

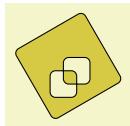
When you specify a coordinate like `(1cm,1cm)`, where is that coordinate placed on the page? To determine the position, TikZ, \TeX , and PDF or PostScript all apply certain transformations to the given coordinate in order to determine the final position on the page.

TikZ provides numerous options that allow you to transform coordinates in TikZ’s private coordinate system. For example, the `xshift` option allows you to shift all subsequent points by a certain amount:

```
\tikz \draw (0,0) -- (0,0.5) [xshift=2pt] (0,0) -- (0,0.5);
```

It is important to note that you can change transformation “in the middle of a path”, a feature that is not supported by PDF or PostScript. The reason is that TikZ keeps track of its own transformation matrix.

Here is a more complicated example:



```
\begin{tikzpicture}[even odd rule,rounded corners=2pt,x=10pt,y=10pt]
\filldraw[fill=yellow!80!black] (0,0) rectangle (1,1)
[xshift=5pt,yshift=5pt] (0,0) rectangle (1,1)
[rotate=30] (-1,-1) rectangle (2,2);
\end{tikzpicture}
```

The most useful transformations are `xshift` and `yshift` for shifting, `shift` for shifting to a given point as in `shift={(1,0)}` or `shift={+(0,0)}` (the braces are necessary so that \TeX does not mistake the comma for separating options), `rotate` for rotating by a certain angle (there is also a `rotate around` for rotating around a given point), `scale` for scaling by a certain factor, `xscale` and `yscale` for scaling only in the x - or y -direction (`xscale=-1` is a flip), and `xslant` and `yslant` for slanting. If these transformation and those that I have not mentioned are not sufficient, the `cm` option allows you to apply an arbitrary transformation matrix. Karl’s students, by the way, do not know what a transformation matrix is.

2.20 Repeating Things: For-Loops

Karl’s next aim is to add little ticks on the axes at positions -1 , $-1/2$, $1/2$, and 1 . For this, it would be nice to use some kind of “loop”, especially since he wishes to do the same thing at each of these positions. There are different packages for doing this. \LaTeX has its own internal command for this, `pstricks` comes along with the powerful `\multido` command. All of these can be used together with TikZ, so if you are familiar with them, feel free to use them. TikZ introduces yet another command, called `\foreach`, which

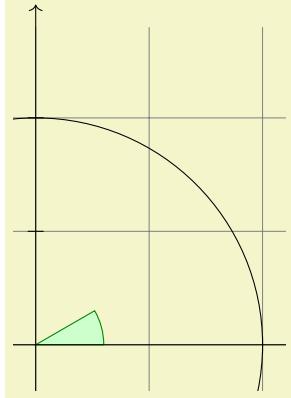
I introduced since I could never remember the syntax of the other packages. `\foreach` is defined in the package `pgffor` and can be used independently of TikZ, but TikZ includes it automatically.

In its basic form, the `\foreach` command is easy to use:

```
x = 1, x = 2, x = 3, \foreach \x in {1,2,3} {$x =\x$, }
```

The general syntax is `\foreach <variable> in {<list of values>} <commands>`. Inside the `<commands>`, the `<variable>` will be assigned to the different values. If the `<commands>` do not start with a brace, everything up to the next semicolon is used as `<commands>`.

For Karl and the ticks on the axes, he could use the following code:



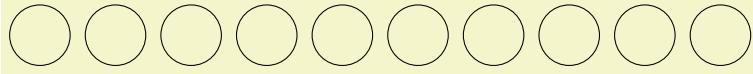
```
\begin{tikzpicture}[scale=3]
\clip (-0.1,-0.2) rectangle (1.1,1.51);
\draw[step=.5cm,gray,very thin] (-1.4,-1.4) grid (1.4,1.4);
\filldraw[fill=green!20,draw=green!50!black] (0,0) -- (3mm,0mm) arc [start angle=0, end angle=30, radius=3mm] -- cycle;
\draw[->] (-1.5,0) -- (1.5,0);
\draw[->] (0,-1.5) -- (0,1.5);
\draw (0,0) circle [radius=1cm];

\foreach \x in {-1cm,-0.5cm,1cm}
\draw (\x,-1pt) -- (\x,1pt);
\foreach \y in {-1cm,-0.5cm,0.5cm,1cm}
\draw (-1pt,\y) -- (1pt,\y);
\end{tikzpicture}
```

As a matter of fact, there are many different ways of creating the ticks. For example, Karl could have put the `\draw ...;` inside curly braces. He could also have used, say,

```
\foreach \x in {-1,-0.5,1}
\draw[xshift=\x cm] (0pt,-1pt) -- (0pt,1pt);
```

Karl is curious what would happen in a more complicated situation where there are, say, 20 ticks. It seems bothersome to explicitly mention all these numbers in the set for `\foreach`. Indeed, it is possible to use `...` inside the `\foreach` statement to iterate over a large number of values (which must, however, be dimensionless real numbers) as in the following example:



```
\tikz \foreach \x in {1,...,10}
\draw (\x,0) circle (0.4cm);
```

If you provide *two* numbers before the `...`, the `\foreach` statement will use their difference for the stepping:

```
[ ] \tikz \foreach \x in {-1,-0.5,...,1}
\draw (\x cm,-1pt) -- (\x cm,1pt);
```

We can also nest loops to create interesting effects:

1,5	2,5	3,5	4,5	5,5	7,5	8,5	9,5	10,5	11,5	12,5
1,4	2,4	3,4	4,4	5,4	7,4	8,4	9,4	10,4	11,4	12,4
1,3	2,3	3,3	4,3	5,3	7,3	8,3	9,3	10,3	11,3	12,3
1,2	2,2	3,2	4,2	5,2	7,2	8,2	9,2	10,2	11,2	12,2
1,1	2,1	3,1	4,1	5,1	7,1	8,1	9,1	10,1	11,1	12,1

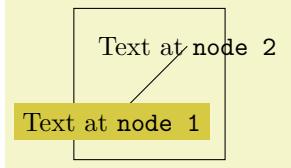
```
\begin{tikzpicture}
\foreach \x in {1,2,\dots,5,7,8,\dots,12}
\foreach \y in {1,\dots,5}
{
  \draw (\x,\y) +(-.5,-.5) rectangle +(.,.);
  \draw (\x,\y) node[\x,\y];
}
\end{tikzpicture}
```

The `\foreach` statement can do even trickier stuff, but the above gives the idea.

2.21 Adding Text

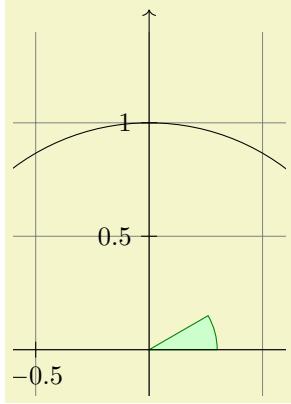
Karl is, by now, quite satisfied with the picture. However, the most important parts, namely the labels, are still missing!

TikZ offers an easy-to-use and powerful system for adding text and, more generally, complex shapes to a picture at specific positions. The basic idea is the following: When TikZ is constructing a path and encounters the keyword `node` in the middle of a path, it reads a *node specification*. The keyword `node` is typically followed by some options and then some text between curly braces. This text is put inside a normal TeX box (if the node specification directly follows a coordinate, which is usually the case, TikZ is able to perform some magic so that it is even possible to use verbatim text inside the boxes) and then placed at the current position, that is, at the last specified position (possibly shifted a bit, according to the given options). However, all nodes are drawn only after the path has been completely drawn/filled/shaded/clipped/whatever.



```
\begin{tikzpicture}
\draw (0,0) rectangle (2,2);
\draw (0.5,0.5) node [fill=yellow!80!black]
  {Text at \verb!node 1!}
  -- (1.5,1.5) node {Text at \verb!node 2!};
\end{tikzpicture}
```

Obviously, Karl would not only like to place nodes *on* the last specified position, but also to the left or the right of these positions. For this, every node object that you put in your picture is equipped with several *anchors*. For example, the `north` anchor is in the middle at the upper end of the shape, the `south` anchor is at the bottom and the `north east` anchor is in the upper right corner. When you give the option `anchor=north`, the text will be placed such that this northern anchor will lie on the current position and the text is, thus, below the current position. Karl uses this to draw the ticks as follows:



```
\begin{tikzpicture}[scale=3]
\clip (-0.6,-0.2) rectangle (0.6,1.51);
\draw[step=.5cm,help lines] (-1.4,-1.4) grid (1.4,1.4);
\filldraw[fill=green!20,draw=green!50!black] (0,0) -- (3mm,0mm)
  arc [start angle=0, end angle=30, radius=3mm] -- cycle;
\draw[->] (-1.5,0) -- (1.5,0); \draw[->] (0,-1.5) -- (0,1.5);
\draw (0,0) circle [radius=1cm];

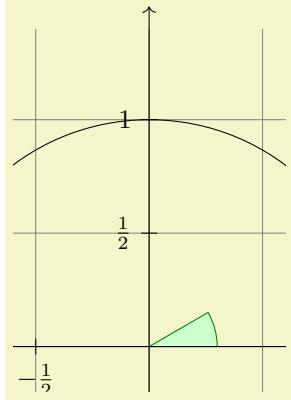
\foreach \x in {-1,-0.5,1}
  \draw (\x cm,1pt) -- (\x cm,-1pt) node[anchor=north] {$\x$};
\foreach \y in {-1,-0.5,0.5,1}
  \draw (1pt,\y cm) -- (-1pt,\y cm) node[anchor=east] {$\y$};
\end{tikzpicture}
```

This is quite nice, already. Using these anchors, Karl can now add most of the other text elements. However, Karl thinks that, though “correct”, it is quite counter-intuitive that in order to place something *below* a given point, he has to use the `north` anchor. For this reason, there is an option called `below`, which does the same as `anchor=north`. Similarly, `above right` does the same as `anchor=south west`. In addition, `below` takes an optional dimension argument. If given, the shape will additionally be shifted downwards by the given amount. So, `below=1pt` can be used to put a text label below some point and, additionally shift it 1pt downwards.

Karl is not quite satisfied with the ticks. He would like to have $1/2$ or $\frac{1}{2}$ shown instead of 0.5, partly to show off the nice capabilities of TeX and TikZ, partly because for positions like $1/3$ or π it is certainly very

much preferable to have the “mathematical” tick there instead of just the “numeric” tick. His students, on the other hand, prefer 0.5 over $1/2$ since they are not too fond of fractions in general.

Karl now faces a problem: For the `\foreach` statement, the position `\x` should still be given as 0.5 since TikZ will not know where `\frac{1}{2}` is supposed to be. On the other hand, the typeset text should really be `\frac{1}{2}`. To solve this problem, `\foreach` offers a special syntax: Instead of having one variable `\x`, Karl can specify two (or even more) variables separated by a slash as in `\x / \xtext`. Then, the elements in the set over which `\foreach` iterates must also be of the form `\langle first\rangle / \langle second\rangle`. In each iteration, `\x` will be set to `\langle first\rangle` and `\xtext` will be set to `\langle second\rangle`. If no `\langle second\rangle` is given, the `\langle first\rangle` will be used again. So, here is the new code for the ticks:



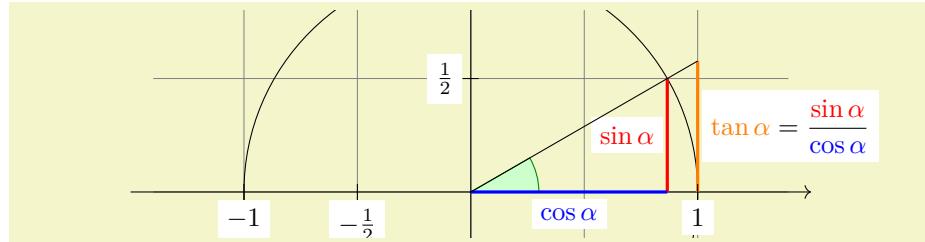
```
\begin{tikzpicture}[scale=3]
\clip (-0.6,-0.2) rectangle (0.6,1.51);
\draw[step=.5cm,help lines] (-1.4,-1.4) grid (1.4,1.4);
\filldraw[fill=green!20,draw=green!50!black] (0,0) -- (3mm,0mm)
arc [start angle=0, end angle=30, radius=3mm] -- cycle;
\draw[->] (-1.5,0) -- (1.5,0); \draw[->] (0,-1.5) -- (0,1.5);
\draw (0,0) circle [radius=1cm];

\foreach \x/\xtext in {-1, -0.5/-\frac{1}{2}, 1}
\draw (\x cm,1pt) -- (\x cm,-1pt) node[anchor=north] {$\xtext$};
\foreach \y/\ytext in {-1, -0.5/-\frac{1}{2}, 0.5/\frac{1}{2}, 1}
\draw (1pt,\y cm) -- (-1pt,\y cm) node[anchor=east] {$\ytext$};

\end{tikzpicture}
```

Karl is quite pleased with the result, but his son points out that this is still not perfectly satisfactory: The grid and the circle interfere with the numbers and decrease their legibility. Karl is not very concerned by this (his students do not even notice), but his son insists that there is an easy solution: Karl can add the `[fill=white` option to fill out the background of the text shape with a white color.

The next thing Karl wants to do is to add the labels like $\sin \alpha$. For this, he would like to place a label “in the middle of the line”. To do so, instead of specifying the label `node {$\sin \alpha$}` directly after one of the endpoints of the line (which would place the label at that endpoint), Karl can give the label directly after the `--`, before the coordinate. By default, this places the label in the middle of the line, but the `pos=` options can be used to modify this. Also, options like `near start` and `near end` can be used to modify this position:



```

\usetikzlibrary {intersections}
\begin{tikzpicture}[scale=3]
  \clip (-2,-0.2) rectangle (2,0.8);
  \draw[step=.5cm,gray,very thin] (-1.4,-1.4) grid (1.4,1.4);
  \filldraw[fill=green!20,draw=green!50!black] (0,0) -- (3mm,0mm) -- cycle;
  \arc [start angle=0, end angle=30, radius=3mm] -- cycle;
  \draw[->] (-1.5,0) -- (1.5,0) coordinate (x axis);
  \draw[->] (0,-1.5) -- (0,1.5) coordinate (y axis);
  \draw (0,0) circle [radius=1cm];

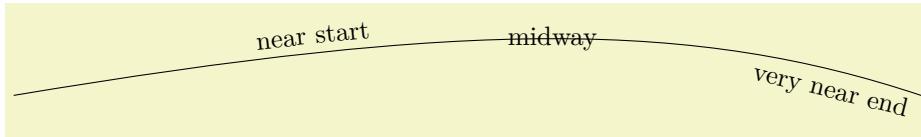
  \draw[very thick,red]
    (30:1cm) -- node[left=1pt,fill=white] {$\sin \alpha$} (30:1cm |- x axis);
  \draw[very thick,blue]
    (30:1cm |- x axis) -- node[below=2pt,fill=white] {$\cos \alpha$} (0,0);
  \path [name path=upward line] (1,0) -- (1,1);
  \path [name path=sloped line] (0,0) -- (30:1.5cm);
  \draw [name intersections={of=upward line and sloped line, by=t}]
    [very thick,orange] (1,0) -- node [right=1pt,fill=white]
    {\displaystyle \tan \alpha \color{black}=
      \frac{\color{red}\sin \alpha}{\color{blue}\cos \alpha}} (t);

  \draw (0,0) -- (t);

  \foreach \x/\xtext in {-1, -0.5/-\frac{1}{2}, 1}
    \draw (\x cm,1pt) -- (\x cm,-1pt) node[anchor=north,fill=white] {$\xtext$};
  \foreach \y/\ytext in {-1, -0.5/-\frac{1}{2}, 0.5/\frac{1}{2}, 1}
    \draw (1pt,\y cm) -- (-1pt,\y cm) node[anchor=east,fill=white] {$\ytext$};
\end{tikzpicture}

```

You can also position labels on curves and, by adding the `sloped` option, have them rotated such that they match the line's slope. Here is an example:



```

\begin{tikzpicture}
  \draw (0,0) .. controls (6,1) and (9,1) .. 
    node[near start,sloped,above] {near start}
    node[midway]
    node[very near end,sloped,below] {very near end} (12,0);
\end{tikzpicture}

```

It remains to draw the explanatory text at the right of the picture. The main difficulty here lies in limiting the width of the text “label”, which is quite long, so that line breaking is used. Fortunately, Karl can use the option `text width=6cm` to get the desired effect. So, here is the full code:

```

\begin{tikzpicture}
[scale=3,line cap=round,
% Styles
axes/.style=,
important line/.style={very thick},
information text/.style={rounded corners,fill=red!10,inner sep=1ex}]

% Colors
\colorlet{anglecolor}{green!50!black}
\colorlet{sincolor}{red}
\colorlet{tancolor}{orange!80!black}
\colorlet{coscolor}{blue}

% The graphic
\draw[help lines,step=0.5cm] (-1.4,-1.4) grid (1.4,1.4);

\draw (0,0) circle [radius=1cm];

\begin{scope}[axes]
\draw[->] (-1.5,0) -- (1.5,0) node[right] {$x$} coordinate(x axis);
\draw[->] (0,-1.5) -- (0,1.5) node[above] {$y$} coordinate(y axis);

\foreach \x/\xtext in {-1, -.5/-\frac{1}{2}, 1}
\draw[xshift=\x cm] (0pt,1pt) -- (0pt,-1pt) node[below,fill=white] {$\xtext$};

\foreach \y/\ytext in {-1, -.5/-\frac{1}{2}, .5/\frac{1}{2}, 1}
\draw[yshift=\y cm] (1pt,0pt) -- (-1pt,0pt) node[left,fill=white] {$\ytext$};
\end{scope}

\filldraw[fill=green!20,draw=anglecolor] (0,0) -- (3mm,0pt)
arc [start angle=0, end angle=30, radius=3mm];
\draw (15:2mm) node[anglecolor] {$\alpha$};

\draw[important line,sincolor]
(30:1cm) -- node[left=1pt,fill=white] {$\sin \alpha$} (30:1cm |- x axis);

\draw[important line,coscolor]
(30:1cm |- x axis) -- node[below=2pt,fill=white] {$\cos \alpha$} (0,0);

\path [name path=upward line] (1,0) -- (1,1);
\path [name path=sloped line] (0,0) -- (30:1.5cm);
\draw [name intersections={of=upward line and sloped line, by=t}]
[very thick,orange] (1,0) -- node [right=1pt,fill=white]
{$\tan \alpha$};
\frac{\sin \alpha}{\cos \alpha} (t);

\draw (0,0) -- (t);

\draw[xshift=1.85cm]
node[right,text width=6cm,information text]
{
The $\angle \alpha$ is $30^\circ$ in the example ($\pi/6$ in radians). The $\sin \alpha$ is
$\frac{1}{2}$.
}
;

\end{tikzpicture}

```

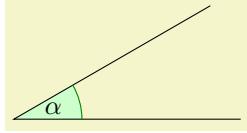
2.22 Pics: The Angle Revisited

Karl expects that the code of certain parts of the picture he created might be so useful that he might wish to reuse them in the future. A natural thing to do is to create TeX macros that store the code he wishes to reuse. However, TikZ offers another way that is integrated directly into its parser: pics!

A “pic” is “not quite a full picture”, hence the short name. The idea is that a pic is simply some code that you can add to a picture at different places using the `pic` command whose syntax is almost identical to the `node` command. The main difference is that instead of specifying some text in curly braces that should be shown, you specify the name of a predefined picture that should be shown.

Defining new pics is easy enough, see Section 18, but right now we just want to use one such predefined pic: the `angle` pic. As the name suggests, it is a small drawing of an angle consisting of a little wedge and an arc together with some text (Karl needs to load the `angles` library and the `quotes` for the following examples). What makes this pic useful is the fact that the size of the wedge will be computed automatically.

The `angle` pic draws an angle between the two lines BA and BC , where A , B , and C are three coordinates. In our case, B is the origin, A is somewhere on the x -axis and C is somewhere on a line at 30° .



```
\usetikzlibrary {angles,quotes}
\begin{tikzpicture}[scale=3]
  \coordinate (A) at (1,0);
  \coordinate (B) at (0,0);
  \coordinate (C) at (30:1cm);

  \draw (A) -- (B) -- (C)
    pic [draw=green!50!black, fill=green!20, angle radius=9mm,
          "\$\alpha\$"] {angle = A--B--C};
\end{tikzpicture}
```

Let us see, what is happening here. First we have specified three *coordinates* using the `\coordinate` command. It allows us to name a specific coordinate in the picture. Then comes something that starts as a normal `\draw`, but then comes the `pic` command. This command gets lots of options and, in curly braces, comes the most important point: We specify that we want to add an `angle` pic and this angle should be between the points we named `A`, `B`, and `C` (we could use other names). Note that the text that we want to be shown in the pic is specified in quotes inside the options of the `pic`, not inside the curly braces.

To learn more about pics, please see Section 18.

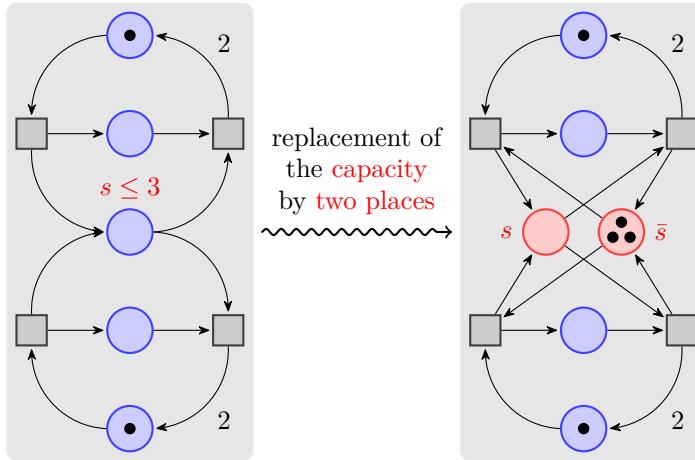
3 Tutorial: A Petri-Net for Hagen

In this second tutorial we explore the node mechanism of TikZ and PGF.

Hagen must give a talk tomorrow about his favorite formalism for distributed systems: Petri nets! Hagen used to give his talks using a blackboard and everyone seemed to be perfectly content with this. Unfortunately, his audience has been spoiled recently with fancy projector-based presentations and there seems to be a certain amount of peer pressure that his Petri nets should also be drawn using a graphic program. One of the professors at his institute recommends TikZ for this and Hagen decides to give it a try.

3.1 Problem Statement

For his talk, Hagen wishes to create a graphic that demonstrates how a net with place capacities can be simulated by a net without capacities. The graphic should look like this, ideally:



replacement of
the capacity
by two places

3.2 Setting up the Environment

For the picture Hagen will need to load the TikZ package as did Karl in the previous tutorial. However, Hagen will also need to load some additional *library packages* that Karl did not need. These library packages contain additional definitions like extra arrow tips that are typically not needed in a picture and that need to be loaded explicitly.

Hagen will need to load several libraries: The `arrows` library for the special arrow tip used in the graphic, the `decorations.pathmorphing` library for the “snaking line” in the middle, the `backgrounds` library for the two rectangular areas that are behind the two main parts of the picture, the `fit` library to easily compute the sizes of these rectangles, and the `positioning` library for placing nodes relative to other nodes.

3.2.1 Setting up the Environment in L^AT_EX

When using L^AT_EX use:

```
\documentclass{article} % say

\usepackage{tikz}
\usetikzlibrary{arrows,decorations.pathmorphing,backgrounds,positioning,fit,petri}

\begin{document}
\begin{tikzpicture}
  \draw (0,0) -- (1,1);
\end{tikzpicture}
\end{document}
```

3.2.2 Setting up the Environment in Plain T_EX

When using plain T_EX use:

```
%% Plain TeX file
\input tikz.tex
\usetikzlibrary{arrows,decorations.pathmorphing,backgrounds,positioning,fit,petri}
\baselineskip=12pt
\hsize=6.3truein
\vsize=8.7truein
\tikzpicture
  \draw (0,0) -- (1,1);
\endtikzpicture
\bye
```

3.2.3 Setting up the Environment in ConTeXt

When using ConTeXt, use:

```
%% ConTeXt file
\usemodule[tikz]
\usetikzlibrary[arrows,decorations.pathmorphing,backgrounds,positioning,fit,petri]

\starttext
  \starttikzpicture
    \draw (0,0) -- (1,1);
  \stoptikzpicture
\stoptext
```

3.3 Introduction to Nodes

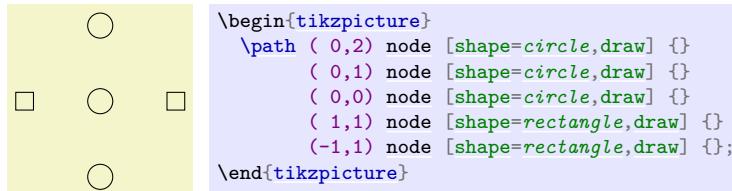
In principle, we already know how to create the graphics that Hagen desires (except perhaps for the snaked line, we will come to that): We start with big light gray rectangle and then add lots of circles and small rectangle, plus some arrows.

However, this approach has numerous disadvantages: First, it is hard to change anything at a later stage. For example, if we decide to add more places to the Petri nets (the circles are called places in Petri net theory), all of the coordinates change and we need to recalculate everything. Second, it is hard to read the code for the Petri net as it is just a long and complicated list of coordinates and drawing commands – the underlying structure of the Petri net is lost.

Fortunately, TikZ offers a powerful mechanism for avoiding the above problems: nodes. We already came across nodes in the previous tutorial, where we used them to add labels to Karl's graphic. In the present tutorial we will see that nodes are much more powerful.

A node is a small part of a picture. When a node is created, you provide a position where the node should be drawn and a *shape*. A node of shape `circle` will be drawn as a circle, a node of shape `rectangle` as a rectangle, and so on. A node may also contain some text, which is why Karl used nodes to show text. Finally, a node can get a *name* for later reference.

In Hagen's picture we will use nodes for the places and for the transitions of the Petri net (the places are the circles, the transitions are the rectangles). Let us start with the upper half of the left Petri net. In this upper half we have three places and two transitions. Instead of drawing three circles and two rectangles, we use three nodes of shape `circle` and two nodes of shape `rectangle`.



Hagen notes that this does not quite look like the final picture, but it seems like a good first step.

Let us have a more detailed look at the code. The whole picture consists of a single path. Ignoring the `node` operations, there is not much going on in this path: It is just a sequence of coordinates with nothing “happening” between them. Indeed, even if something were to happen like a line-to or a curve-to, the `\path` command would not “do” anything with the resulting path. So, all the magic must be in the `node` commands.

In the previous tutorial we learned that a `node` will add a piece of text at the last coordinate. Thus, each of the five nodes is added at a different position. In the above code, this text is empty (because of the

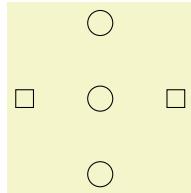
empty `{}`). So, why do we see anything at all? The answer is the `draw` option for the `node` operation: It causes the “shape around the text” to be drawn.

So, the code `(0,2) node [shape=circle,draw] {}` means the following: “In the main path, add a move-to to the coordinate `(0,2)`. Then, temporarily suspend the construction of the main path while the node is built. This node will be a `circle` around an empty text. This circle is to be `drawn`, but not filled or otherwise used. Once this whole node is constructed, it is saved until after the main path is finished. Then, it is drawn.” The following `(0,1) node [shape=circle,draw] {}` then has the following effect: “Continue the main path with a move-to to `(0,1)`. Then construct a node at this position also. This node is also shown after the main path is finished.” And so on.

3.4 Placing Nodes Using the At Syntax

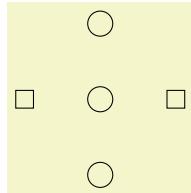
Hagen now understands how the `node` operation adds nodes to the path, but it seems a bit silly to create a path using the `\path` operation, consisting of numerous superfluous move-to operations, only to place nodes. He is pleased to learn that there are ways to add nodes in a more sensible manner.

First, the `node` operation allows one to add `at (<coordinate>)` in order to directly specify where the node should be placed, sidestepping the rule that nodes are placed on the last coordinate. Hagen can then write the following:



```
\begin{tikzpicture}
  \path node at ( 0,2) [shape=circle,draw] {};
  node at ( 0,1) [shape=circle,draw] {};
  node at ( 0,0) [shape=circle,draw] {};
  node at ( 1,1) [shape=rectangle,draw] {};
  node at (-1,1) [shape=rectangle,draw] {};
\end{tikzpicture}
```

Now Hagen is still left with a single empty path, but at least the path no longer contains strange move-to's. It turns out that this can be improved further: The `\node` command is an abbreviation for `\path node`, which allows Hagen to write:

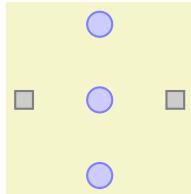


```
\begin{tikzpicture}
  \node at ( 0,2) [circle,draw] {};
  \node at ( 0,1) [circle,draw] {};
  \node at ( 0,0) [circle,draw] {};
  \node at ( 1,1) [rectangle,draw] {};
  \node at (-1,1) [rectangle,draw] {};
\end{tikzpicture}
```

Hagen likes this syntax much better than the previous one. Note that Hagen has also omitted the `shape=` since, like `color=`, TikZ allows you to omit the `shape=` if there is no confusion.

3.5 Using Styles

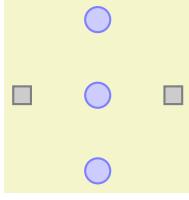
Feeling adventurous, Hagen tries to make the nodes look nicer. In the final picture, the circles and rectangle should be filled with different colors, resulting in the following code:



```
\begin{tikzpicture}[thick]
  \node at ( 0,2) [circle,draw=blue!50,fill=blue!20] {};
  \node at ( 0,1) [circle,draw=blue!50,fill=blue!20] {};
  \node at ( 0,0) [circle,draw=blue!50,fill=blue!20] {};
  \node at ( 1,1) [rectangle,draw=black!50,fill=black!20] {};
  \node at (-1,1) [rectangle,draw=black!50,fill=black!20] {};
\end{tikzpicture}
```

While this looks nicer in the picture, the code starts to get a bit ugly. Ideally, we would like our code to transport the message “there are three places and two transitions” and not so much which filling colors should be used.

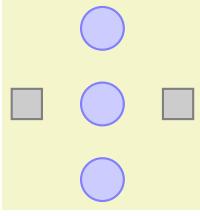
To solve this problem, Hagen uses styles. He defines a style for places and another style for transitions:



```
\begin{tikzpicture}
[place/.style={circle,draw=blue!50,fill=blue!20,thick},
 transition/.style={rectangle,draw=black!50,fill=black!20,thick}]
\node at ( 0,2) [place] {};
\node at ( 0,1) [place] {};
\node at ( 0,0) [place] {};
\node at ( 1,1) [transition] {};
\node at (-1,1) [transition] {};
\end{tikzpicture}
```

3.6 Node Size

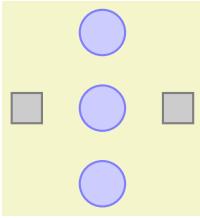
Before Hagen starts naming and connecting the nodes, let us first make sure that the nodes get their final appearance. They are still too small. Indeed, Hagen wonders why they have any size at all, after all, the text is empty. The reason is that TikZ automatically adds some space around the text. The amount is set using the option `inner sep`. So, to increase the size of the nodes, Hagen could write:



```
\begin{tikzpicture}
[inner sep=2mm,
place/.style={circle,draw=blue!50,fill=blue!20,thick},
transition/.style={rectangle,draw=black!50,fill=black!20,thick}]
\node at ( 0,2) [place] {};
\node at ( 0,1) [place] {};
\node at ( 0,0) [place] {};
\node at ( 1,1) [transition] {};
\node at (-1,1) [transition] {};
\end{tikzpicture}
```

However, this is not really the best way to achieve the desired effect. It is much better to use the `minimum size` option instead. This option allows Hagen to specify a minimum size that the node should have. If the node actually needs to be bigger because of a longer text, it will be larger, but if the text is empty, then the node will have `minimum size`. This option is also useful to ensure that several nodes containing different amounts of text have the same size. The options `minimum height` and `minimum width` allow you to specify the minimum height and width independently.

So, what Hagen needs to do is to provide `minimum size` for the nodes. To be on the safe side, he also sets `inner sep=0pt`. This ensures that the nodes will really have size `minimum size` and not, for very small `minimum sizes`, the minimal size necessary to encompass the automatically added space.



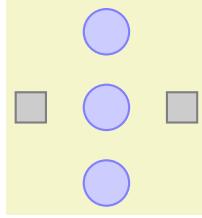
```
\begin{tikzpicture}
[place/.style={circle,draw=blue!50,fill=blue!20,thick,
inner sep=0pt,minimum size=6mm},
transition/.style={rectangle,draw=black!50,fill=black!20,thick,
inner sep=0pt,minimum size=4mm}]
\node at ( 0,2) [place] {};
\node at ( 0,1) [place] {};
\node at ( 0,0) [place] {};
\node at ( 1,1) [transition] {};
\node at (-1,1) [transition] {};
\end{tikzpicture}
```

3.7 Naming Nodes

Hagen's next aim is to connect the nodes using arrows. This seems like a tricky business since the arrows should not start in the middle of the nodes, but somewhere on the border and Hagen would very much like to avoid computing these positions by hand.

Fortunately, PGF will perform all the necessary calculations for him. However, he first has to assign names to the nodes so that he can reference them later on.

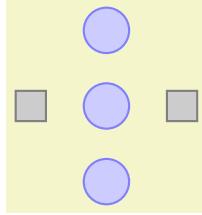
There are two ways to name a node. The first is to use the `name=` option. The second method is to write the desired name in parentheses after the `node` operation. Hagen thinks that this second method seems strange, but he will soon change his opinion.



```
% ... set up styles
\begin{tikzpicture}
  \node (waiting 1)      at ( 0,2) [place] {};
  \node (critical 1)    at ( 0,1) [place] {};
  \node (semaphore)     at ( 0,0) [place] {};
  \node (leave critical) at ( 1,1) [transition] {};
  \node (enter critical) at (-1,1) [transition] {};
\end{tikzpicture}
```

Hagen is pleased to note that the names help in understanding the code. Names for nodes can be pretty arbitrary, but they should not contain commas, periods, parentheses, colons, and some other special characters. However, they can contain underscores and hyphens.

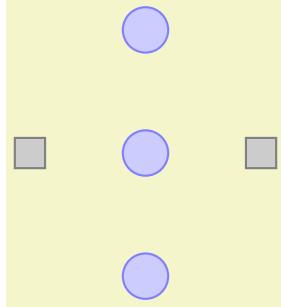
The syntax for the `node` operation is quite liberal with respect to the order in which node names, the `at` specifier, and the options must come. Indeed, you can even have multiple option blocks between the `node` and the text in curly braces, they accumulate. You can rearrange them arbitrarily and perhaps the following might be preferable:



```
\begin{tikzpicture}
  \node[place] (waiting 1)      at ( 0,2) {};
  \node[place] (critical 1)    at ( 0,1) {};
  \node[place] (semaphore)     at ( 0,0) {};
  \node[transition] (leave critical) at ( 1,1) {};
  \node[transition] (enter critical) at (-1,1) {};
\end{tikzpicture}
```

3.8 Placing Nodes Using Relative Placement

Although Hagen still wishes to connect the nodes, he first wishes to address another problem again: The placement of the nodes. Although he likes the `at` syntax, in this particular case he would prefer placing the nodes “relative to each other”. So, Hagen would like to say that the `critical 1` node should be below the `waiting 1` node, wherever the `waiting 1` node might be. There are different ways of achieving this, but the nicest one in Hagen’s case is the `below` option:



```
\usetikzlibrary {positioning}
\begin{tikzpicture}
  \node[place] (waiting)      [below=of critical] {};
  \node[place] (critical)    [below=of waiting] {};
  \node[place] (semaphore)   [below=of critical] {};
  \node[transition] (leave critical) [right=of critical] {};
  \node[transition] (enter critical) [left=of critical] {};
\end{tikzpicture}
```

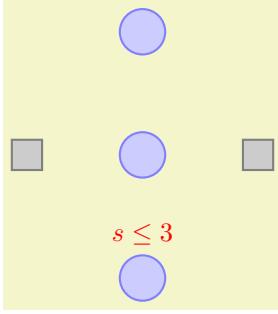
With the `positioning` library loaded, when an option like `below` is followed by `of`, then the position of the node is shifted in such a manner that it is placed at the distance `node distance` in the specified direction of the given direction. The `node distance` is either the distance between the centers of the nodes (when the `on grid` option is set to true) or the distance between the borders (when the `on grid` option is set to false, which is the default).

Even though the above code has the same effect as the earlier code, Hagen can pass it to his colleagues who will be able to just read and understand it, perhaps without even having to see the picture.

3.9 Adding Labels Next to Nodes

Before we have a look at how Hagen can connect the nodes, let us add the capacity “ $s \leq 3$ ” to the bottom node. For this, two approaches are possible:

1. Hagen can just add a new node above the `north` anchor of the `semaphore` node.

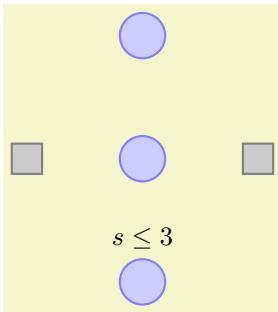


```
\usetikzlibrary {positioning}
\begin{tikzpicture}
  \node[place] (waiting) (waiting);
  \node[place] (critical) [below=of waiting] (critical);
  \node[place] (semaphore) [below=of critical] (semaphore);
  \node[transition] (leave critical) [right=of critical] (leave critical);
  \node[transition] (enter critical) [left=of critical] (enter critical);

  \node [red,above] at (semaphore.north) {$s \leq 3$};
\end{tikzpicture}
```

This is a general approach that will “always work”.

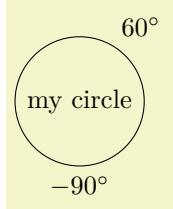
- Hagen can use the special `label` option. This option is given to a `node` and it causes *another* node to be added next to the node where the option is given. Here is the idea: When we construct the `semaphore` node, we wish to indicate that we want another node with the capacity above it. For this, we use the option `label=above:$s \leq 3$`. This option is interpreted as follows: We want a node above the `semaphore` node and this node should read “ $s \leq 3$ ”. Instead of `above` we could also use things like `below left` before the colon or a number like 60.



```
\usetikzlibrary {positioning}
\begin{tikzpicture}
  \node[place] (waiting) (waiting);
  \node[place] (critical) [below=of waiting] (critical);
  \node[place] (semaphore) [below=of critical,
                           label=above:$s \leq 3$] (semaphore);
  \node[transition] (leave critical) [right=of critical] (leave critical);
  \node[transition] (enter critical) [left=of critical] (enter critical);

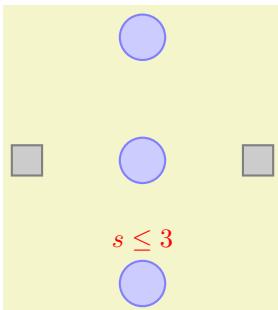
\end{tikzpicture}
```

It is also possible to give multiple `label` options, this causes multiple labels to be drawn.



```
\tikz
  \node [circle,draw,label=60:$60^\circ$ circ$,label=below:$-90^\circ$ circ$] {my circle};
```

Hagen is not fully satisfied with the `label` option since the label is not red. To achieve this, he has two options: First, he can redefine the `every label` style. Second, he can add options to the label’s node. These options are given following the `label=`, so he would write `label={[red]above:$s \leq 3$}`. However, this does not quite work since TeX thinks that the] closes the whole option list of the `semaphore` node. So, Hagen has to add braces and writes `label={[red]above:$s \leq 3$}`. Since this looks a bit ugly, Hagen decides to redefine the `every label` style.

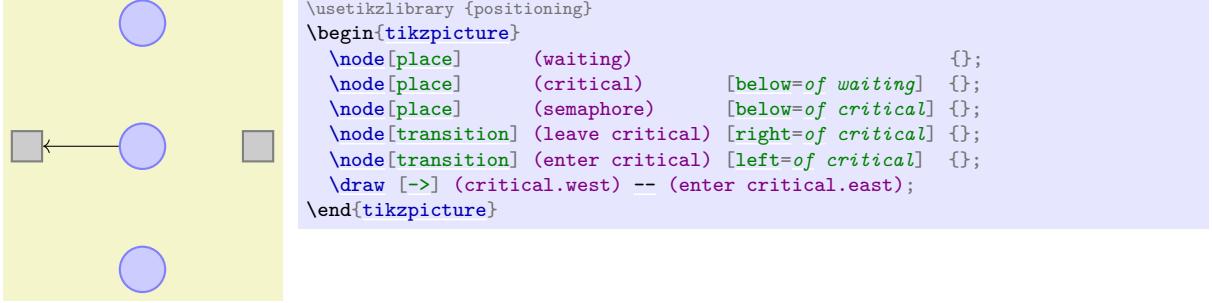


```
\usetikzlibrary {positioning}
\begin{tikzpicture} [every label/.style={red}]
  \node[place] (waiting) (waiting);
  \node[place] (critical) [below=of waiting] (critical);
  \node[place] (semaphore) [below=of critical,
                           label=above:$s \leq 3$] (semaphore);
  \node[transition] (leave critical) [right=of critical] (leave critical);
  \node[transition] (enter critical) [left=of critical] (enter critical);

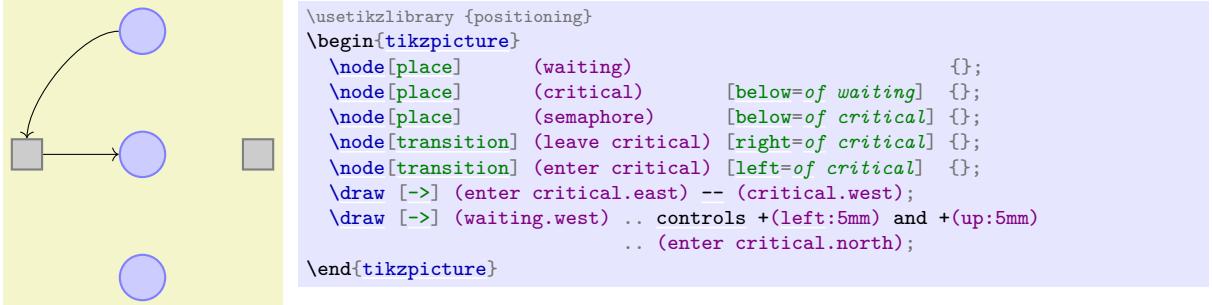
\end{tikzpicture}
```

3.10 Connecting Nodes

It is now high time to connect the nodes. Let us start with something simple, namely with the straight line from `enter critical` to `critical`. We want this line to start at the right side of `enter critical` and to end at the left side of `critical`. For this, we can use the *anchors* of the nodes. Every node defines a whole bunch of anchors that lie on its border or inside it. For example, the `center` anchor is at the center of the node, the `west` anchor is on the left of the node, and so on. To access the coordinate of a node, we use a coordinate that contains the node's name followed by a dot, followed by the anchor's name:

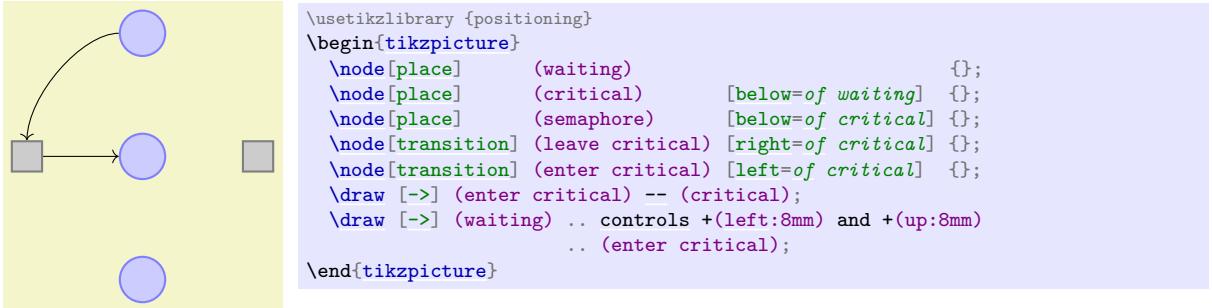


Next, let us tackle the curve from `waiting` to `enter critical`. This can be specified using curves and controls:



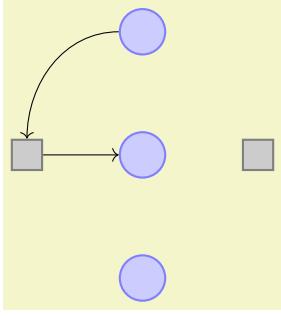
Hagen sees how he can now add all his edges, but the whole process seems a bit awkward and not very flexible. Again, the code seems to obscure the structure of the graphic rather than showing it.

So, let us start improving the code for the edges. First, Hagen can leave out the anchors:



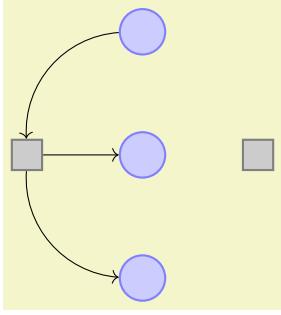
Hagen is a bit surprised that this works. After all, how did TikZ know that the line from `enter critical` to `critical` should actually start on the borders? Whenever TikZ encounters a whole node name as a “coordinate”, it tries to “be smart” about the anchor that it should choose for this node. Depending on what happens next, TikZ will choose an anchor that lies on the border of the node on a line to the next coordinate or control point. The exact rules are a bit complex, but the chosen point will usually be correct – and when it is not, Hagen can still specify the desired anchor by hand.

Hagen would now like to simplify the curve operation somehow. It turns out that this can be accomplished using a special path operation: the `to` operation. This operation takes many options (you can even define new ones yourself). One pair of options is useful for Hagen: The pair `in` and `out`. These options take angles at which a curve should leave or reach the start or target coordinates. Without these options, a straight line is drawn:



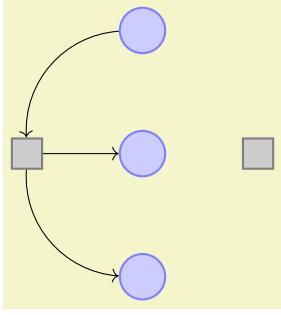
```
\usetikzlibrary {positioning}
\begin{tikzpicture}
  \node[place]      (waiting)
  \node[place]      (critical)      [below=of waiting] {};
  \node[place]      (semaphore)    [below=of critical] {};
  \node[transition] (leave critical) [right=of critical] {};
  \node[transition] (enter critical) [left=of critical] {};
  \draw [->] (enter critical) to (critical);
  \draw [->] (waiting)          to [out=180,in=90] (enter critical);
\end{tikzpicture}
```

There is another option for the `to` operation, that is even better suited to Hagen's problem: The `bend right` option. This option also takes an angle, but this angle only specifies the angle by which the curve is bent to the right:



```
\usetikzlibrary {positioning}
\begin{tikzpicture}
  \node[place]      (waiting)
  \node[place]      (critical)      [below=of waiting] {};
  \node[place]      (semaphore)    [below=of critical] {};
  \node[transition] (leave critical) [right=of critical] {};
  \node[transition] (enter critical) [left=of critical] {};
  \draw [->] (enter critical) to (critical);
  \draw [->] (waiting)          to [bend right=45] (enter critical);
  \draw [->] (enter critical) to [bend right=45] (semaphore);
\end{tikzpicture}
```

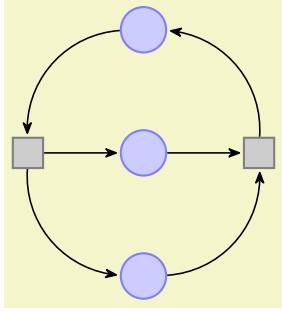
It is now time for Hagen to learn about yet another way of specifying edges: Using the `edge` path operation. This operation is very similar to the `to` operation, but there is one important difference: Like a node the edge generated by the `edge` operation is not part of the main path, but is added only later. This may not seem very important, but it has some nice consequences. For example, every edge can have its own arrow tips and its own color and so on and, still, all the edges can be given on the same path. This allows Hagen to write the following:



```
\usetikzlibrary {positioning}
\begin{tikzpicture}
  \node[place]      (waiting)
  \node[place]      (critical)      [below=of waiting] {};
  \node[place]      (semaphore)    [below=of critical] {};
  \node[transition] (leave critical) [right=of critical] {};
  \node[transition] (enter critical) [left=of critical] {};
  edge [->] (enter critical) to (critical)
  edge [<-,bend left=45] (waiting)
  edge [->,bend right=45] (semaphore);
\end{tikzpicture}
```

Each `edge` caused a new path to be constructed, consisting of a `to` between the node `enter critical` and the node following the `edge` command.

The finishing touch is to introduce two styles `pre` and `post` and to use the `bend angle=45` option to set the bend angle once and for all:



```
\usetikzlibrary {arrows.meta,positioning}
% Styles place and transition as before
\begin{tikzpicture}
[bend angle=45,
 pre/.style={<-,shorten <=1pt,>={Stealth[round]},semithick},
 post/.style={->,shorten >=1pt,>={Stealth[round]},semithick}]

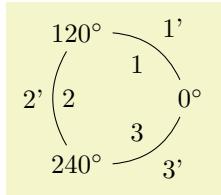
\node[place] (waiting)
\node[place] (critical) [below=of waiting]
\node[place] (semaphore) [below=of critical]

\node[transition] (leave critical) [right=of critical]
edge [pre] (critical)
edge [post,bend right] (waiting)
edge [pre, bend left] (semaphore);
\node[transition] (enter critical) [left=of critical]
edge [post] (critical)
edge [pre, bend left] (waiting)
edge [post,bend right] (semaphore);

\end{tikzpicture}
```

3.11 Adding Labels Next to Lines

The next thing that Hagen needs to add is the “2” at the arcs. For this Hagen can use TikZ’s automatic node placement: By adding the option `auto`, TikZ will position nodes on curves and lines in such a way that they are not on the curve but next to it. Adding `swap` will mirror the label with respect to the line. Here is a general example:



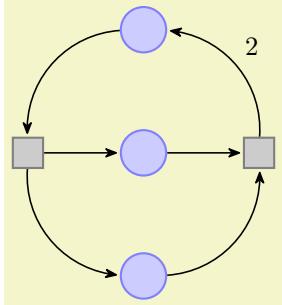
```
\begin{tikzpicture}[auto,bend right]
\node (a) at (0:1) {$0^\circ \textcircled{a}$};
\node (b) at (120:1) {$120^\circ \textcircled{b}$};
\node (c) at (240:1) {$240^\circ \textcircled{c}$};

\draw (a) to node {1} node [swap] {1'} (b)
(b) to node {2} node [swap] {2'} (c)
(c) to node {3} node [swap] {3'} (a);

\end{tikzpicture}
```

What is happening here? The nodes are given somehow inside the `to` operation! When this is done, the node is placed on the middle of the curve or line created by the `to` operation. The `auto` option then causes the node to be moved in such a way that it does not lie on the curve, but next to it. In the example we provide even two nodes on each `to` operation.

For Hagen that `auto` option is not really necessary since the two “2” labels could also easily be placed “by hand”. However, in a complicated plot with numerous edges automatic placement can be a blessing.



```
\usetikzlibrary {arrows.meta,positioning}
% Styles as before
\begin{tikzpicture}[bend angle=45]
\node[place] (waiting)
\node[place] (critical) [below=of waiting]
\node[place] (semaphore) [below=of critical]

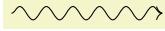
\node[transition] (leave critical) [right=of critical]
edge [pre] (critical)
edge [post,bend right] node[auto,swap] {2} (waiting)
edge [pre, bend left] (semaphore);
\node[transition] (enter critical) [left=of critical]
edge [post] (critical)
edge [pre, bend left] (waiting)
edge [post,bend right] (semaphore);

\end{tikzpicture}
```

3.12 Adding the Snaked Line and Multi-Line Text

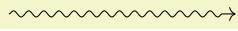
With the node mechanism Hagen can now easily create the two Petri nets. What he is unsure of is how he can create the snaked line between the nets.

For this he can use a *decoration*. To draw the snaked line, Hagen only needs to set the two options `decoration=snake` and `decorate` on the path. This causes all lines of the path to be replaced by snakes. It is also possible to use snakes only in certain parts of a path, but Hagen will not need this.



```
\usetikzlibrary {decorations.pathmorphing}
\begin{tikzpicture}
  \draw [->,decorate,decoration=snake] (0,0) -- (2,0);
\end{tikzpicture}
```

Well, that does not look quite right, yet. The problem is that the snake happens to end exactly at the position where the arrow begins. Fortunately, there is an option that helps here. Also, the snake should be a bit smaller, which can be influenced by even more options.



```
\usetikzlibrary {decorations.pathmorphing}
\begin{tikzpicture}
  \draw [->,decorate,
    decoration={snake,amplitude=.4mm,segment length=2mm,post length=1mm}]
  (0,0) -- (3,0);
\end{tikzpicture}
```

Now Hagen needs to add the text above the snake. This text is a bit challenging since it is a multi-line text. Hagen has two options for this: First, he can specify an `align=center` and then use the `\backslash` command to enforce the line breaks at the desired positions.

replacement of
the capacity
by two places

```
\usetikzlibrary {decorations.pathmorphing}
\begin{tikzpicture}
  \draw [->,decorate,
    decoration={snake,amplitude=.4mm,segment length=2mm,post length=1mm}]
  (0,0) -- (3,0)
  node [above,align=center,midway]
  {
    replacement of\\
    the \textcolor{red}{capacity}\\
    by \textcolor{red}{two places}
  };
\end{tikzpicture}
```

Instead of specifying the line breaks “by hand”, Hagen can also specify a width for the text and let `TEX` perform the line breaking for him:

replacement of
the capacity
by two places

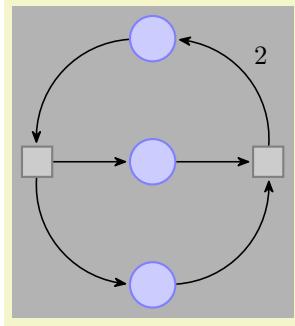
```
\usetikzlibrary {decorations.pathmorphing}
\begin{tikzpicture}
  \draw [->,decorate,
    decoration={snake,amplitude=.4mm,segment length=2mm,post length=1mm}]
  (0,0) -- (3,0)
  node [above,text width=3cm,align=center,midway]
  {
    replacement of the \textcolor{red}{capacity} by \\
    \textcolor{red}{two places}
  };
\end{tikzpicture}
```

3.13 Using Layers: The Background Rectangles

Hagen still needs to add the background rectangles. These are a bit tricky: Hagen would like to draw the rectangles *after* the Petri nets are finished. The reason is that only then can he conveniently refer to the coordinates that make up the corners of the rectangle. If Hagen draws the rectangle first, then he needs to know the exact size of the Petri net – which he does not.

The solution is to use *layers*. When the `backgrounds` library is loaded, Hagen can put parts of his picture inside a scope with the `on background layer` option. Then this part of the picture becomes part of the layer that is given as an argument to this environment. When the `{tikzpicture}` environment ends, the layers are put on top of each other, starting with the background layer. This causes everything drawn on the background layer to be behind the main text.

The next tricky question is, how big should the rectangle be? Naturally, Hagen can compute the size “by hand” or using some clever observations concerning the *x*- and *y*-coordinates of the nodes, but it would be nicer to just have TikZ compute a rectangle into which all the nodes “fit”. For this, the `fit` library can be used. It defines the `fit` options, which, when given to a node, causes the node to be resized and shifted such that it exactly covers all the nodes and coordinates given as parameters to the `fit` option.



```
\usetikzlibrary {arrows.meta,backgrounds,fit,positioning}
% Styles as before
\begin{tikzpicture}[bend angle=45]
\node[place] (waiting) [below=of critical] {};
\node[place] (critical) [below=of waiting] {};
\node[place] (semaphore) [below=of critical] {};

\node[transition] (leave critical) [right=of critical] {};
edge [pre] (leave critical) (critical);
edge [post,bend right] node[auto,swap] {2} (waiting);
edge [pre, bend left] (waiting) (semaphore);
\node[transition] (enter critical) [left=of critical] {};
edge [post] (enter critical) (critical);
edge [pre, bend left] (critical) (waiting);
edge [post,bend right] (semaphore);

\begin{scope}[on background layer]
\node [fill=black!30,fit=(waiting) (critical) (semaphore)
      (leave critical) (enter critical)] {};
\end{scope}
\end{tikzpicture}
```

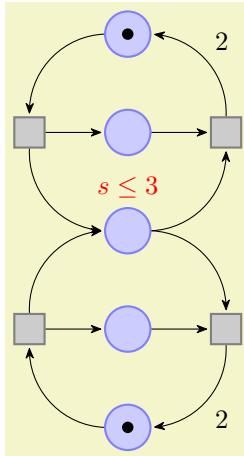
3.14 The Complete Code

Hagen has now finally put everything together. Only then does he learn that there is already a library for drawing Petri nets! It turns out that this library mainly provides the same definitions as Hagen did. For example, it defines a `place` style in a similar way as Hagen did. Adjusting the code so that it uses the library shortens Hagen code a bit, as shown in the following.

First, Hagen needs less style definitions, but he still needs to specify the colors of places and transitions.

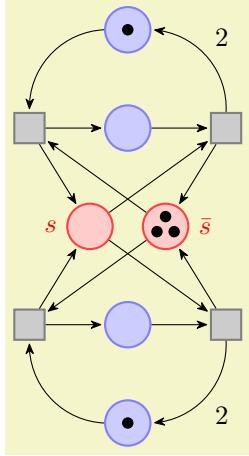
```
\begin{tikzpicture}
[node distance=1.3cm,on grid,>={Stealth[round]},bend angle=45,auto,
every place/.style= {minimum size=6mm,thick,draw=blue!75,fill=blue!20},
every transition/.style= {thick,draw=black!75,fill=black!20},
red place/.style= {place,draw=red!75,fill=red!20},
every label/.style= {red}]
```

Now comes the code for the nets:



```
\usetikzlibrary {arrows.meta,petri,positioning}
\node [place,tokens=1] (w1) {};
\node [place] (c1) [below=of w1] {};
\node [place] (s) [below=of c1,label=above:$s \leq 3$] {};
\node [place] (c2) [below=of s] {};
\node [place,tokens=1] (w2) [below=of c2] {};

\node [transition] (e1) [left=of c1] {};
edge [pre,bend left] (e1) (w1);
edge [post,bend right] (e1) (s);
edge [post] (e1) (c1);
\node [transition] (e2) [left=of c2] {};
edge [pre,bend right] (e2) (w2);
edge [post,bend left] (e2) (s);
edge [post] (e2) (c2);
\node [transition] (l1) [right=of c1] {};
edge [pre] (l1) (c1);
edge [pre,bend left] (l1) (s);
edge [post,bend right] node[swap] {2} (w1);
\node [transition] (l2) [right=of c2] {};
edge [pre] (l2) (c2);
edge [pre,bend right] (l2) (s);
edge [post,bend left] node {2} (w2);
```



```
\usetikzlibrary {arrows.meta,petri,positioning}
\begin{scope}[xshift=6cm]
\node [place,tokens=1] (w1') (c1') [below=of w1'] (s1') [below=of c1',xshift=-5mm] (s2') [below=of c1',xshift=5mm] (c2') [below=of s1',xshift=5mm] (w2') [below=of c2'];
\node [red place,tokens=3] (s1bar) [below=of c1',xshift=5mm] (label=right:$\bar{s}$);
\node [place] (c1bar) [below=of s1',xshift=5mm];
\node [place,tokens=1] (w1bar) [below=of c1bar];
\node [transition] (e1') [left=of c1'] {};
edge [pre,bend left] (w1') (e1');
edge [post] (e1') (s1');
edge [pre] (e1') (s2');
edge [post] (e1') (c1bar);
\node [transition] (e2') [left=of c2'] {};
edge [pre,bend right] (w2') (e2');
edge [post] (e2') (s1');
edge [pre] (e2') (s2');
edge [post] (e2') (c2');
\node [transition] (l1') [right=of c1'] {};
edge [pre] (c1') (l1');
edge [pre] (s1') (l1');
edge [post] (s2') (l1');
edge [post,bend right] node[swap]{2} (w1') (l1');
\node [transition] (l2') [right=of c2'] {};
edge [pre] (c2') (l2');
edge [pre] (s1') (l2');
edge [post] (s2') (l2');
edge [post,bend left] node{2} (w2') (l2');
\end{scope}
```

The code for the background and the snake is the following:

```
\begin{tikzpicture}[on background layer]
\node (r1) [fill=black!10,rounded corners,fit=(w1)(w2)(e1)(e2)(l1)(l2)] {};
\node (r2) [fill=black!10,rounded corners,fit=(w1')(w2')(e1')(e2')(l1')(l2')] {};
\end{scope}

\draw [shorten >=1mm,-to,thick,decorate,
decoration={snake,amplitude=.4mm,segment length=2mm,
pre=moveto,pre length=1mm,post length=2mm}]
(r1) -- (r2) node [above=1mm,midway,text width=3cm,align=center]
{replacement of the \textcolor{red}{\{capacity\}} by \textcolor{red}{\{two places\}}};
\end{tikzpicture}
```

4 Tutorial: Euclid's Amber Version of the *Elements*

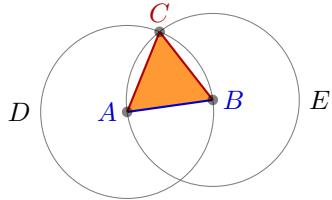
In this third tutorial we have a look at how TikZ can be used to draw geometric constructions.

Euclid is currently quite busy writing his new book series, whose working title is “Elements” (Euclid is not quite sure whether this title will convey the message of the series to future generations correctly, but he intends to change the title before it goes to the publisher). Up to know, he wrote down his text and graphics on papyrus, but his publisher suddenly insists that he must submit in electronic form. Euclid tries to argue with the publisher that electronics will only be discovered thousands of years later, but the publisher informs him that the use of papyrus is no longer cutting edge technology and Euclid will just have to keep up with modern tools.

Slightly disgruntled, Euclid starts converting his papyrus entitled “Book I, Proposition I” to an amber version.

4.1 Book I, Proposition I

The drawing on his papyrus looks like this:¹



Proposition I

To construct an *equilateral triangle* on a given *finite straight line*.

Let AB be the given *finite straight line*. It is required to construct an *equilateral triangle* on the *straight line* AB .

Describe the circle BCD with center A and radius AB . Again describe the circle ACE with center B and radius BA . Join the *straight lines* CA and CB from the point C at which the circles cut one another to the points A and B .

Now, since the point A is the center of the circle CD , therefore AC equals AB . Again, since the point B is the center of the circle CAE , therefore BC equals BA . But AC was proved equal to AB , therefore each of the straight lines AC and BC equals AB . And things which equal the same thing also equal one another, therefore AC also equals BC . Therefore the three straight lines AC , AB , and BC equal one another. Therefore the *triangle* ABC is equilateral, and it has been constructed on the given finite *straight line* AB .

Let us have a look at how Euclid can turn this into TikZ code.

4.1.1 Setting up the Environment

As in the previous tutorials, Euclid needs to load TikZ, together with some libraries. These libraries are `calc`, `intersections`, `through`, and `backgrounds`. Depending on which format he uses, Euclid would use one of the following in the preamble:

```
% For LaTeX:  
\usepackage{tikz}  
\usetikzlibrary{calc,intersections,through,backgrounds}
```

```
% For plain TeX:  
\input tikz.tex  
\usetikzlibrary{calc,intersections,through,backgrounds}
```

```
% For ConTeXt:  
\usemodule[tikz]  
\usetikzlibrary[calc,intersections,through,backgrounds]
```

¹The text is taken from the wonderful interactive version of Euclid's Elements by David E. Joyce, to be found on his website at Clark University.

4.1.2 The Line AB

The first part of the picture that Euclid wishes to draw is the line AB . That is easy enough, something like `\draw (0,0) --(2,1);` might do. However, Euclid does not wish to reference the two points A and B as $(0,0)$ and $(2,1)$ subsequently. Rather, he wishes to just write A and B . Indeed, the whole point of his book is that the points A and B can be arbitrary and all other points (like C) are constructed in terms of their positions. It would not do if Euclid were to write down the coordinates of C explicitly.

So, Euclid starts with defining two coordinates using the `\coordinate` command:



```
\begin{tikzpicture}
\coordinate (A) at (0,0);
\coordinate (B) at (1.25,0.25);

\draw[blue] (A) -- (B);
\end{tikzpicture}
```

That was easy enough. What is missing at this point are the labels for the coordinates. Euclid does not want them *on* the points, but next to them. He decides to use the `label` option:



```
\begin{tikzpicture}
\coordinate [label=left:\textcolor{blue}{\$A\$}] (A) at (0,0);
\coordinate [label=right:\textcolor{blue}{\$B\$}] (B) at (1.25,0.25);

\draw[blue] (A) -- (B);
\end{tikzpicture}
```

At this point, Euclid decides that it would be even nicer if the points A and B were in some sense “random”. Then, neither Euclid nor the reader can make the mistake of taking “anything for granted” concerning these position of these points. Euclid is pleased to learn that there is a `rand` function in TikZ that does exactly what he needs: It produces a number between -1 and 1 . Since TikZ can do a bit of math, Euclid can change the coordinates of the points as follows:

```
\coordinate [...] (A) at (0+0.1*rand,0+0.1*rand);
\coordinate [...] (B) at (1.25+0.1*rand,0.25+0.1*rand);
```

This works fine. However, Euclid is not quite satisfied since he would prefer that the “main coordinates” $(0,0)$ and $(1.25,0.25)$ are “kept separate” from the perturbation $0.1(rand,rand)$. This means, he would like to specify that coordinate A as “the point that is at $(0,0)$ plus one tenth of the vector $(rand,rand)$ ”.

It turns out that the `calc` library allows him to do exactly this kind of computation. When this library is loaded, you can use special coordinates that start with `($` and end with `$)` rather than just `(` and `)`. Inside these special coordinates you can give a linear combination of coordinates. (Note that the dollar signs are only intended to signal that a “computation” is going on; no mathematical typesetting is done.)

The new code for the coordinates is the following:

```
\coordinate [...] (A) at ($ (0,0) + .1*(rand,rand) $);
\coordinate [...] (B) at ($ (1.25,0.25) + .1*(rand,rand) $);
```

Note that if a coordinate in such a computation has a factor (like `.1`), you must place a `*` directly before the opening parenthesis of the coordinate. You can nest such computations.

4.1.3 The Circle Around A

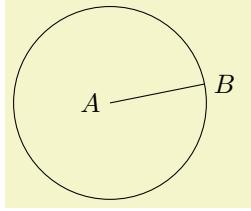
The first tricky construction is the circle around A . We will see later how to do this in a very simple manner, but first let us do it the “hard” way.

The idea is the following: We draw a circle around the point A whose radius is given by the length of the line AB . The difficulty lies in computing the length of this line.

Two ideas “nearly” solve this problem: First, we can write `($ (A) - (B) $)` for the vector that is the difference between A and B . All we need is the length of this vector. Second, given two numbers x and y , one can write `veclen(x,y)` inside a mathematical expression. This gives the value $\sqrt{x^2 + y^2}$, which is exactly the desired length.

The only remaining problem is to access the x - and y -coordinate of the vector AB . For this, we need a new concept: the *let operation*. A let operation can be given anywhere on a path where a normal path operation like a line-to or a move-to is expected. The effect of a let operation is to evaluate some coordinates and to assign the results to special macros. These macros make it easy to access the x - and y -coordinates of the coordinates.

Euclid would write the following:



```
\usetikzlibrary {calc}
\begin{tikzpicture}
  \coordinate [label=left:$A$] (A) at (0,0);
  \coordinate [label=right:$B$] (B) at (1.25,0.25);
  \draw (A) -- (B);

  \draw (A) let
    \p1 = ($ (B) - (A) $)
    in
    circle ({veclen(\x1,\y1)});
\end{tikzpicture}
```

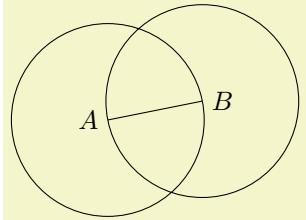
Each assignment in a let operation starts with `\p`, usually followed by a *<digit>*. Then comes an equal sign and a coordinate. The coordinate is evaluated and the result is stored internally. From then on you can use the following expressions:

1. `\x<digit>` yields the *x*-coordinate of the resulting point.
2. `\y<digit>` yields the *y*-coordinate of the resulting point.
3. `\p<digit>` yields the same as `\x<digit>, \y<digit>`.

You can have multiple assignments in a let operation, just separate them with commas. In later assignments you can already use the results of earlier assignments.

Note that `\p1` is not a coordinate in the usual sense. Rather, it just expands to a string like `10pt,20pt`. So, you cannot write, for instance, `(\p1.center)` since this would just expand to `(10pt,20pt.center)`, which makes no sense.

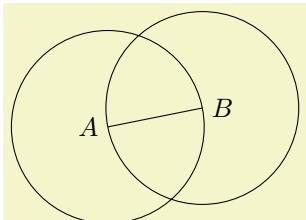
Next, we want to draw both circles at the same time. Each time the radius is `veclen(\x1,\y1)`. It seems natural to compute this radius only once. For this, we can also use a let operation: Instead of writing `\p1 = ...`, we write `\n2 = ...`. Here, “n” stands for “number” (while “p” stands for “point”). The assignment of a number should be followed by a number in curly braces.



```
\usetikzlibrary {calc}
\begin{tikzpicture}
  \coordinate [label=left:$A$] (A) at (0,0);
  \coordinate [label=right:$B$] (B) at (1.25,0.25);
  \draw (A) -- (B);

  \draw let \p1 = ($ (B) - (A) $),
    \n2 = {veclen(\x1,\y1)}
    in
    (A) circle (\n2)
    (B) circle (\n2);
\end{tikzpicture}
```

In the above example, you may wonder, what `\n1` would yield? The answer is that it would be undefined – the `\p`, `\x`, and `\y` macros refer to the same logical point, while the `\n` macro has “its own namespace”. We could even have replaced `\n2` in the example by `\n1` and it would still work. Indeed, the digits following these macros are just normal TeX parameters. We could also use a longer name, but then we have to use curly braces:



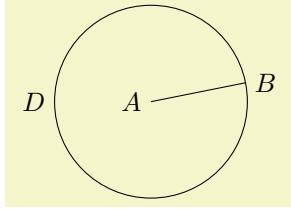
```
\usetikzlibrary {calc}
\begin{tikzpicture}
  \coordinate [label=left:$A$] (A) at (0,0);
  \coordinate [label=right:$B$] (B) at (1.25,0.25);
  \draw (A) -- (B);

  \draw let \p1 = ($ (B) - (A) $),
    \n{radius} = {veclen(\x1,\y1)}
    in
    (A) circle (\n{radius})
    (B) circle (\n{radius});
\end{tikzpicture}
```

At the beginning of this section it was promised that there is an easier way to create the desired circle. The trick is to use the `through` library. As the name suggests, it contains code for creating shapes that go through a given point.

The option that we are looking for is `circle through`. This option is given to a *node* and has the following effects: First, it causes the node’s inner and outer separations to be set to zero. Then it sets the

shape of the node to `circle`. Finally, it sets the radius of the node such that it goes through the parameter given to `circle through`. This radius is computed in essentially the same way as above.



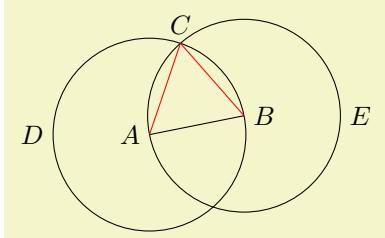
```
\usetikzlibrary {through}
\begin{tikzpicture}
\coordinate [label=left:$A$] (A) at (0,0);
\coordinate [label=right:$B$] (B) at (1.25,0.25);
\draw (A) -- (B);

\node [draw,circle through=(B),label=left:$D$] at (A) {};
\end{tikzpicture}
```

4.1.4 The Intersection of the Circles

Euclid can now draw the line and the circles. The final problem is to compute the intersection of the two circles. This computation is a bit involved if you want to do it “by hand”. Fortunately, the `intersections` library allows us to compute the intersection of arbitrary paths.

The idea is simple: First, you “name” two paths using the `name path` option. Then, at some later point, you can use the option `name intersections`, which creates coordinates called `intersection-1`, `intersection-2`, and so on at all intersections of the paths. Euclid assigns the names `D` and `E` to the paths of the two circles (which happen to be the same names as the nodes themselves, but nodes and their paths live in different “namespaces”).



```
\usetikzlibrary {intersections,through}
\begin{tikzpicture}
\coordinate [label=left:$A$] (A) at (0,0);
\coordinate [label=right:$B$] (B) at (1.25,0.25);
\draw (A) -- (B);

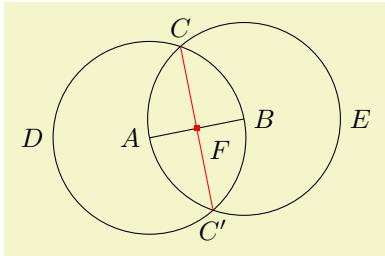
\node (D) [name path=D,draw,circle through=(B),label=left:$D$] at (A) {};
\node (E) [name path=E,draw,circle through=(A),label=right:$E$] at (B) {};

% Name the coordinates, but do not draw anything:
\path [name intersections={of=D and E}];

\coordinate [label=above:$C$] (C) at (intersection-1);
\draw [red] (A) -- (C);
\draw [red] (B) -- (C');

\end{tikzpicture}
```

It turns out that this can be further shortened: The `name intersections` takes an optional argument `by`, which lets you specify names for the coordinates and options for them. This creates more compact code. Although Euclid does not need it for the current picture, it is just a small step to computing the bisection of the line `AB`:



```

\usetikzlibrary {intersections,through}
\begin{tikzpicture}
  \coordinate [label=left:$A$] (A) at (0,0);
  \coordinate [label=right:$B$] (B) at (1.25,0.25);
  \draw [name path=A--B] (A) -- (B);

  \node (D) [name path=D,draw,circle through=(B),label=left:$D$] at (A) {};
  \node (E) [name path=E,draw,circle through=(A),label=right:$E$] at (B) {};

  \path [name intersections={of=D and E, by={[label=above:$C$]C, [label=below:$C'$]C'}}];

  \draw [name path=C--C',red] (C) -- (C');

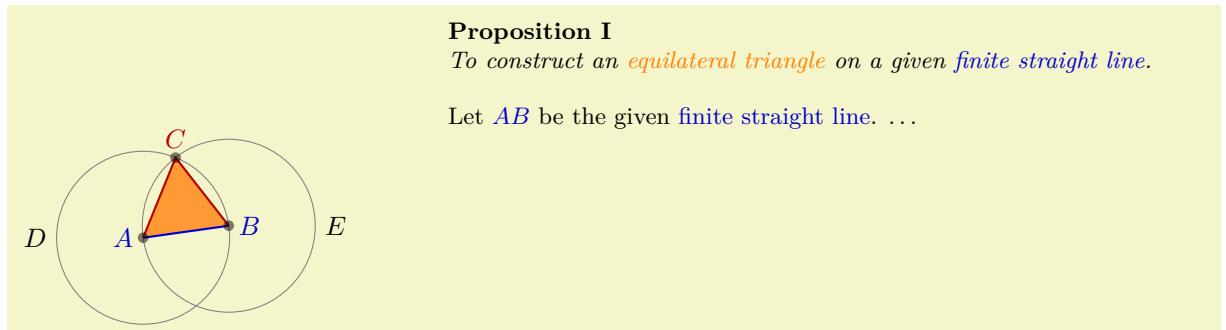
  \path [name intersections={of=A--B and C--C',by=F}];
  \node [fill=red,inner sep=1pt,label=-45:$F$] at (F) {};

\end{tikzpicture}

```

4.1.5 The Complete Code

Back to Euclid's code. He introduces a few macros to make life simpler, like a `\A` macro for typesetting a blue A . He also uses the `background` layer for drawing the triangle behind everything at the end.



```

\usetikzlibrary {backgrounds,calc,intersections,through}
\begin{tikzpicture}[thick,help lines/.style={thin,draw=black!50}]
 \def\A{\textcolor{input}{$\text{A}$}} \def\B{\textcolor{input}{$\text{B}$}}
 \def\C{\textcolor{output}{$\text{C}$}} \def\D{$\text{D}$}
 \def\E{$\text{E}$}

 \colorlet{input}{blue!80!black} \colorlet{output}{red!70!black}
 \colorlet{triangle}{orange}

 \coordinate [label=left:\textcolor{input}{A}] (A) at ($ (0,0) + .1*(rand,rand) $);
 \coordinate [label=right:\textcolor{input}{B}] (B) at ($ (1.25,0.25) + .1*(rand,rand) $);

 \draw [input] (A) -- (B);

 \node [name path=D,help lines,draw,label=left:\textcolor{input}{D}] (D) at (A) [circle through=(B)] {};
 \node [name path=E,help lines,draw,label=right:\textcolor{input}{E}] (E) at (B) [circle through=(A)] {};

 \path [name intersections={of=D and E,by={[label=above:\textcolor{output}{C}]}};

 \draw [output] (A) -- (C) -- (B);

 \foreach \point in {A,B,C}
   \fill [black,opacity=.5] (\point) circle (2pt);

 \begin{pgfonlayer}{background}
   \fill[triangle!80] (A) -- (C) -- (B) -- cycle;
 \end{pgfonlayer}

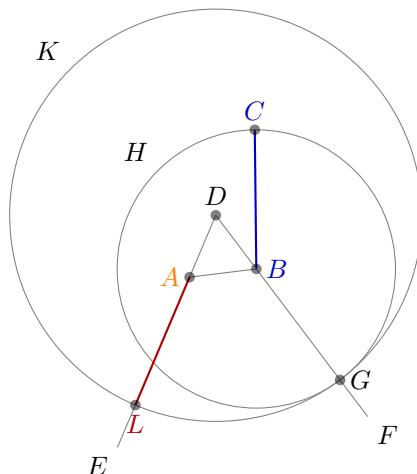
 \node [below right, text width=10cm,align=justify] at (4,3) {
   \small\textbf{Proposition I}\par
   To construct an \textcolor{triangle}{equilateral triangle}\newline
   on a given \textcolor{input}{finite straight line}.}
   \par\vskip1em
   Let \textcolor{input}{A}\textcolor{black}{B} be the given \textcolor{input}{finite straight line}. \dots
 };

\end{tikzpicture}

```

4.2 Book I, Proposition II

The second proposition in the Elements is the following:



Proposition II

To place a straight line equal to a given straight line with one end at a given point.

Let A be the given point, and BC the given straight line. It is required to place a straight line equal to the given straight line BC with one end at the point A .

Join the straight line AB from the point A to the point B , and construct the equilateral triangle DAB on it.

Produce the straight lines AE and BF in a straight line with DA and DB . Describe the circle CGH with center B and radius BC , and again, describe the circle GKL with center D and radius DG .

Since the point B is the center of the circle CGH , therefore BC equals BG . Again, since the point D is the center of the circle GKL , therefore DL equals DG . And in these DA equals DB , therefore the remainder AL equals the remainder BG . But BC was also proved equal to BG , therefore each of the straight lines AL and BC equals BG . And things which equal the same thing also equal one another, therefore AL also equals BC .

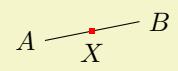
Therefore the straight line AL equal to the given straight line BC has been placed with one end at the given point A .

4.2.1 Using Partway Calculations for the Construction of D

Euclid's construction starts with "referencing" Proposition I for the construction of the point D . Now, while we could simply repeat the construction, it seems a bit bothersome that one has to draw all these circles and do all these complicated constructions.

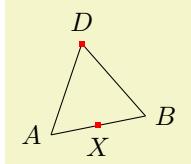
For this reason, TikZ supports some simplifications. First, there is a simple syntax for computing a point that is "partway" on a line from p to q : You place these two points in a coordinate calculation – remember, they start with $(\$$ and end with $\$)$ – and then combine them using $!(part)!$. A $\langle part \rangle$ of 0 refers to the *first* coordinate, a $\langle part \rangle$ of 1 refers to the second coordinate, and a value in between refers to a point on the line from p to q . Thus, the syntax is similar to the `xcolor` syntax for mixing colors.

Here is the computation of the point in the middle of the line AB :



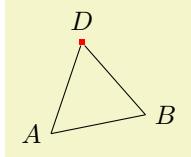
```
\usetikzlibrary {calc}
\begin{tikzpicture}
\coordinate [label=left:$A$] (A) at (0,0);
\coordinate [label=right:$B$] (B) at (1.25,0.25);
\draw (A) -- (B);
\node [fill=red,inner sep=1pt,label=below:$X$] (X) at ($ (A)! .5! (B) $) {};
\end{tikzpicture}
```

The computation of the point D in Euclid's second proposition is a bit more complicated. It can be expressed as follows: Consider the line from X to B . Suppose we rotate this line around X for 90° and then stretch it by a factor of $\sin(60^\circ) \cdot 2$. This yields the desired point D . We can do the stretching using the partway modifier above, for the rotation we need a new modifier: the rotation modifier. The idea is that the second coordinate in a partway computation can be prefixed by an angle. Then the partway point is computed normally (as if no angle were given), but the resulting point is rotated by this angle around the first point.



```
\usetikzlibrary {calc}
\begin{tikzpicture}
\coordinate [label=left:$A$] (A) at (0,0);
\coordinate [label=right:$B$] (B) at (1.25,0.25);
\draw (A) -- (B);
\node [fill=red,inner sep=1pt,label=below:$X$] (X) at ($ (A)! .5! (B) $) {};
\node [fill=red,inner sep=1pt,label=above:$D$] (D) at
($ (X) ! {\sin(60)*2} ! 90:(B) $) {};
\draw (A) -- (D) -- (B);
\end{tikzpicture}
```

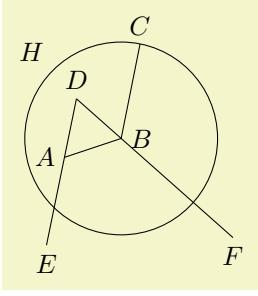
Finally, it is not necessary to explicitly name the point X . Rather, again like in the `xcolor` package, it is possible to chain partway modifiers:



```
\usetikzlibrary {calc}
\begin{tikzpicture}
\coordinate [label=left:$A$] (A) at (0,0);
\coordinate [label=right:$B$] (B) at (1.25,0.25);
\draw (A) -- (B);
\node [fill=red,inner sep=1pt,label=above:$D$] (D) at
    ($ (A) ! .5 ! (B) ! {\sin(60)*2} ! 90:(B) $) {};
\draw (A) -- (D) -- (B);
\end{tikzpicture}
```

4.2.2 Intersecting a Line and a Circle

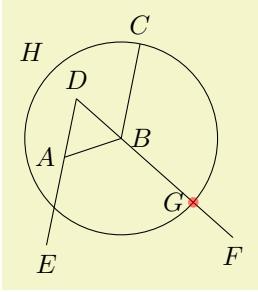
The next step in the construction is to draw a circle around B through C , which is easy enough to do using the `circle through` option. Extending the lines DA and DB can be done using partway calculations, but this time with a part value outside the range $[0,1]$:



```
\usetikzlibrary {calc,through}
\begin{tikzpicture}
\coordinate [label=left:$A$] (A) at (0,0);
\coordinate [label=right:$B$] (B) at (0.75,0.25);
\coordinate [label=above:$C$] (C) at (1,1.5);
\draw (A) -- (B) -- (C);
\coordinate [label=above:$D$] (D) at
    ($ (A) ! .5 ! (B) ! {\sin(60)*2} ! 90:(B) $) {};
\node (H) [label=135:$H$,draw,circle through=(C)] at (B) {};
\draw (D) -- ($ (D) ! 3.5 ! (B) $) coordinate [label=below:$F$] (F);
\draw (D) -- ($ (D) ! 2.5 ! (A) $) coordinate [label=below:$E$] (E);
\end{tikzpicture}
```

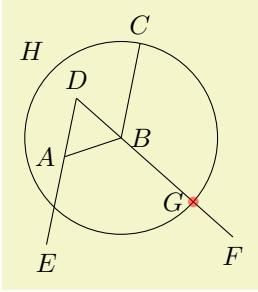
We now face the problem of finding the point G , which is the intersection of the line BF and the circle H . One way is to use yet another variant of the partway computation: Normally, a partway computation has the form $\langle p \rangle ! \langle factor \rangle ! \langle q \rangle$, resulting in the point $(1 - \langle factor \rangle)\langle p \rangle + \langle factor \rangle \langle q \rangle$. Alternatively, instead of $\langle factor \rangle$ you can also use a $\langle dimension \rangle$ between the points. In this case, you get the point that is $\langle dimension \rangle$ away from $\langle p \rangle$ on the straight line to $\langle q \rangle$.

We know that the point G is on the way from B to F . The distance is given by the radius of the circle H . Here is the code for computing H :



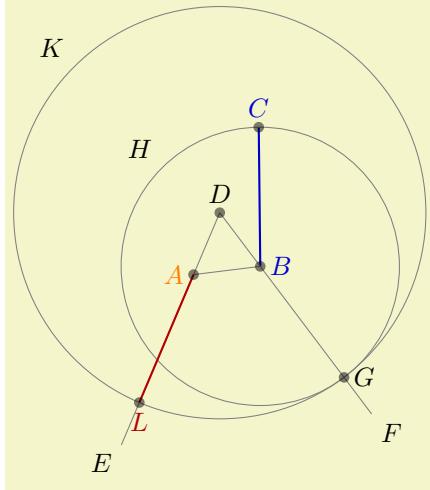
```
\usetikzlibrary {calc,through}
\node (H) [label=135:$H$,draw,circle through=(C)] at (B) {};
\path let \p1 = ($ (B) - (C) $) in
    coordinate [label=left:$G$] (G) at ($ (B) ! veclen(\x1,\y1) ! (F) $);
\fill[red,opacity=.5] (G) circle (2pt);
```

However, there is a simpler way: We can simply name the path of the circle and of the line in question and then use `name intersections` to compute the intersections.



```
\usetikzlibrary {calc,intersections,through}
\node (H) [name path=H,label=135:$H$,draw,circle through=(C)] at (B) {};
\path [name path=B--F] (B) -- (F);
\path [name intersections={of=H and B--F,by={[label=left:$G$]G}}];
\fill[red,opacity=.5] (G) circle (2pt);
```

4.2.3 The Complete Code



```
\usetikzlibrary {calc,intersections,through}
\begin{tikzpicture}[thick,help lines/.style={thin,draw=black!50}]
\def\A{\textcolor{orange}{$\textcolor{orange}{A}$}} \def\B{\textcolor{input}{$\textcolor{input}{B}$}}
\def\C{\textcolor{input}{$\textcolor{input}{C}$}} \def\D{\textcolor{D}{$\textcolor{D}{D}$}}
\def\E{\textcolor{E}{$\textcolor{E}{E}$}} \def\F{\textcolor{F}{$\textcolor{F}{F}$}}
\def\G{\textcolor{G}{$\textcolor{G}{G}$}} \def\H{\textcolor{H}{$\textcolor{H}{H}$}}
\def\K{\textcolor{K}{$\textcolor{K}{K}$}} \def\L{\textcolor{output}{$\textcolor{red}{L}$} \textcolor{output}{\colorlet{output}{red!70!black}}}

\colorlet{input}{blue!80!black} \colorlet{output}{red!70!black}

\coordinate [label=left:\textcolor{A}{A}] (A) at ($ (0,0) + .1*(rand,rand) $);
\coordinate [label=right:\textcolor{B}{B}] (B) at ($ (1,0.2) + .1*(rand,rand) $);
\coordinate [label=above:\textcolor{C}{C}] (C) at ($ (1,2) + .1*(rand,rand) $);

\draw [input] (B) -- (C);
\draw [help lines] (A) -- (B);

\coordinate [label=above:\textcolor{D}{D}] (D) at ($ (A)! .5! (B) ! {\sin(60)*2} ! 90:(B) $);
\draw [help lines] (D) -- ($ (D)!3.75!(A) $) coordinate [label=-135:\textcolor{E}{E}] (E);
\draw [help lines] (D) -- ($ (D)!3.75!(B) $) coordinate [label=-45:\textcolor{F}{F}] (F);

\node (\textcolor{H}{H}) at (B) [name path=H,help lines,circle through=(C),draw,label=135:\textcolor{H}{H}] {};
\path [name path=B--F] (B) -- (F);
\path [name intersections={of=H and B--F,by={[label=right:\textcolor{G}{G}]G}}];

\node (\textcolor{K}{K}) at (D) [name path=K,help lines,circle through=(G),draw,label=135:\textcolor{K}{K}] {};
\path [name path=A--E] (A) -- (E);
\path [name intersections={of=K and A--E,by={[label=below:\textcolor{L}{L}]L}}];

\draw [output] (A) -- (L);

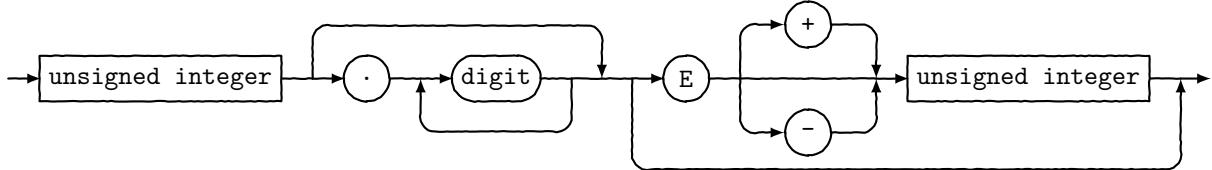
\foreach \point in {A,B,C,D,G,L}
\fill [black,opacity=.5] (\point) circle (2pt);

% \node ...
\end{tikzpicture}
```

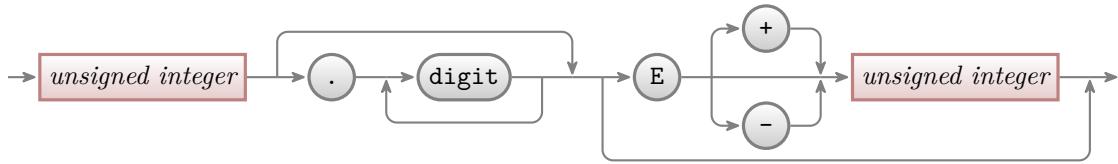
5 Tutorial: Diagrams as Simple Graphs

In this tutorial we have a look at how graphs and matrices can be used to typeset a diagram.

Ilka, who just got tenure for her professorship on Old and Lovable Programming Languages, has recently dug up a technical report entitled *The Programming Language Pascal* in the dusty cellar of the library of her university. Having been created in the good old times using pens and rules, it looks like this²:



For her next lecture, Ilka decides to redo this diagram, but this time perhaps a bit cleaner and perhaps also bit “cooler”.



Having read the previous tutorials, Ilka knows already how to set up the environment for her diagram, namely using a `tikzpicture` environment. She wonders which libraries she will need. She decides that she will postpone the decision and add the necessary libraries as needed as she constructs the picture.

5.1 Styling the Nodes

The bulk of this tutorial will be about arranging the nodes and connecting them using chains, but let us start with setting up styles for the nodes.

There are two kinds of nodes in the diagram, namely what theoreticians like to call *terminals* and *nonterminals*. For the terminals, Ilka decides to use a black color, which visually shows that “nothing needs to be done about them”. The nonterminals, which still need to be “processed” further, get a bit of red mixed in.

Ilka starts with the simpler nonterminals, as there are no rounded corners involved. Naturally, she sets up a style:

```

\usepackage{tikz}
\usetikzlibrary {positioning}

\begin{tikzpicture}[%
    nonterminal/.style={%
        % The shape:
        rectangle,
        % The size:
        minimum size=6mm,
        % The border:
        very thick,
        draw=red!50!black!50,           % 50% red and 50% black,
                                         % and that mixed with 50% white
        % The filling:
        top color=white,               % a shading that is white at the top...
        bottom color=red!50!black!20,   % and something else at the bottom
        % Font
        font=\itshape
    }
]
\node [nonterminal] {unsigned integer};
\end{tikzpicture}

```

Ilka is pretty proud of the use of the `minimum size` option: As the name suggests, this option ensures that the node is at least 6mm by 6mm, but it will expand in size as necessary to accommodate longer text. By giving this option to all nodes, they will all have the same height of 6mm.

Styling the terminals is a bit more difficult because of the round corners. Ilka has several options how she can achieve them. One way is to use the `rounded corners` option. It gets a dimension as parameter

²The shown diagram was not scanned, but rather typeset using TikZ. The jittering lines were created using the `randomsteps` decoration.

and causes all corners to be replaced by little arcs with the given dimension as radius. By setting the radius to 3mm, she will get exactly what she needs: circles, when the shapes are, indeed, exactly 6mm by 6mm and otherwise half circles on the sides:



```
\usetikzlibrary {positioning}
\begin{tikzpicture}[node distance=5mm,
    terminal/.style={%
        % The shape:
        rectangle,minimum size=6mm,rounded corners=3mm,
        % The rest
        very thick,draw=black!50,
        top color=white,bottom color=black!20,
        font=\ttfamily}]
\node (dot) [terminal]           {.};
\node (digit) [terminal,right=of dot] {digit};
\node (E) [terminal,right=of digit] {E};
\end{tikzpicture}
```

Another possibility is to use a shape that is specially made for typesetting rectangles with arcs on the sides (she has to use the `shapes.misic` library to use it). This shape gives Ilka much more control over the appearance. For instance, she could have an arc only on the left side, but she will not need this.



```
\usetikzlibrary {positioning,shapes.misic}
\begin{tikzpicture}[node distance=5mm,
    terminal/.style={%
        % The shape:
        rounded rectangle,
        minimum size=6mm,
        % The rest
        very thick,draw=black!50,
        top color=white,bottom color=black!20,
        font=\ttfamily}]
\node (dot) [terminal]           {.};
\node (digit) [terminal,right=of dot] {digit};
\node (E) [terminal,right=of digit] {E};
\end{tikzpicture}
```

At this point, she notices a problem. The baseline of the text in the nodes is not aligned:



```
\usetikzlibrary {calc,positioning,shapes.misic}
\begin{tikzpicture}[node distance=5mm]
\node (dot) [terminal]           {.};
\node (digit) [terminal,right=of dot] {digit};
\node (E) [terminal,right=of digit] {E};

\draw [help lines] let \p1 = (dot.base),
                  \p2 = (digit.base),
                  \p3 = (E.base)
            in (-.5,\y1) -- (3.5,\y1)
                (-.5,\y2) -- (3.5,\y2)
                (-.5,\y3) -- (3.5,\y3);
\end{tikzpicture}
```

(Ilka has moved the style definition to the preamble by saying `\tikzset{terminal/.style=...}`, so that she can use it in all pictures.)

For the `digit` and the `E` the difference in the baselines is almost imperceptible, but for the dot the problem is quite severe: It looks more like a multiplication dot than a period.

Ilka toys with the idea of using the `base right=of...` option rather than `right=of...` to align the nodes in such a way that the baselines are all on the same line (the `base right` option places a node right of something so that the baseline is right of the baseline of the other object). However, this does not have the desired effect:



```
\usetikzlibrary {positioning,shapes.misic}
\begin{tikzpicture}[node distance=5mm]
\node (dot) [terminal]           {.};
\node (digit) [terminal,base right=of dot] {digit};
\node (E) [terminal,base right=of digit] {E};
\end{tikzpicture}
```

The nodes suddenly “dance around”! There is no hope of changing the position of text inside a node using anchors. Instead, Ilka must use a trick: The problem of mismatching baselines is caused by the fact

that `.` and `digit` and `E` all have different heights and depth. If they all had the same, they would all be positioned vertically in the same manner. So, all Ilka needs to do is to use the `text height` and `text depth` options to explicitly specify a height and depth for the nodes.



```
\usetikzlibrary {positioning,shapes.misc}
\begin{tikzpicture} [node distance=5mm,
text height=1.5ex, text depth=.25ex]
\node (dot) [terminal] {.};
\node (digit) [terminal,right=of dot] {digit};
\node (E) [terminal,right=of digit] {E};
\end{tikzpicture}
```

5.2 Aligning the Nodes Using Positioning Options

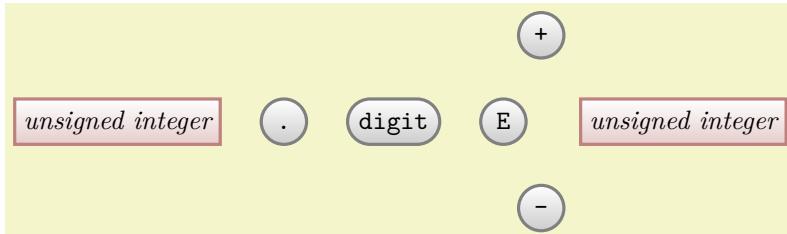
Ilka now has the “styling” of the nodes ready. The next problem is to place them in the right places. There are several ways to do this. The most straightforward is to simply explicitly place the nodes at certain coordinates “calculated by hand”. For very simple graphics this is perfectly alright, but it has several disadvantages:

1. For more difficult graphics, the calculation may become complicated.
2. Changing the text of the nodes may make it necessary to recalculate the coordinates.
3. The source code of the graphic is not very clear since the relationships between the positions of the nodes are not made explicit.

For these reasons, Ilka decides to try out different ways of arranging the nodes on the page.

The first method is the use of *positioning options*. To use them, you need to load the `positioning` library. This gives you access to advanced implementations of options like `above` or `left`, since you can now say `above=of some node` in order to place a node above of `some node`, with the borders separated by `node distance`.

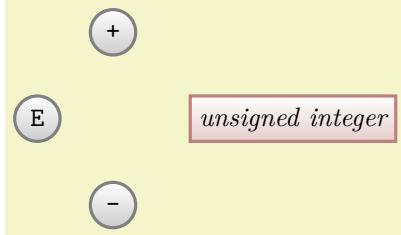
Ilka can use this to draw the place the nodes in a long row:



```
\usetikzlibrary {positioning,shapes.misc}
\begin{tikzpicture} [node distance=5mm and 5mm]
\node (ui1) [nonterminal] {unsigned integer};
\node (dot) [terminal,right=of ui1] {.};
\node (digit) [terminal,right=of dot] {digit};
\node (E) [terminal,right=of digit] {E};
\node (plus) [terminal,above right=of E] {+};
\node (minus) [terminal,below right=of E] {-};
\node (ui2) [nonterminal,below right=of plus] {unsigned integer};
\end{tikzpicture}
```

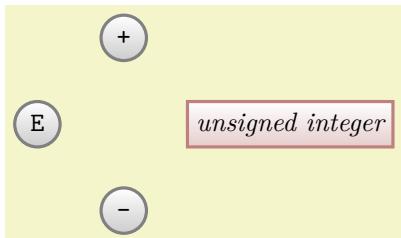
For the plus and minus nodes, Ilka is a bit startled by their placements. Shouldn’t they be more to the right? The reason they are placed in that manner is the following: The `north east` anchor of the `E` node lies at the “upper start of the right arc”, which, a bit unfortunately in this case, happens to be the top of the node. Likewise, the `south west` anchor of the `+` node is actually at its bottom and, indeed, the horizontal and vertical distances between the top of the `E` node and the bottom of the `+` node are both 5mm.

There are several ways of fixing this problem. The easiest way is to simply add a little bit of horizontal shift by hand:



```
\usetikzlibrary {positioning,shapes.misc}
\begin{tikzpicture} [node distance=5mm and 5mm]
\node (E) [terminal] {E};
\node (plus) [terminal,above right=of E,xshift=5mm] {+};
\node (minus) [terminal,below right=of E,xshift=5mm] {-};
\node (ui2) [nonterminal,below right=of plus,xshift=5mm] {unsigned integer};
\end{tikzpicture}
```

A second way is to revert back to the idea of using a normal rectangle for the terminals, but with rounded corners. Since corner rounding does not affect anchors, she gets the following result:



```
\usetikzlibrary {positioning,shapes.misc}
\begin{tikzpicture} [node distance=5mm and 5mm,terminal/.append style={rectangle,rounded corners=3mm}]
\node (E) [terminal] {E};
\node (plus) [terminal,above right=of E] {+};
\node (minus) [terminal,below right=of E] {-};
\node (ui2) [nonterminal,below right=of plus] {unsigned integer};
\end{tikzpicture}
```

A third way is to use matrices, which we will do later.

Now that the nodes have been placed, Ilka needs to add connections. Here, some connections are more difficult than others. Consider for instance the “repeat” line around the `digit`. One way of describing this line is to say “it starts a little to the right of `digit` than goes down and then goes to the left and finally ends at a point a little to the left of `digit`”. Ilka can put this into code as follows:



```
\usetikzlibrary {calc,positioning,shapes.misc}
\begin{tikzpicture} [node distance=5mm and 5mm]
\node (dot) [terminal] {.};
\node (digit) [terminal,right=of dot] {digit};
\node (E) [terminal,right=of digit] {E};

\path (dot) edge[->] (digit) % simple edges
(digit) edge[->] (E);

\draw [->]
% start right of digit.east, that is, at the point that is the
% linear combination of digit.east and the vector (2mm,0pt). We
% use the ($ ... $) notation for computing linear combinations
($ (digit.east) + (2mm,0) $)
% Now go down
-- +(0,-.5)
% And back to the left of digit.west
-| ($ (digit.west) - (2mm,0) $);
\end{tikzpicture}
```

Since Ilka needs this “go up/down then horizontally and then up/down to a target” several times, it seems sensible to define a special `to-path` for this. Whenever the `edge` command is used, it simply adds the current value of `to path` to the path. So, Ilka can set up a style that contains the correct path:



```
\usetikzlibrary {calc,positioning,shapes.misc}
\begin{tikzpicture}[node distance=5mm and 5mm,
  skip loop/.style={to path={-- ++(0,-.5) -| (\tikztotarget)}}
]
\node (dot) [terminal] {.};
\node (digit) [terminal,right=of dot] {digit};
\node (E) [terminal,right=of digit] {E};

\path (dot) edge[->] (digit) % simple edges
(digit) edge[->] (E)
($ (digit.east) + (2mm,0) $)
edge[->,skip loop] ($ (digit.west) - (2mm,0) $);
\end{tikzpicture}
```

Ilka can even go a step further and make her `skip loop` style parameterized. For this, the skip loop's vertical offset is passed as parameter #1. Also, in the following code Ilka specifies the start and targets differently, namely as the positions that are “in the middle between the nodes”.



```
\usetikzlibrary {calc,positioning,shapes.misc}
\begin{tikzpicture}[node distance=5mm and 5mm,
  skip loop/.style={to path={-- ++(0,#1) -| (\tikztotarget)}}
]
\node (dot) [terminal] {.};
\node (digit) [terminal,right=of dot] {digit};
\node (E) [terminal,right=of digit] {E};

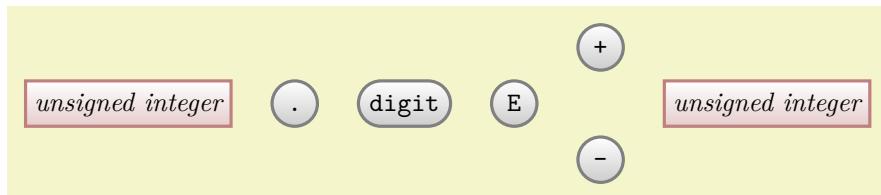
\path (dot) edge[->] (digit) % simple edges
(digit) edge[->] (E)
($ (digit.east)! .5! (E.west) $)
edge[->,skip loop=-5mm] ($ (digit.west)! .5! (dot.east) $);
\end{tikzpicture}
```

5.3 Aligning the Nodes Using Matrices

Ilka is still bothered a bit by the placement of the plus and minus nodes. Somehow, having to add an explicit `xshift` seems too much like cheating.

A perhaps better way of positioning the nodes is to use a *matrix*. In TikZ matrices can be used to align quite arbitrary graphical objects in rows and columns. The syntax is very similar to the use of arrays and tables in TeX (indeed, internally TeX tables are used, but a lot of stuff is going on additionally).

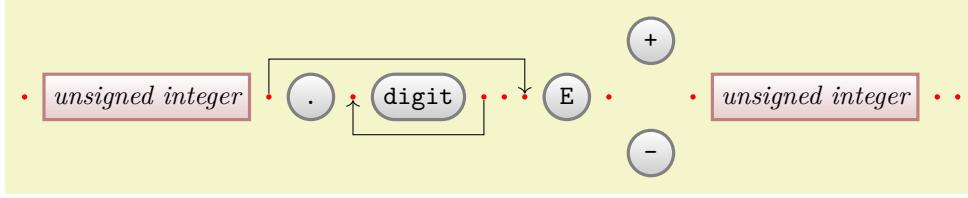
In Ilka's graphic, there will be three rows: One row containing only the plus node, one row containing the main nodes and one row containing only the minus node.



```
\usetikzlibrary {shapes.misc}
\begin{tikzpicture}
\matrix[row sep=1mm,column sep=5mm] {
% First row:
&&&\node [terminal] {+}; & \\
% Second row:
\node [nonterminal] {unsigned integer}; &
\node [terminal] {.}; & & & \\
\node [terminal] {digit}; & & & & \\
\node [terminal] {E}; & & & & \\
&&&\node [terminal] {-}; & \\
% Third row:
&&&\node [terminal] {-}; & \\
};
\end{tikzpicture}
```

That was easy! By toying around with the row and columns separations, Ilka can achieve all sorts of pleasing arrangements of the nodes.

Ilka now faces the same connecting problem as before. This time, she has an idea: She adds small nodes (they will be turned into coordinates later on and be invisible) at all the places where she would like connections to start and end.



```
\usetikzlibrary {shapes.misc}
\begin{tikzpicture}[point/.style={circle,inner sep=0pt,minimum size=2pt,fill=red},
    skip loop/.style={to path={-- ++(0,#1) -/ (\tikztotarget)}}
\matrix [row sep=1mm,column sep=2mm] {
    % First row:
    & & & & & & \node (plus) [terminal] {+}; \\
    % Second row:
    \node (p1) [point] {}; & \node (ui1) [nonterminal] {unsigned integer}; & & & & & \\
    \node (p2) [point] {}; & \node (dot) [terminal] {.}; & & & & & \\
    \node (p3) [point] {}; & \node (digit) [terminal] {digit}; & & & & & \\
    \node (p4) [point] {}; & \node (p5) [point] {}; & & & & & \\
    \node (p6) [point] {}; & \node (e) [terminal] {E}; & & & & & \\
    \node (p7) [point] {}; & \node (ui2) [nonterminal] {unsigned integer}; & & & & & \\
    \node (p8) [point] {}; & \node (p9) [point] {}; & & & & & \\
    \node (p9) [point] {}; & \node (p10) [point] {}; & & & & & \\
    % Third row:
    & & & & & & \node (minus) [terminal] {-}; \\
};

\path (p4) edge [->,skip loop=-5mm] (p3)
      (p2) edge [->,skip loop=5mm] (p6);
\end{tikzpicture}
```

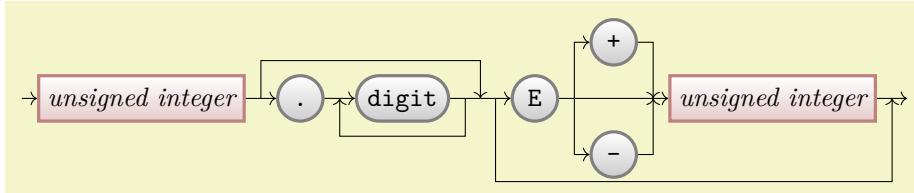
Now, it's only a small step to add all the missing edges.

5.4 The Diagram as a Graph

Matrices allow Ilka to align the nodes nicely, but the connections are not quite perfect. The problem is that the code does not really reflect the paths that underlie the diagram. For this, it seems natural enough to Ilka to use the `graphs` library since, after all, connecting nodes by edges is exactly what happens in a graph. The `graphs` library can both be used to connect nodes that have already been created, but it can also be used to create nodes “on the fly” and these processes can also be mixed.

5.4.1 Connecting Already Positioned Nodes

Ilka has already a fine method for positioning her nodes (using a `matrix`), so all that she needs is an easy way of specifying the edges. For this, she uses the `\graph` command (which is actually just a shorthand for `\path graph`). It allows her to write down edges between them in a simple way (the macro `\matrixcontent` contains exactly the matrix content from the previous example; no need to repeat it here):

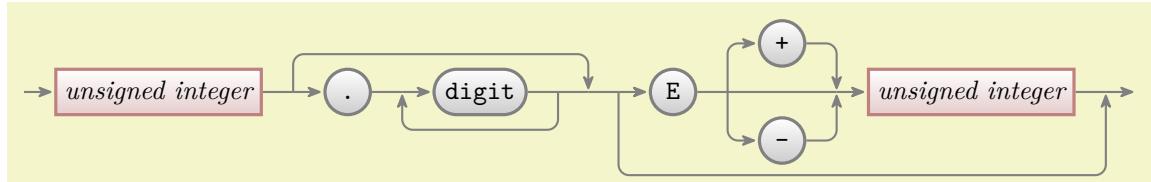


```
\usetikzlibrary {graphs,shapes.mis}
\begin{tikzpicture}[skip loop/.style={to path={-- ++(0,#1) -/ (\tikztotarget)}},
    hv path/.style={to path={-/ (\tikztotarget)}},
    vh path/.style={to path={/- (\tikztotarget)}}
\matrix [row sep=1mm,column sep=2mm] { \matrixcontent };
\graph {
    (p1) --> (ui1) -- (p2) --> (dot) -- (p3) --> (digit) -- (p4)
    -- (p5) -- (p6) --> (e) -- (p7) -- (p8) --> (ui2) -- (p9) --> (p10);
    (p4) ->[skip loop=-5mm] (p3);
    (p2) ->[skip loop=5mm] (p5);
    (p6) ->[skip loop=-11mm] (p9);
    (p7) ->[vh path] (plus) -> [hv path] (p8);
    (p7) ->[vh path] (minus) -> [hv path] (p8);
};
\end{tikzpicture}
```

This is already pretty near to the desired result, just a few “finishing touches” are needed to style the edges more nicely.

However, Ilka does not have the feeling that the `graph` command is all that hot in the example. It certainly does cut down on the number of characters she has to write, but the overall graph structure is not that much clear – it is still mainly a list of paths through the graph. It would be nice to specify that, say, there the path from (p7) sort of splits to (plus) and (minus) and then merges once more at (p8). Also, all these parentheses are bit hard to type.

It turns out that edges from a node to a whole group of nodes are quite easy to specify, as shown in the next example. Additionally, by using the `use existing nodes` option, Ilka can also leave out all the parentheses (again, some options have been moved outside to keep the examples shorter):



```
\usetikzlibrary {arrows.meta,graphs,shapes.mis}
\begin{tikzpicture} [Stealth[round], thick, black!50, text=black,
    every new ->/.style={shorten >=1pt},
    graphs/every graph/.style={edges=rounded corners}]
\matrix [column sep=4mm] { \matrixcontent };

\graph [use existing nodes] {
    p1 -> ui1 -- p2 -> dot -- p3 -> digit -- p4 -- p5 -- p6 -> e -- p7 -- p8 -> ui2 -- p9 -> p10;
    p4 ->[skip loop=-5mm] p3;
    p2 ->[skip loop=5mm] p5;
    p6 ->[skip loop=-11mm] p9;
    p7 ->[vh path] { plus, minus } -> [hv path] p8;
};
\end{tikzpicture}
```

5.4.2 Creating Nodes Using the Graph Command

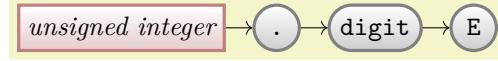
Ilka has heard that the `graph` command is also supposed to make it easy to create nodes, not only to connect them. This is, indeed, correct: When the `use existing nodes` option is not used and when a node name is not surrounded by parentheses, then TikZ will actually create a node whose name and text is the node name:

`unsigned integer → d —————→ digit —————→ E`

```
\usetikzlibrary {graphs}
\tikz \graph [grow right=2cm] { unsigned integer -> d -> digit -> E };
```

Not quite perfect, but we are getting somewhere. First, let us change the positioning algorithm by saying `grow right sep`, which causes new nodes to be placed to the right of the previous nodes with a certain fixed separation (`1em` by default). Second, we add some options to make the node “look nice”. Third,

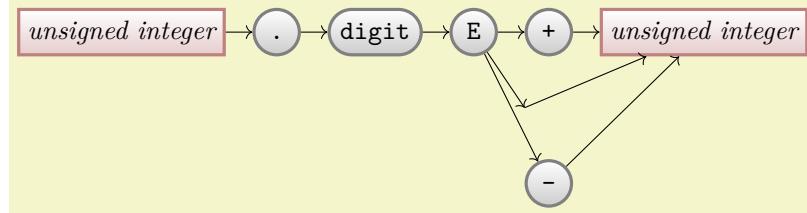
note the funny `d` node above: Ilka tried writing just `.` there first, but got some error messages. The reason is that a node cannot be called `.` in TikZ, so she had to choose a different name – which is not good, since she wants a dot to be shown! The trick is to put the dot in quotation marks, this allows you to use “quite arbitrary text” as a node name:



```
\usetikzlibrary {graphs,shapes.misc}
\begin{tikzpicture}
\graph [grow right sep] {
    unsigned integer[nonterminal] -> ". "[terminal] -> digit[terminal] -> E[terminal]
};

```

Now comes the fork to the plus and minus signs. Here, Ilka can use the grouping mechanism of the `graph` command to create a split:



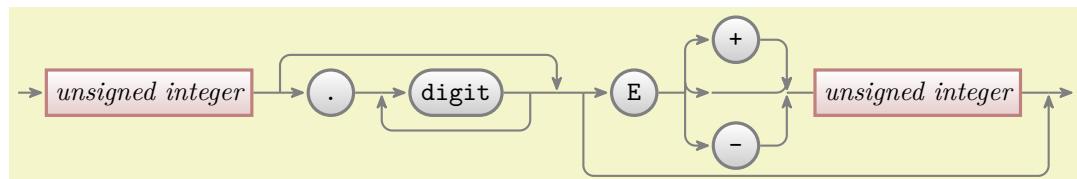
```
\usetikzlibrary {graphs,shapes.misc}
\begin{tikzpicture}
\graph [grow right sep] {
    unsigned integer [nonterminal] ->
    ". " [terminal] ->
    digit [terminal] ->
    E [terminal] ->
    "+" [terminal],
    "" [coordinate], 
    "-" [terminal]
} ->
ui2/unsigned integer [nonterminal];

```

Let us see, what is happening here. We want two `unsigned integer` nodes, but if we just were to use this text twice, then TikZ would have noticed that the same name was used already in the current graph and, being smart (actually too smart in this case), would have created an edge back to the already-created node. Thus, a fresh name is needed here. However, Ilka also cannot just write `unsigned integer2`, because she wants the original text to be shown, after all! The trick is to use a slash inside the node name: In order to “render” the node, the text following the slash is used instead of the node name, which is the text before the slash. Alternatively, the `as` option can be used, which also allows you to specify how a node should be rendered.

It turns out that Ilka does not need to invent a name like `ui2` for a node that she will not reference again anyway. In this case, she can just leave out the name (write nothing before `/`), which always stands for a “fresh, anonymous” node name.

Next, Ilka needs to add some coordinates in between of some nodes where the back-loops should got and she needs to shift the nodes a bit:



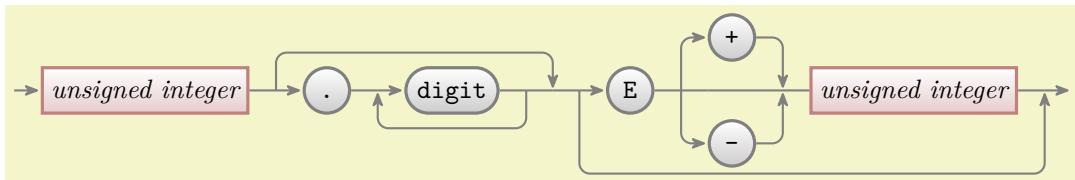
```

\usetikzlibrary {arrows.meta,graphs,shapes.mis}
\begin{tikzpicture} [->={Stealth[round]}, thick, black!50, text=black,
    every new ->/ .style={shorten >=1pt},
    graphs/every graph/.style={edges=rounded corners}]
\graph [grow right sep, branch down=7mm] {
    / [coordinate] -->
    unsigned integer [nonterminal] --
    p1 [coordinate] -->
    ". ." [terminal] --
    p2 [coordinate] -->
    digit [terminal] --
    p3 [coordinate] --
    p4 [coordinate] --
    p5 [coordinate] -->
    E [terminal] --
    q1 [coordinate] -->[vh path]
    { [nodes={yshift=7mm}]
        "+" [terminal],
        q2/ [coordinate],
        "-_" [terminal]
    } --> [hv path]
    q3 [coordinate] --
    /unsigned integer [nonterminal] --
    p6 [coordinate] -->
    / [coordinate];
    p1 -->[skip loop=5mm] p4;
    p3 -->[skip loop=-5mm] p2;
    p5 -->[skip loop=-11mm] p6;
};
\end{tikzpicture}

```

All that remains to be done is to somehow get rid of the strange curves between the `E` and the `unsigned integer`. They are caused by TikZ's attempt at creating an edge that first goes vertical and then horizontal but is actually just horizontal. Additionally, the edge should not really be pointed; but it seems difficult to get rid of this since the *other* edges from `q1`, namely to `plus` and `minus` should be pointed.

It turns out that there is a nice way of solving this problem: You can specify that a graph is `simple`. This means that there can be at most one edge between any two nodes. Now, if you specify an edge twice, the options of the second specification "win". Thus, by adding two more lines that "correct" these edges, we get the final diagram with its complete code:



```

\usetikzlibrary {arrows.meta,graphs,shapes.mis}
\tikz [>={Stealth[round]}, black!50, text=black, thick,
every new -/.style      = {shorten >=1pt},
graphs/every graph/.style = {edges=rounded corners},
skip loop/.style         = {to path={-- ++(0,#1) -- (\tikztotarget)}},
hv path/.style            = {to path={/- (\tikztotarget)}},
vh path/.style            = {to path={/- (\tikztotarget)}},
nonterminal/.style        = {
    rectangle, minimum size=6mm, very thick, draw=red!50!black!50, top color=white,
    bottom color=red!50!black!20, font=\itshape, text height=1.5ex, text depth=.25ex},
terminal/.style           = {
    rounded rectangle, minimum size=6mm, very thick, draw=black!50, top color=white,
    bottom color=black!20, font=\ttfamily, text height=1.5ex, text depth=.25ex},
shape                     = coordinate
]
\graph [grow right sep, branch down=7mm, simple] {
/ -> unsigned integer [nonterminal] -- p1 -> ." [terminal] -- p2 -> digit [terminal] --
p3 -- p4 -- p5 -> E [terminal] -- q1 -> [vh path]
{[nodes={yshift=7mm}]
 "+" [terminal], q2, "-" [terminal]
} -> [hv path]
q3 -- /unsigned integer [nonterminal] -- p6 -> /;

p1 -> [skip loop=5mm] p4;
p3 -> [skip loop=-5mm] p2;
p5 -> [skip loop=-11mm] p6;

q1 -- q2 -- q3; % make these edges plain
};

```

6 Tutorial: A Lecture Map for Johannes

In this tutorial we explore the tree and mind map mechanisms of TikZ.

Johannes is quite excited: For the first time he will be teaching a course all by himself during the upcoming semester! Unfortunately, the course is not on his favorite subject, which is of course Theoretical Immunology, but on Complexity Theory, but as a young academic Johannes is not likely to complain too loudly. In order to help the students get a general overview of what is going to happen during the course as a whole, he intends to draw some kind of tree or graph containing the basic concepts. He got this idea from his old professor who seems to be using these “lecture maps” with some success. Independently of the success of these maps, Johannes thinks they look quite neat.

6.1 Problem Statement

Johannes wishes to create a lecture map with the following features:

1. It should contain a tree or graph depicting the main concepts.
2. It should somehow visualize the different lectures that will be taught. Note that the lectures are not necessarily the same as the concepts since the graph may contain more concepts than will be addressed in lectures and some concepts may be addressed during more than one lecture.
3. The map should also contain a calendar showing when the individual lectures will be given.
4. The aesthetical reasons, the whole map should have a visually nice and information-rich background.

As always, Johannes will have to include the right libraries and set up the environment. Johannes is going to use the `mindmap` library and since he wishes to show a calendar, he will also need the `calendar` library. In order to put something on a background layer, it seems like a good idea to also include the `backgrounds` library.

6.2 Introduction to Trees

The first choice Johannes must make is whether he will organize the concepts as a tree, with root concepts and concept branches and leaf concepts, or as a general graph. The tree implicitly organizes the concepts, while a graph is more flexible. Johannes decides to compromise: Basically, the concepts will be organized as a tree. However, he will selectively add connections between concepts that are related, but which appear on different levels or branches of the tree.

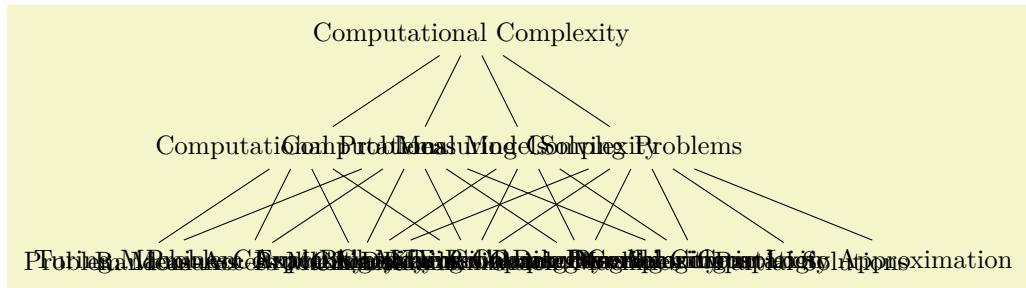
Johannes starts with a tree-like list of concepts that he feels are important in Computational Complexity:

- Computational Problems
 - Problem Measures
 - Problem Aspects
 - Problem Domains
 - Key Problems
- Computational Models
 - Turing Machines
 - Random-Access Machines
 - Circuits
 - Binary Decision Diagrams
 - Oracle Machines
 - Programming in Logic
- Measuring Complexity
 - Complexity Measures
 - Classifying Complexity
 - Comparing Complexity
 - Describing Complexity
- Solving Problems

- Exact Algorithms
 - Randomization
 - Fixed-Parameter Algorithms
 - Parallel Computation
 - Partial Solutions
 - Approximation

Johannes will surely need to modify this list later on, but it looks good as a first approximation. He will also need to add a number of subtopics (like *lots* of complexity classes under the topic “classifying complexity”), but he will do this as he constructs the map.

Turning the list of topics into a TikZ-tree is easy, in principle. The basic idea is that a node can have *children*, which in turn can have children of their own, and so on. To add a child to a node, Johannes can simply write `child {<node>}` right after a node. The `<node>` should, in turn, be the code for creating a node. To add another node, Johannes can use `child` once more, and so on. Johannes is eager to try out this construct and writes down the following:



```

\tikz
\node {Computational Complexity} % root
  child { node {Computational Problems}
    child { node {Problem Measures} }
    child { node {Problem Aspects} }
    child { node {Problem Domains} }
    child { node {Key Problems} }
  }
  child { node {Computational Models}
    child { node {Turing Machines} }
    child { node {Random-Access Machines} }
    child { node {Circuits} }
    child { node {Binary Decision Diagrams} }
    child { node {Oracle Machines} }
    child { node {Programming in Logic} }
  }
  child { node {Measuring Complexity}
    child { node {Complexity Measures} }
    child { node {Classifying Complexity} }
    child { node {Comparing Complexity} }
    child { node {Describing Complexity} }
  }
  child { node {Solving Problems}
    child { node {Exact Algorithms} }
    child { node {Randomization} }
    child { node {Fixed-Parameter Algorithms} }
    child { node {Parallel Computation} }
    child { node {Partial Solutions} }
    child { node {Approximation} }
  }
;

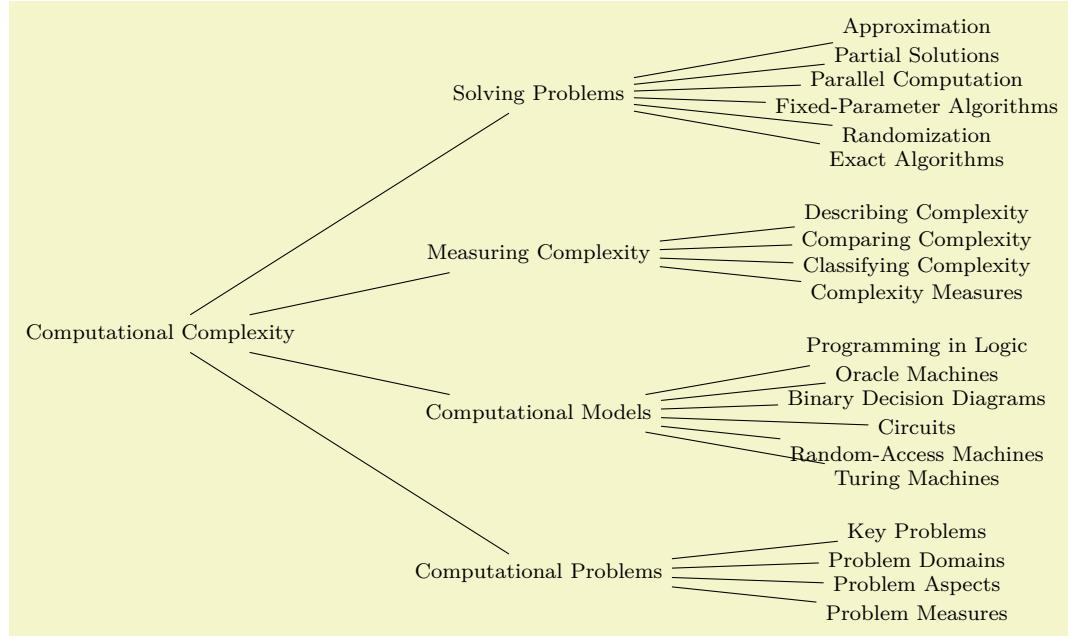
```

Well, that did not quite work out as expected (although, what, exactly, did one expect?). There are two problems:

1. The overlap of the nodes is due to the fact that TikZ is not particularly smart when it comes to placing child nodes. Even though it is possible to configure TikZ to use rather clever placement methods, TikZ has no way of taking the actual size of the child nodes into account. This may seem strange but the reason is that the child nodes are rendered and placed one at a time, so the size of the last node is not known when the first node is being processed. In essence, you have to specify appropriate level and sibling node spacings “by hand”.

2. The standard computer-science-top-down rendering of a tree is rather ill-suited to visualizing the concepts. It would be better to either rotate the map by ninety degrees or, even better, to use some sort of circular arrangement.

Johannes redraws the tree, but this time with some more appropriate options set, which he found more or less by trial-and-error:



```

\usetikzlibrary {trees}
\tikz [font=\footnotesize,
      grow=right, level 1/.style={sibling distance=6em},
      level 2/.style={sibling distance=1em}, level distance=5cm]
\node {Computational Complexity} % root
  child { node {Computational Problems}
    child { node {Problem Measures} }
    child { node {Problem Aspects} }
    ... % as before
  }

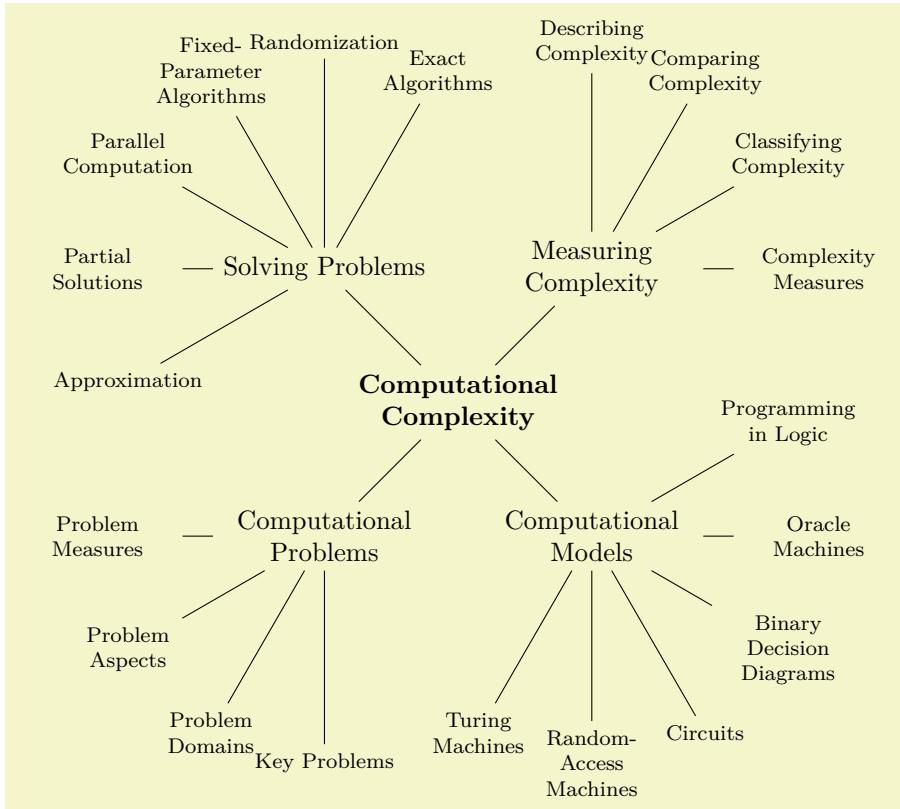
```

Still not quite what Johannes had in mind, but he is getting somewhere.

For configuring the tree, two parameters are of particular importance: The `level distance` tells TikZ the distance between (the centers of) the nodes on adjacent levels or layers of a tree. The `sibling distance` is, as the name suggests, the distance between (the centers of) siblings of the tree.

You can globally set these parameters for a tree by simply setting them somewhere before the tree starts, but you will typically wish them to be different for different levels of the tree. In this case, you should set styles like `level 1` or `level 2`. For the first level of the tree, the `level 1` style is used, for the second level the `level 2` style, and so on. You can also set the sibling and level distances only for certain nodes by passing these options to the `child` command as options. (Note that the options of a `node` command are local to the node and have no effect on the children. Also note that it is possible to specify options that do have an effect on the children. Finally note that specifying options for children “at the right place” is an arcane art and you should peruse Section 21.4 on a rainy Sunday afternoon, if you are really interested.)

The `grow` key is used to configure the direction in which a tree grows. You can change growth direction “in the middle of a tree” simply by changing this key for a single child or a whole level. By including the `trees` library you also get access to additional growth strategies such as a “circular” growth:



```

\usetikzlibrary {trees}
\tikz [text width=2.7cm, align=flush center,
      grow cyclic,
      level 1/.style=[level distance=2.5cm,sibling angle=90],
      level 2/.style=[text width=2cm, font=\footnotesize, level distance=3cm,sibling angle=30]]
\node[font=\bfseries] {Computational Complexity} % root
    child { node {Computational Problems}}
    child { node {Problem Measures} }
    child { node {Problem Aspects} }
    ...

```

Johannes is pleased to learn that he can access and manipulate the nodes of the tree like any normal node. In particular, he can name them using the `name=` option or the `(name)` notation and he can use any available shape or style for the trees nodes. He can connect trees later on using the normal `\draw (some node) --(another node);` syntax. In essence, the `child` command just computes an appropriate position for a node and adds a line from the child to the parent node.

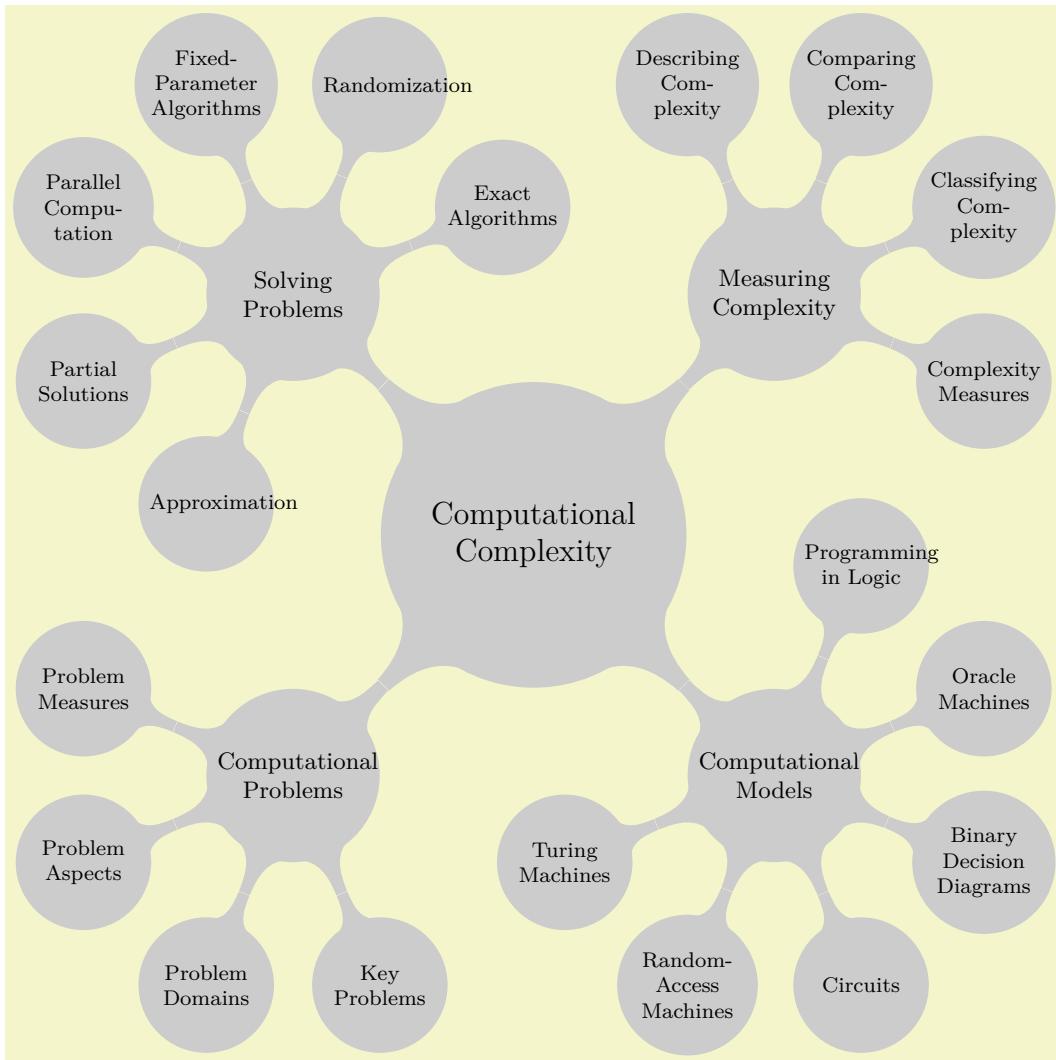
6.3 Creating the Lecture Map

Johannes now has a first possible layout for his lecture map. The next step is to make it “look nicer”. For this, the `mindmap` library is helpful since it makes a number of styles available that will make a tree look like a nice “mind map” or “concept map”.

The first step is to include the `mindmap` library, which Johannes already did. Next, he must add one of the following options to a scope that will contain the lecture map: `mindmap` or `large mindmap` or `huge mindmap`. These options all have the same effect, except that for a `large mindmap` the predefined font size and node sizes are somewhat larger than for a standard `mindmap` and for a `huge mindmap` they are even larger. So, a `large mindmap` does not necessarily need to have a lot of concepts, but it will need a lot of paper.

The second step is to add the `concept` option to every node that will, indeed, be a concept of the mindmap. The idea is that some nodes of a tree will be real concepts, while other nodes might just be “simple children”. Typically, this is not the case, so you might consider saying `every node/.style=concept`.

The third step is to set up the `sibling angle` (rather than a sibling distance) to specify the angle between sibling concepts.

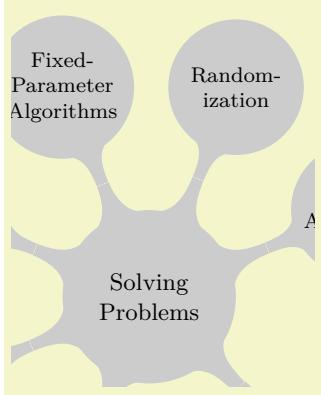


```

\usetikzlibrary {mindmap}
\tikz [mindmap, every node/.style=concept, concept color=black!20,
      grow cyclic,
      level 1/.append style={level distance=4.5cm,sibling angle=90},
      level 2/.append style={level distance=3cm,sibling angle=45}]
\node [root concept] {Computational Complexity} % root
    child { node {Computational Problems}}
        child { node {Problem Measures} }
        child { node {Problem Aspects} }
        ...
    ... % as before

```

When Johannes typesets the above map, \TeX (rightfully) starts complaining about several overfull boxes and, indeed, words like “Randomization” stretch out beyond the circle of the concept. This seems a bit mysterious at first sight: Why does \TeX not hyphenate the word? The reason is that \TeX will never hyphenate the first word of a paragraph because it starts looking for “hyphenatable” letters only after a so-called glue. In order to have \TeX hyphenate these single words, Johannes must use a bit of evil trickery: He inserts a $\backslash hskip0pt$ before the word. This has no effect except for inserting an (invisible) glue before the word and, thereby, allowing \TeX to hyphenate the first word also. Since Johannes does not want to add $\backslash hskip0pt$ inside each node, he uses the `execute at begin node` option to make TikZ insert this text with every node.



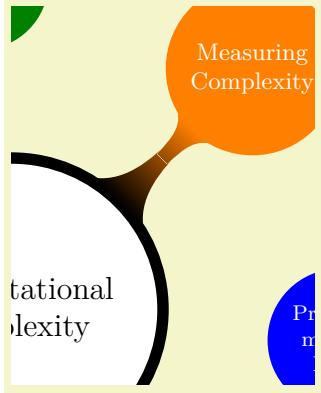
```
\usetikzlibrary {mindmap}
\begin{tikzpicture}
[mindmap,
 every node/.style={concept, execute at begin node=\hspace{0pt},
 concept color=black!20,
 grow cyclic,
 level 1/.append style={level distance=4.5cm,sibling angle=90},
 level 2/.append style={level distance=3cm,sibling angle=45}]
\clip (-1,2) rectangle ++ (-4,5);
\node [root concept] {Computational Complexity} % root
    child { node {Computational Problems}
        child { node {Problem Measures} }
        child { node {Problem Aspects} }
        ... % as before
    };
\end{tikzpicture}
```

In the above example a clipping was used to show only part of the lecture map, in order to save space. The same will be done in the following examples, we return to the complete lecture map at the end of this tutorial.

Johannes is now eager to colorize the map. The idea is to use different colors for different parts of the map. He can then, during his lectures, talk about the “green” or the “red” topics. This will make it easier for his students to locate the topic he is talking about on the map. Since “computational problems” somehow sounds “problematic”, Johannes chooses red for them, while he picks green for the “solving problems”. The topics “measuring complexity” and “computational models” get more neutral colors; Johannes picks orange and blue.

To set the colors, Johannes must use the `concept color` option, rather than just, say, `node [fill=red]`. Setting just the fill color to `red` would, indeed, make the node red, but it would *just* make the node red and not the bar connecting the concept to its parent and also not its children. By comparison, the special `concept color` option will not only set the color of the node and its children, but it will also (magically) create appropriate shadings so that the color of a parent concept smoothly changes to the color of a child concept.

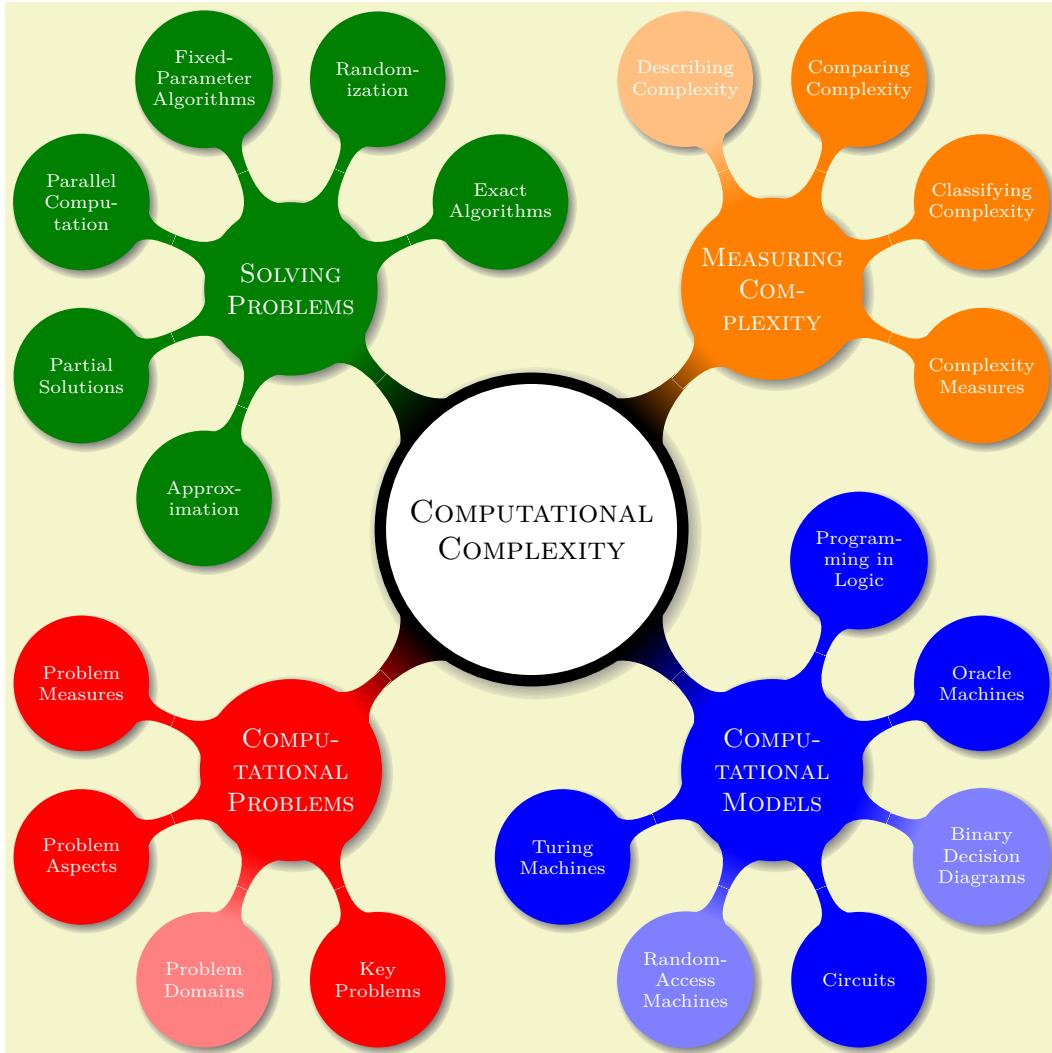
For the root concept Johannes decides to do something special: He sets the concept color to black, sets the line width to a large value, and sets the fill color to white. The effect of this is that the root concept will be encircled with a thick black line and the children are connected to the central concept via bars.



```
\usetikzlibrary {mindmap}
\begin{tikzpicture}
[mindmap,
 every node/.style={concept, execute at begin node=\hspace{0pt},
 root concept/.append style={
     concept color=black, fill=white, line width=1ex, text=black},
 text=white,
 grow cyclic,
 level 1/.append style={level distance=4.5cm,sibling angle=90},
 level 2/.append style={level distance=3cm,sibling angle=45}]
\clip (0,+1) rectangle ++ (4,5);
\node [root concept] {Computational Complexity} % root
    child [concept color=red] { node {Computational Problems}
        child { node {Problem Measures} }
        child { node {Problem Aspects} }
        ... % as before
    }
    child [concept color=blue] { node {Computational Models}
        child { node {Turing Machines} }
        ... % as before
    }
    child [concept color=orange] { node {Measuring Complexity}
        child { node {Complexity Measures} }
        ... % as before
    }
    child [concept color=green!50!black] { node {Solving Problems}
        child { node {Exact Algorithms} }
        ... % as before
    };
\end{tikzpicture}
```

Johannes adds three finishing touches: First, he changes the font of the main concepts to small caps. Second, he decides that some concepts should be “faded”, namely those that are important in principle

and belong on the map, but which he will not talk about in his lecture. To achieve this, Johannes defines four styles, one for each of the four main branches. These styles (a) set up the correct concept color for the whole branch and (b) define the `faded` style appropriately for this branch. Third, he adds a `circular drop shadow`, defined in the `shadows` library, to the concepts, just to make things look a bit more fancy.



```
\usetikzlibrary {mindmap,shadows}
\begin{tikzpicture}[mindmap]
\begin{scope}[
    every node/.style={concept, circular drop shadow, execute at begin node=\hspace*{0pt}, 
    root concept/.append style={ 
        concept color=black, fill=white, line width=1ex, text=black, font=\large\scshape}, 
    text=white, 
    computational problems/.style={concept color=red,faded/.style={concept color=red!50}}, 
    computational models/.style={concept color=blue,faded/.style={concept color=blue!50}}, 
    measuring complexity/.style={concept color=orange,faded/.style={concept color=orange!50}}, 
    solving problems/.style={concept color=green!50!black,faded/.style={concept color=green!50!black!50}}, 
    grow cyclic, 
    level 1/.append style={level distance=4.5cm,sibling angle=90,font=\scshape}, 
    level 2/.append style={level distance=3cm,sibling angle=45,font=\scriptsize}] 
\node [root concept] {Computational Complexity} % root 
    child [computational problems] { node {Computational Problems} 
        child { node {Problem Measures} } 
        child { node {Problem Aspects} } 
        child [faded] { node {Problem Domains} } 
        child { node {Key Problems} } 
    } 
    child [computational models] { node {Computational Models} 
        child { node {Turing Machines} } 
        child [faded] { node {Random-Access Machines} } 
    } 
    ...
\end{scope}
\end{tikzpicture}
```

6.4 Adding the Lecture Annotations

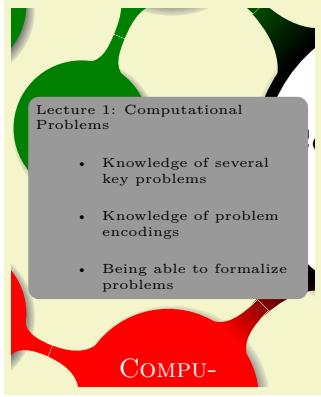
Johannes will give about a dozen lectures during the course “computational complexity”. For each lecture he has compiled a (short) list of learning targets that state what knowledge and qualifications his students should acquire during this particular lecture (note that learning targets are not the same as the contents of a lecture). For each lecture he intends to put a little rectangle on the map containing these learning targets and the name of the lecture, each time somewhere near the topic of the lecture. Such “little rectangles” are called “annotations” by the `mindmap` library.

In order to place the annotations next to the concepts, Johannes must assign names to the nodes of the concepts. He could rely on TikZ’s automatic naming of the nodes in a tree, where the children of a node named `root` are named `root-1`, `root-2`, `root-3`, and so on. However, since Johannes is not sure about the final order of the concepts in the tree, it seems better to explicitly name all concepts of the tree in the following manner:

```
\node [root concept] (Computational Complexity) {Computational Complexity} 
    child [computational problems] { node (Computational Problems) {Computational Problems} 
        child { node (Problem Measures) {Problem Measures} } 
        child { node (Problem Aspects) {Problem Aspects} } 
        child [faded] { node (Problem Domains) {Problem Domains} } 
        child { node (Key Problems) {Key Problems} } 
    } 
    ...

```

The `annotation` style of the `mindmap` library mainly sets up a rectangular shape of appropriate size. Johannes configures the style by defining `every annotation` appropriately.

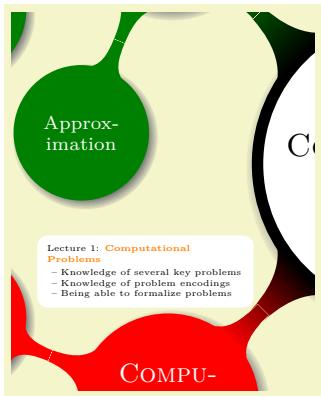


```
\usetikzlibrary {mindmap,shadows}
\begin{tikzpicture}[mindmap]
\clip (-5,-5) rectangle ++ (4,5);
\begin{scope}[
    every node/.style={concept, circular drop shadow, ...}] % as before
\node [root concept] (Computational Complexity) ... % as before
\end{scope}

\begin{scope}[every annotation/.style={fill=black!40}]
\node [annotation, above] at (Computational Problems.north) {
    Lecture 1: Computational Problems
    \begin{itemize}
        \item Knowledge of several key problems
        \item Knowledge of problem encodings
        \item Being able to formalize problems
    \end{itemize}
};
\end{scope}
\end{tikzpicture}
```

Well, that does not yet look quite perfect. The spacing or the `\itemize` is not really appropriate and the node is too large. Johannes can configure these things “by hand”, but it seems like a good idea to define a macro that will take care of these things for him. The “right” way to do this is to define a `\lecture` macro that takes a list of key-value pairs as argument and produces the desired annotation. However, to keep things simple, Johannes’ `\lecture` macro simply takes a fixed number of arguments having the following meaning: The first argument is the number of the lecture, the second is the name of the lecture, the third are positioning options like `above`, the fourth is the position where the node is placed, the fifth is the list of items to be shown, and the sixth is a date when the lecture will be held (this parameter is not yet needed, we will, however, need it later on).

```
\def\lecture#1#2#3#4#5#6{
\node [annotation, #3, scale=0.65, text width=4cm, inner sep=2mm] at (#4) {
    Lecture #1: \textcolor{orange}{\textbf{#2}}
    \list{}{#5}
    \begin{list}{}
        \topsep=2pt\itemsep=0pt\parsep=0pt
        \parskip=0pt\labelwidth=8pt\leftmargin=8pt
        \itemindent=0pt\labelsep=2pt
    \end{list}
    \#5
    \endlist
};
```



```
\usetikzlibrary {mindmap,shadows}
\begin{tikzpicture}[mindmap,every annotation/.style={fill=white}]
\clip (-5,-5) rectangle ++ (4,5);
\begin{scope}[
    every node/.style={concept, circular drop shadow, ...}] % as before
\node [root concept] (Computational Complexity) ... % as before
\end{scope}

\lecture{1}{Computational Problems}{above,xshift=-3mm}{Computational Problems.north}{%
    \item Knowledge of several key problems
    \item Knowledge of problem encodings
    \item Being able to formalize problems
}{2009-04-08}
```

In the same fashion Johannes can now add the other lecture annotations. Obviously, Johannes will have some trouble fitting everything on a single A4-sized page, but by adjusting the spacing and some experimentation he can quickly arrange all the annotations as needed.

6.5 Adding the Background

Johannes has already used colors to organize his lecture map into four regions, each having a different color. In order to emphasize these regions even more strongly, he wishes to add a background coloring to each of these regions.

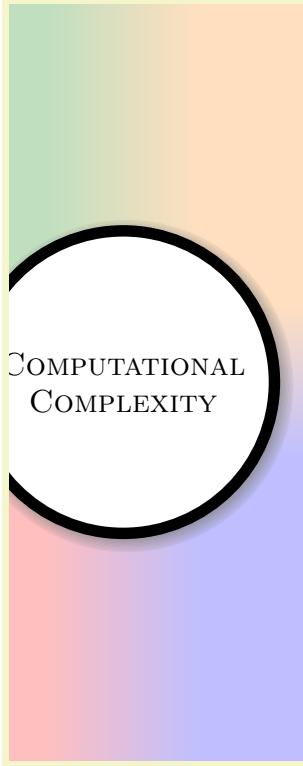
Adding these background colors turns out to be more tricky than Johannes would have thought. At first sight, what he needs is some sort of “color wheel” that is blue in the lower right direction and then

changes smoothly to orange in the upper right direction and then to green in the upper left direction and so on. Unfortunately, there is no easy way of creating such a color wheel shading (although it can be done, in principle, but only at a very high cost, see page 619 for an example).

Johannes decides to do something a bit more basic: He creates four large rectangles, one for each of the four quadrants around the central concept, each colored with a light version of the quadrant. Then, in order to “smooth” the change between adjacent rectangles, he puts four shadings on top of them.

Since these background rectangles should go “behind” everything else, Johannes puts all his background stuff on the `background` layer.

In the following code, only the central concept is shown to save some space:



```
\usetikzlibrary {backgrounds,mindmap,shadows}
\begin{tikzpicture}[
    mindmap,
    concept color=black,
    root concept/.append style={
        concept,
        circular drop shadow,
        fill=white, line width=1ex,
        text=black, font=\large\scshape
    }
]

\clip (-1.5,-5) rectangle ++(4,10);

\node [root concept] (Computational Complexity) {Computational Complexity};

\begin{pgfonlayer}{background}
\clip (-1.5,-5) rectangle ++(4,10);

\colorlet{upperleft}{green!50!black!25}
\colorlet{upperright}{orange!25}
\colorlet{lowerleft}{red!25}
\colorlet{lowerright}{blue!25}

% The large rectangles:
\fill [upperleft] (Computational Complexity) rectangle ++(-20,20);
\fill [upperright] (Computational Complexity) rectangle ++(20,20);
\fill [lowerleft] (Computational Complexity) rectangle ++(-20,-20);
\fill [lowerright] (Computational Complexity) rectangle ++(20,-20);

% The shadings:
\shade [left color=upperleft,right color=upperright]
 ([xshift=-1cm]Computational Complexity) rectangle ++(2,20);
\shade [left color=lowerleft,right color=lowerright]
 ([xshift=-1cm]Computational Complexity) rectangle ++(2,-20);
\shade [top color=upperleft,bottom color=lowerleft]
 ([yshift=-1cm]Computational Complexity) rectangle ++(-20,2);
\shade [top color=upperright,bottom color=lowerright]
 ([yshift=-1cm]Computational Complexity) rectangle ++(20,2);
\end{pgfonlayer}
\end{tikzpicture}
```

6.6 Adding the Calendar

Johannes intends to plan his lecture rather carefully. In particular, he already knows when each of his lectures will be held during the course. Naturally, this does not mean that Johannes will slavishly follow the plan and he might need longer for some subjects than he anticipated, but nevertheless he has a detailed plan of when which subject will be addressed.

Johannes intends to share this plan with his students by adding a calendar to the lecture map. In addition to serving as a reference on which particular day a certain topic will be addressed, the calendar is also useful to show the overall chronological order of the course.

In order to add a calendar to a TikZ graphic, the `calendar` library is most useful. The library provides the `\calendar` command, which takes a large number of options and which can be configured in many ways to produce just about any kind of calendar imaginable. For Johannes’ purposes, a simple `day list downward` will be a nice option since it produces a list of days that go “downward”.

```

1 \usetikzlibrary {calendar}
2 \tiny
3 \begin{tikzpicture}
4   \calendar [day list downward,
5             name=cal,
6             dates=2009-04-01 to 2009-04-14]
7     if (weekend)
8       [black!25];
9 \end{tikzpicture}
10
11
12
13
14

```

Using the `name` option, we gave a name to the calendar, which will allow us to reference the nodes that make up the individual days of the calendar later on. For instance, the rectangular node containing the 1 that represents April 1st, 2009, can be referenced as `(cal-2009-04-01)`. The `dates` option is used to specify an interval for which the calendar should be drawn. Johannes will need several months in his calendar, but the above example only shows two weeks to save some space.

Note the `if (weekend)` construct. The `\calendar` command is followed by options and then by `if`-statements. These `if`-statements are checked for each day of the calendar and when a date passes this test, the options or the code following the `if`-statement is executed. In the above example, we make weekend days (Saturdays and Sundays, to be precise) lighter than normal days. (Use your favorite calendar to check that, indeed, April 5th, 2009, is a Sunday.)

As mentioned above, Johannes can reference the nodes that are used to typeset days. Recall that his `\lecture` macro already got passed a date, which we did not use, yet. We can now use it to place the lecture's title next to the date when the lecture will be held:

```

\def\lecture#1#2#3#4#5#6{
% As before:
\node [annotation, #3, scale=0.65, text width=4cm, inner sep=2mm] at (#4) {
  Lecture #1: \textcolor{orange}{\textbf{#2}}
  \list{}{\topsep=2pt\itemsep=0pt\parsep=0pt
    \parskip=0pt\labelwidth=8pt\leftmargin=8pt
    \itemindent=0pt\labelsep=2pt}
#5
\endlist
};
% New:
\node [anchor=base west] at (cal-#6.base east) {\textcolor{orange}{\textbf{#2}}};
}

```

Johannes can now use this new `\lecture` command as follows (in the example, only the new part of the definition is used):

```

1 \usetikzlibrary {calendar}
2 \tiny
3 \begin{tikzpicture}
4   \calendar [day list downward,
5             name=cal,
6             dates=2009-04-01 to 2009-04-14]
7     if (weekend)
8       [black!25];
9
10
11
12
13
14
Computational Problems
%
% As before:
\lecture{1}{Computational Problems}{above,xshift=-3mm}
{Computational Problems.north}{

  \item Knowledge of several key problems
  \item Knowledge of problem encodings
  \item Being able to formalize problems
}{2009-04-08}
\end{tikzpicture}

```

As a final step, Johannes needs to add a few more options to the `calendar` command: He uses the `month text` option to configure how the text of a month is rendered (see Section 48 for details) and then typesets the month text at a special position at the beginning of each month.

```

1      April 2009
2
3
4
5
6
7
8 Computational Problems
9
10
11
12
13
14 Computational Models
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30

May 2009
1

\usetikzlibrary {calendar}
\tiny
\begin{tikzpicture}
\calendar [day list downward,
month text=\%mt\ \%y0,
month yshift=3.5em,
name=cal,
dates=2009-04-01 to 2009-05-01]
if (weekend)
[black!25]
if (day of month=1) {
\node at (0pt,1.5em) [anchor=base west] {\small\tikzmonthtext};
};

\lecture{1}{Computational Problems}{above,xshift=-3mm}
{Computational Problems.north}[
\item Knowledge of several key problems
\item Knowledge of problem encodings
\item Being able to formalize problems
]{2009-04-08}

\lecture{2}{Computational Models}{above,xshift=-3mm}
{Computational Models.north}[
\item Knowledge of Turing machines
\item Being able to compare the computational power of different
models
]{2009-04-15}
\end{tikzpicture}

```

6.7 The Complete Code

Putting it all together, Johannes gets the following code:

First comes the definition of the \lecture command:

```

\def\lecture#1#2#3#4#5#6{
% As before:
\node [annotation, #3, scale=0.65, text width=4cm, inner sep=2mm, fill=white] at (#4) {
Lecture #1: \textcolor{orange}{\textbf{#2}}
\list{--}{\topsep=2pt\itemsep=0pt\parsep=0pt
\parskip=0pt\labelwidth=8pt\leftmargin=8pt
\itemindent=0pt\labelsep=2pt}
#5
\endlist
};
% New:
\node [anchor=base west] at (cal-#6.base east) {\textcolor{orange}{\textbf{#2}}};
}

```

This is followed by the main mindmap setup...

```

\noindent
\begin{tikzpicture}
\begin{scope}[
mindmap,
every node/.style={concept, circular drop shadow, execute at begin node=\hspace{0pt},
root concept/.append style={
concept color=black,
fill=white, line width=1ex,
text=black, font=\large\scshape},
text=white,
computational problems/.style={concept color=red,faded/.style={concept color=red!50}},
computational models/.style={concept color=blue,faded/.style={concept color=blue!50}},
measuring complexity/.style={concept color=orange,faded/.style={concept color=orange!50}},
solving problems/.style={concept color=green!50!black,faded/.style={concept color=green!50!black!50}},
grow cyclic,
level 1/.append style={level distance=4.5cm,sibling angle=90,font=\scshape},
level 2/.append style={level distance=3cm,sibling angle=45,font=\scriptsize}]

```

...and contents:

```

\node [root concept] (Computational Complexity) {Computational Complexity} % root
    child [computational problems] { node [yshift=-1cm] (Computational Problems) {Computational Problems}
        child { node (Problem Measures) {Problem Measures} }
        child { node (Problem Aspects) {Problem Aspects} }
        child [faded] { node (problem Domains) {Problem Domains} }
        child { node (Key Problems) {Key Problems} }
    }
    child [computational models] { node [yshift=-1cm] (Computational Models) {Computational Models}
        child { node (Turing Machines) {Turing Machines} }
        child [faded] { node (Random-Access Machines) {Random-Access Machines} }
        child { node (Circuits) {Circuits} }
        child [faded] { node (Binary Decision Diagrams) {Binary Decision Diagrams} }
        child { node (Oracle Machines) {Oracle Machines} }
        child { node (Programming in Logic) {Programming in Logic} }
    }
    child [measuring complexity] { node [yshift=1cm] (Measuring Complexity) {Measuring Complexity}
        child { node (Complexity Measures) {Complexity Measures} }
        child { node (Classifying Complexity) {Classifying Complexity} }
        child { node (Comparing Complexity) {Comparing Complexity} }
        child [faded] { node (Describing Complexity) {Describing Complexity} }
    }
    child [solving problems] { node [yshift=1cm] (Solving Problems) {Solving Problems}
        child { node (Exact Algorithms) {Exact Algorithms} }
        child { node (Randomization) {Randomization} }
        child { node (Fixed-Parameter Algorithms) {Fixed-Parameter Algorithms} }
        child { node (Parallel Computation) {Parallel Computation} }
        child { node (Partial Solutions) {Partial Solutions} }
        child { node (Approximation) {Approximation} }
    }
};

\end{scope}

```

Now comes the calendar code:

```

\tiny
\calendar [day list downward,
            month text=\%mt\ \%y0,
            month yshift=3.5em,
            name=cal,
            at={(-.5\textwidth-5mm,.5\textheight-1cm)},
            dates=2009-04-01 to 2009-06-last]
if (weekend)
    [black!25]
if (day of month=1) {
    \node at (0pt,1.5em) [anchor=base west] {\small\tikzmonthtext};
};

```

The lecture annotations:

```

\lecture{1}{Computational Problems}{above,xshift=-5mm,yshift=5mm}{Computational Problems.north}{

    \item Knowledge of several key problems
    \item Knowledge of problem encodings
    \item Being able to formalize problems
}{2009-04-08}

\lecture{2}{Computational Models}{above left}{Computational Models.west}{

    \item Knowledge of Turing machines
    \item Being able to compare the computational power of different
          models
}{2009-04-15}

```

Finally, the background:

```

\begin{pgfonlayer}{background}
\clip[xshift=-1cm] (-.5\textwidth,-.5\textheight) rectangle ++(\textwidth,\textheight);

\colorlet{upperleft}{green!50!black!25}
\colorlet{upperright}{orange!25}
\colorlet{lowerleft}{red!25}
\colorlet{lowerright}{blue!25}

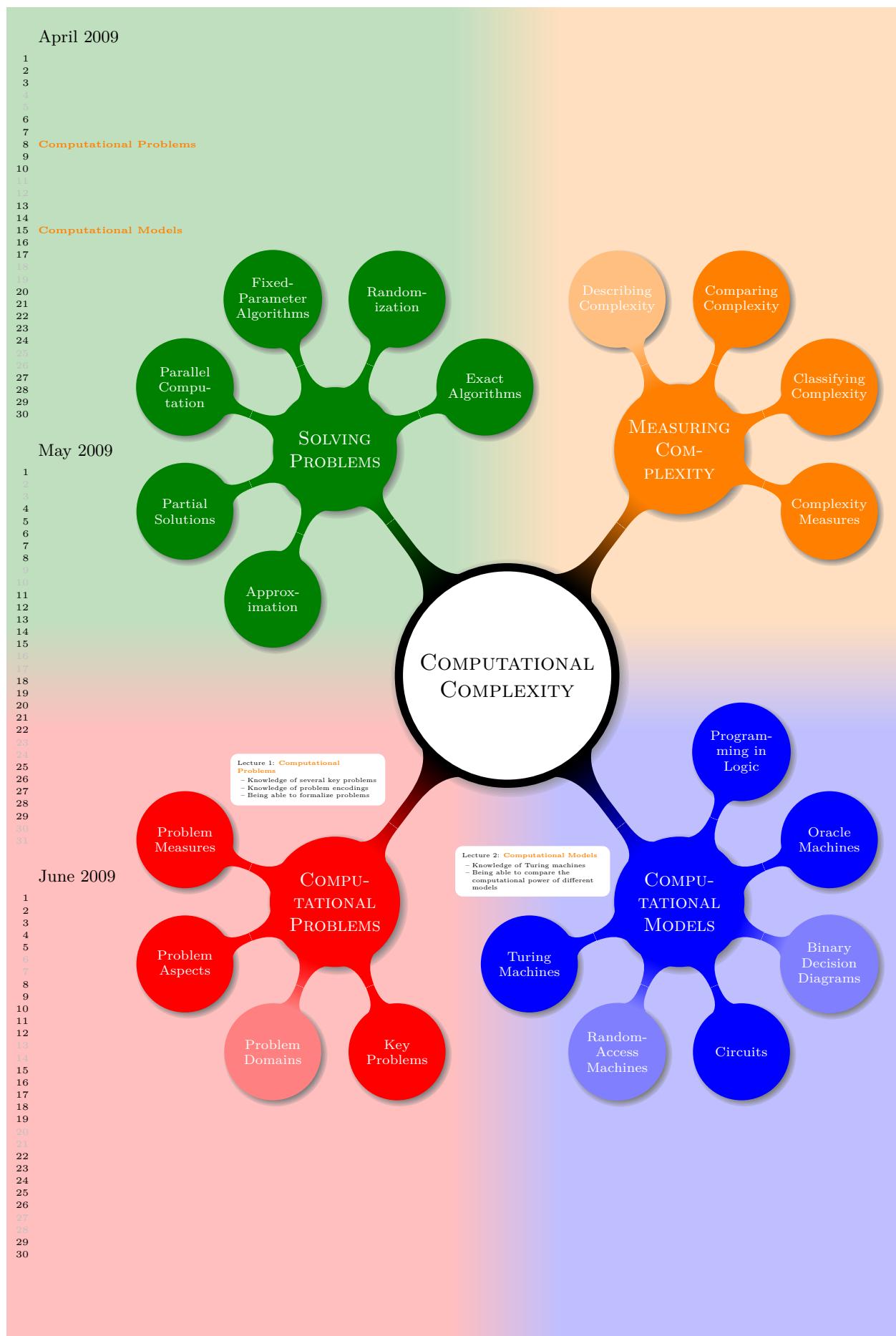
% The large rectangles:
\fill [upperleft] (Computational Complexity) rectangle ++(-20,20);
\fill [upperright] (Computational Complexity) rectangle ++(20,20);
\fill [lowerleft] (Computational Complexity) rectangle ++(-20,-20);
\fill [lowerright] (Computational Complexity) rectangle ++(20,-20);

% The shadings:
\shade [left color=upperleft,right color=upperright]
 ([xshift=-1cm]Computational Complexity) rectangle ++(2,20);
\shade [left color=lowerleft,right color=lowerright]
 ([xshift=-1cm]Computational Complexity) rectangle ++(2,-20);
\shade [top color=upperleft,bottom color=lowerleft]
 ([yshift=-1cm]Computational Complexity) rectangle ++(-20,2);
\shade [top color=upperright,bottom color=lowerright]
 ([yshift=-1cm]Computational Complexity) rectangle ++(20,2);

\end{pgfonlayer}
\end{tikzpicture}

```

The next page shows the resulting lecture map in all its glory (it would be somewhat more glorious, if there were more lecture annotations, but you should get the idea).



7 Guidelines on Graphics

The present section is not about PGF or *TikZ*, but about general guidelines and principles concerning the creation of graphics for scientific presentations, papers, and books.

The guidelines in this section come from different sources. Many of them are just what I would like to claim is “common sense”, some reflect my personal experience (though, hopefully, not my personal preferences), some come from books (the bibliography is still missing, sorry) on graphic design and typography. The most influential source are the brilliant books by Edward Tufte. While I do not agree with everything written in these books, many of Tufte’s arguments are so convincing that I decided to repeat them in the following guidelines.

The first thing you should ask yourself when someone presents a bunch of guidelines is: Should I really follow these guidelines? This is an important question, because there are good reasons not to follow general guidelines. The person who set up the guidelines may have had other objectives than you do. For example, a guideline might say “use the color red for emphasis”. While this guideline makes perfect sense for, say, a presentation using a projector, red “color” has the *opposite* effect of “emphasis” when printed using a black-and-white printer. Guidelines were almost always set up to address a specific situation. If you are not in this situation, following a guideline can do more harm than good.

The second thing you should be aware of is the basic rule of typography is: “Every rule can be broken, as long as you are *aware* that you are breaking a rule.” This rule also applies to graphics. Phrased differently, the basic rule states: “The only mistakes in typography are things done in ignorance.” When you are aware of a rule and when you decide that breaking the rule has a desirable effect, break the rule.

7.1 Planning the Time Needed for the Creation of Graphics

When you create a paper with numerous graphics, the time needed to create these graphics becomes an important factor. How much time should you calculate for the creation of graphics?

As a general rule, assume that a graphic will need as much time to create as would a text of the same length. For example, when I write a paper, I need about one hour per page for the first draft. Later, I need between two and four hours per page for revisions. Thus, I expect to need about half an hour for the creation of a *first draft* of a half page graphic. Later on, I expect another one to two hours before the final graphic is finished.

In many publications, even in good journals, the authors and editors have obviously invested a lot of time on the text, but seem to have spent about five minutes to create all of the graphics. Graphics often seem to have been added as an “afterthought” or look like a screen shot of whatever the authors’s statistical software shows them. As will be argued later on, the graphics that programs like *GNUPLOT* produce by default are of poor quality.

Creating informative graphics that help the reader and that fit together with the main text is a difficult, lengthy process.

- Treat graphics as first-class citizens of your papers. They deserve as much time and energy as the text does. Indeed, the creation of graphics might deserve *even more* time than the writing of the main text since more attention will be paid to the graphics and they will be looked at first.
- Plan as much time for the creation and revision of a graphic as you would plan for text of the same size.
- Difficult graphics with a high information density may require even more time.
- Very simple graphics will require less time, but most likely you do not want to have “very simple graphics” in your paper, anyway; just as you would not like to have a “very simple text” of the same size.

7.2 Workflow for Creating a Graphic

When you write a (scientific) paper, you will most likely follow the following pattern: You have some results/ideas that you would like to report about. The creation of the paper will typically start with compiling a rough outline. Then, the different sections are filled with text to create a first draft. This draft is then revised repeatedly until, often after substantial revision, a final paper results. In a good journal paper there is typically not be a single sentence that has survived unmodified from the first draft.

Creating a graphics follows the same pattern:

- Decide on what the graphic should communicate. Make this a conscious decision, that is, determine “What is the graphic supposed to tell the reader?”
- Create an “outline”, that is, the rough overall “shape” of the graphic, containing the most crucial elements. Often, it is useful to do this using pencil and paper.
- Fill out the finer details of the graphic to create a first draft.
- Revise the graphic repeatedly along with the rest of the paper.

7.3 Linking Graphics With the Main Text

Graphics can be placed at different places in a text. Either, they can be inlined, meaning they are somewhere “in the middle of the text” or they can be placed in stand-alone “figures”. Since printers (the people) like to have their pages “filled”, (both for aesthetic and economic reasons) stand-alone figures may traditionally be placed on pages in the document far away from the main text that refers to them. L^AT_EX and T_EX tend to encourage this “drifting away” of graphics for technical reasons.

When a graphic is inlined, it will more or less automatically be linked with the main text in the sense that the labels of the graphic will be implicitly explained by the surrounding text. Also, the main text will typically make it clear what the graphic is about and what is shown.

Quite differently, a stand-alone figure will often be viewed at a time when the main text that this graphic belongs to either has not yet been read or has been read some time ago. For this reason, you should follow the following guidelines when creating stand-alone figures:

- Stand-alone figures should have a caption than should make them “understandable by themselves”. For example, suppose a graphic shows an example of the different stages of a quicksort algorithm. Then the figure’s caption should, at the very least, inform the reader that “the figure shows the different stages of the quicksort algorithm introduced on page xyz”. and not just “Quicksort algorithm”.
- A good caption adds as much context information as possible. For example, you could say: “The figure shows the different stages of the quicksort algorithm introduced on page xyz. In the first line, the pivot element 5 is chosen. This causes...” While this information can also be given in the main text, putting it in the caption will ensure that the context is kept. Do not feel afraid of a 5-line caption. (Your editor may hate you for this. Consider hating them back.)
- Reference the graphic in your main text as in “for an example of quicksort ‘in action’, see Figure 2.1 on page xyz”.
- Most books on style and typography recommend that you do not use abbreviations as in “Fig. 2.1” but write “Figure 2.1”.

The main argument against abbreviations is that “a period is too valuable to waste it on an abbreviation”. The idea is that a period will make the reader assume that the sentence ends after “Fig.” and it takes a “conscious backtracking” to realize that the sentence did not end after all.

The argument in favor of abbreviations is that they save space.

Personally, I am not really convinced by either argument. On the one hand, I have not yet seen any hard evidence that abbreviations slow readers down. On the other hand, abbreviating all “Figure” by “Fig.” is most unlikely to save even a single line in most documents. I avoid abbreviations.

7.4 Consistency Between Graphics and Text

Perhaps the most common “mistake” people do when creating graphics (remember that a “mistake” in design is always just “ignorance”) is to have a mismatch between the way their graphics look and the way their text looks.

It is quite common that authors use several different programs for creating the graphics of a paper. An author might produce some plots using G_NUPL_OT, a diagram using XFIG, and include an .eps graphic a coauthor contributed using some unknown program. All these graphics will, most likely, use different line widths, different fonts, and have different sizes. In addition, authors often use options like [height=5cm when including graphics to scale them to some “nice size”.

If the same approach were taken to writing the main text, every section would be written in a different font at a different size. In some sections all theorems would be underlined, in another they would be printed

all in uppercase letters, and in another in red. In addition, the margins would be different on each page. Readers and editors would not tolerate a text if it were written in this fashion, but with graphics they often have to.

To create consistency between graphics and text, stick to the following guidelines:

- Do not scale graphics.

This means that when generating graphics using an external program, create them “at the right size”.

- Use the same font(s) both in graphics and the body text.

- Use the same line width in text and graphics.

The “line width” for normal text is the width of the stem of letters like T. For TeX, this is usually 0.4 pt. However, some journals will not accept graphics with a normal line width below 0.5 pt.

- When using colors, use a consistent color coding in the text and in graphics. For example, if red is supposed to alert the reader to something in the main text, use red also in graphics for important parts of the graphic. If blue is used for structural elements like headlines and section titles, use blue also for structural elements of your graphic.

However, graphics may also use a logical intrinsic color coding. For example, no matter what colors you normally use, readers will generally assume, say, that the color green as “positive, go, ok” and red as “alert, warning, action”.

Creating consistency when using different graphic programs is almost impossible. For this reason, you should consider sticking to a single graphics program.

7.5 Labels in Graphics

Almost all graphics will contain labels, that is, pieces of text that explain parts of the graphics. When placing labels, stick to the following guidelines:

- Follow the rule of consistency when placing labels. You should do so in two ways: First, be consistent with the main text, that is, use the same font as the main text also for labels. Second, be consistent between labels, that is, if you format some labels in some particular way, format all labels in this way.
- In addition to using the same fonts in text and graphics, you should also use the same notation. For example, if you write $1/2$ in your main text, also use “ $1/2$ ” as labels in graphics, not “ 0.5 ”. A π is a “ π ” and not “ 3.141 ”. Finally, $e^{-i\pi}$ is “ $e^{-i\pi}$ ”, not “ -1 ”, let alone “ -1 ”.
- Labels should be legible. They should not only have a reasonably large size, they also should not be obscured by lines or other text. This also applies to labels of lines and text *behind* the labels.
- Labels should be “in place”. Whenever there is enough space, labels should be placed next to the thing they label. Only if necessary, add a (subdued) line from the label to the labeled object. Try to avoid labels that only reference explanations in external legends. Reader have to jump back and forth between the explanation and the object that is described.
- Consider subduing “unimportant” labels using, for example, a gray color. This will keep the focus on the actual graphic.

7.6 Plots and Charts

One of the most frequent kind of graphics, especially in scientific papers, are *plots*. They come in a large variety, including simple line plots, parametric plots, three dimensional plots, pie charts, and many more.

Unfortunately, plots are notoriously hard to get right. Partly, the default settings of programs like GNUPLOT or Excel are to blame for this since these programs make it very convenient to create bad plots.

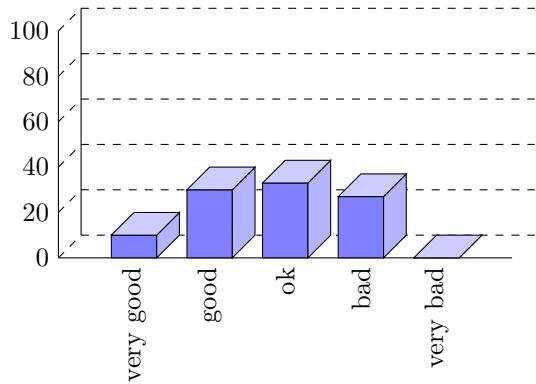
The first question you should ask yourself when creating a plot is: Are there enough data points to merit a plot? If the answer is “not really”, use a table.

A typical situation where a plot is unnecessary is when people present a few numbers in a bar diagram. Here is a real-life example: At the end of a seminar a lecturer asked the participants for feedback. Of the 50 participants, 30 returned the feedback form. According to the feedback, three participants considered the seminar “very good”, nine considered it “good”, ten “ok”, eight “bad”, and no one thought that the seminar was “very bad”.

A simple way of summing up this information is the following table:

<i>Rating given</i>	<i>Participants (out of 50)</i>	<i>Percentage who gave this rating</i>
“very good”	3	6%
“good”	9	18%
“ok”	10	20%
“bad”	8	16%
“very bad”	0	0%
none	20	40%

What the lecturer did was to visualize the data using a 3D bar diagram. It looked like this (except that in reality the numbers were typeset using some extremely low-resolution bitmap font and were near-unreadable):



Both the table and the “plot” have about the same size. If your first thought is “the graphic looks nicer than the table”, try to answer the following questions based on the information in the table or in the graphic:

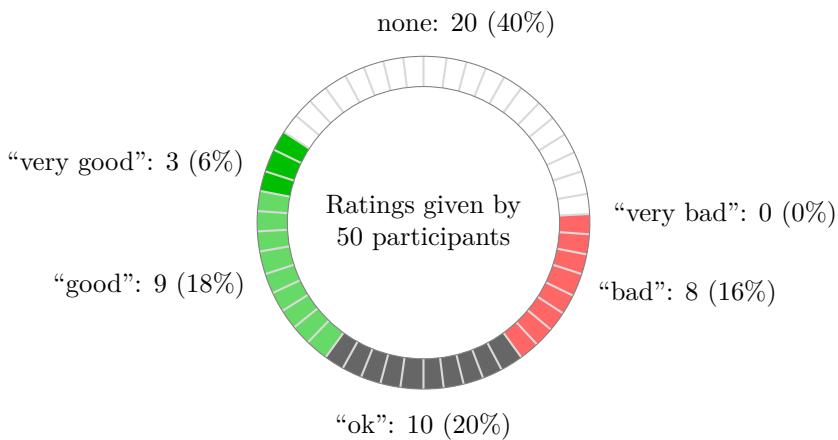
1. How many participants were there?
2. How many participants returned the feedback form?
3. What percentage of the participants returned the feedback form?
4. How many participants checked “very good”?
5. What percentage out of all participants checked “very good”?
6. Did more than a quarter of the participants check “bad” or “very bad”?
7. What percentage of the participants that returned the form checked “very good”?

Sadly, the graphic does not allow us to answer *a single one of these questions*. The table answers all of them directly, except for the last one. In essence, the information density of the graphic is very close to zero. The table has a much higher information density; despite the fact that it uses quite a lot of white space to present a few numbers. Here is the list of things that went wrong with the 3D-bar diagram:

- The whole graphic is dominated by irritating background lines.
- It is not clear what the numbers at the left mean; presumably percentages, but it might also be the absolute number of participants.
- The labels at the bottom are rotated, making them hard to read.
(In the real presentation that I saw, the text was rendered at a very low resolution with about 10 by 6 pixels per letter with wrong kerning, making the rotated text almost impossible to read.)
- The third dimension adds complexity to the graphic without adding information.

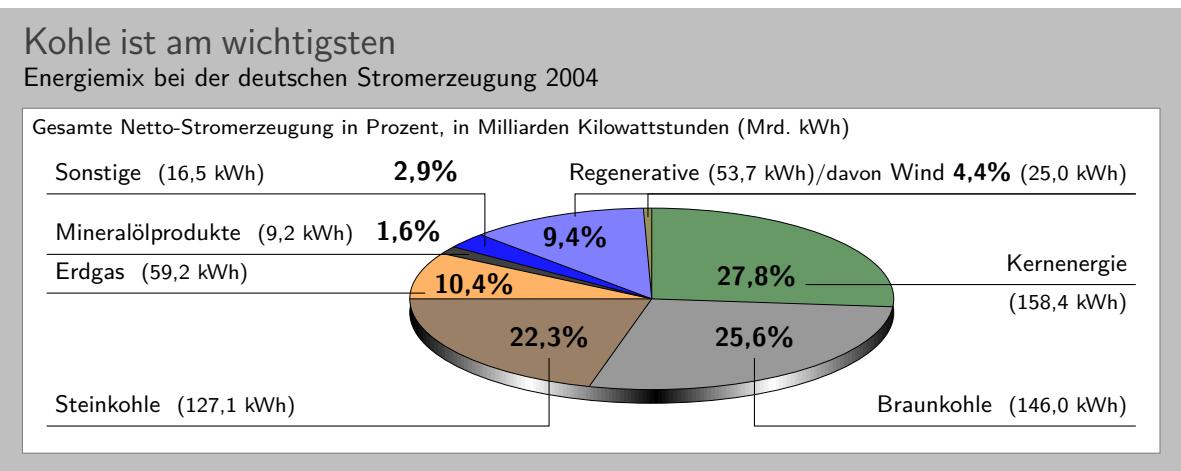
- The three dimensional setup makes it much harder to gauge the height of the bars correctly. Consider the “bad” bar. Is the number this bar stands for more than 20 or less? While the front of the bar is below the 20 line, the back of the bar (which counts) is above.
- It is impossible to tell which numbers are represented by the bars. Thus, the bars needlessly hide the information these bars are all about.
- What do the bar heights add up to? Is it 100% or 60%?
- Does the bar for “very bad” represent 0 or 1?
- Why are the bars blue?

You might argue that in the example the exact numbers are not important for the graphic. The important things is the “message”, which is that there are more “very good” and “good” ratings than “bad” and “very bad”. However, to convey this message either use a sentence that says so or use a graphic that conveys this message more clearly:



The above graphic has about the same information density as the table (about the same size and the same numbers are shown). In addition, one can directly “see” that there are more good or very good ratings than bad ones. One can also “see” that the number of people who gave no rating at all is not negligible, which is quite common for feedback forms.

Charts are not always a good idea. Let us look at an example that I redrew from a pie chart in *Die Zeit*, June 4th, 2005:



This graphic has been redrawn in TikZ, but the original looks almost exactly the same.

At first sight, the graphic looks “nice and informative”, but there are a lot of things that went wrong:

- The chart is three dimensional. However, the shadings add nothing “information-wise”, at best, they distract.

- In a 3D-pie-chart the relative sizes are very strongly distorted. For example, the area taken up by the gray color of “Braunkohle” is larger than the area taken up by the green color of “Kernenergie” *despite the fact that the percentage of Braunkohle is less than the percentage of Kernenergie.*
- The 3D-distortion gets worse for small areas. The area of “Regenerative” somewhat larger than the area of “Erdgas”. The area of “Wind” is slightly smaller than the area of “Mineralölprodukte” *although the percentage of Wind is nearly three times larger than the percentage of Mineralölprodukte.*

In the last case, the different sizes are only partly due to distortion. The designer(s) of the original graphic have also made the “Wind” slice too small, even taking distortion into account. (Just compare the size of “Wind” to “Regenerative” in general.)

- According to its caption, this chart is supposed to inform us that coal was the most important energy source in Germany in 2004. Ignoring the strong distortions caused by the superfluous and misleading 3D-setup, it takes quite a while for this message to get across.

Coal as an energy source is split up into two slices: one for “Steinkohle” and one for “Braunkohle” (two different kinds of coal). When you add them up, you see that the whole lower half of the pie chart is taken up by coal.

The two areas for the different kinds of coal are not visually linked at all. Rather, two different colors are used, the labels are on different sides of the graphic. By comparison, “Regenerative” and “Wind” are very closely linked.

- The color coding of the graphic follows no logical pattern at all. Why is nuclear energy green? Regenerative energy is light blue, “other sources” are blue. It seems more like a joke that the area for “Braunkohle” (which literally translates to “brown coal”) is stone gray, while the area for “Steinkohle” (which literally translates to “stone coal”) is brown.
- The area with the lightest color is used for “Erdgas”. This area stands out most because of the brighter color. However, for this chart “Erdgas” is not really important at all.

Edward Tufte calls graphics like the above “chart junk”. (I am happy to announce, however, that *Die Zeit* has stopped using 3D pie charts and their information graphics have got somewhat better.)

Here are a few recommendations that may help you avoid producing chart junk:

- Do not use 3D pie charts. They are *evil*.
- Consider using a table instead of a pie chart.
- Do not apply colors randomly; use them to direct the reader’s focus and to group things.
- Do not use background patterns, like a crosshatch or diagonal lines, instead of colors. They distract. Background patterns in information graphics are *evil*.

7.7 Attention and Distraction

Pick up your favorite fiction novel and have a look at a typical page. You will notice that the page is very uniform. Nothing is there to distract the reader while reading; no large headlines, no bold text, no large white areas. Indeed, even when the author does wish to emphasize something, this is done using italic letters. Such letters blend nicely with the main text – at a distance you will not be able to tell whether a page contains italic letters, but you would notice a single bold word immediately. The reason novels are typeset this way is the following paradigm: Avoid distractions.

Good typography (like good organization) is something you do *not* notice. The job of typography is to make reading the text, that is, “absorbing” its information content, as effortless as possible. For a novel, readers absorb the content by reading the text line-by-line, as if they were listening to someone telling the story. In this situation anything on the page that distracts the eye from going quickly and evenly from line to line will make the text harder to read.

Now, pick up your favorite weekly magazine or newspaper and have a look at a typical page. You will notice that there is quite a lot “going on” on the page. Fonts are used at different sizes and in different arrangements, the text is organized in narrow columns, typically interleaved with pictures. The reason magazines are typeset in this way is another paradigm: Steer attention.

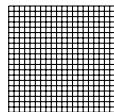
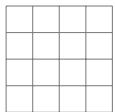
Readers will not read a magazine like a novel. Instead of reading a magazine line-by-line, we use headlines and short abstracts to check whether we want to read a certain article or not. The job of typography is to

steer our attention to these abstracts and headlines, first. Once we have decided that we want to read an article, however, we no longer tolerate distractions, which is why the main text of articles is typeset exactly the same way as a novel.

The two principles “avoid distractions” and “steer attention” also apply to graphics. When you design a graphic, you should eliminate everything that will “distract the eye”. At the same time, you should try to actively help the reader “through the graphic” by using fonts/colors/line widths to highlight different parts.

Here is a non-exhaustive list of things that can distract readers:

- Strong contrasts will always be registered first by the eye. For example, consider the following two grids:



Even though the left grid comes first in English reading order, the right one is much more likely to be seen first: The white-to-black contrast is higher than the gray-to-white contrast. In addition, there are more “places” adding to the overall contrast in the right grid.

Things like grids and, more generally, help lines usually should not grab the attention of the readers and, hence, should be typeset with a low contrast to the background. Also, a loosely-spaced grid is less distracting than a very closely-spaced grid.

- Dashed lines create many points at which there is black-to-white contrast. Dashed or dotted lines can be very distracting and, hence, should be avoided in general.

Do not use different dashing patterns to differentiate curves in plots. You lose data points this way and the eye is not particularly good at “grouping things according to a dashing pattern”. The eye is *much* better at grouping things according to colors.

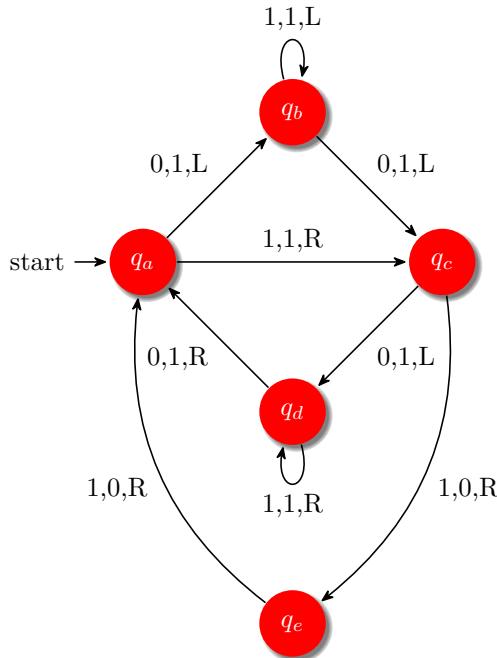
- Background patterns filling an area using diagonal lines or horizontal and vertical lines or just dots are almost always distracting and, usually, serve no real purpose.
- Background images and shadings distract and only seldomly add anything of importance to a graphic.
- Cute little clip arts can easily draw attention away from the data.

Part II

Installation and Configuration

by Till Tantau

This part explains how the system is installed. Typically, someone has already done so for your system, so this part can be skipped; but if this is not the case and you are the poor fellow who has to do the installation, read the present part.



The current candidate for the busy beaver for five states. It is presumed that this Turing machine writes a maximum number of 1's before halting among all Turing machines with five states and the tape alphabet {0, 1}. Proving this conjecture is an open research problem.

```

\usetikzlibrary {arrows.meta,automata,positioning,shadows}
\begin{tikzpicture} [-,>={Stealth[round]},shorten >=1pt,auto,node distance=2.8cm,on grid,semithick,
  every state/.style={fill=red,draw=none,circular drop shadow,text=white}]

\node [initial,state] (A)      {$q_a$};
\node [state] (B) [above right=of A] {$q_b$};
\node [state] (D) [below right=of A] {$q_d$};
\node [state] (C) [below right=of B] {$q_c$};
\node [state] (E) [below=of D]      {$q_e$};

\path (A) edge      node {0,1,L} (B)
      edge      node {1,1,R} (C)
      edge [loop above] node {1,1,L} (B)
      edge      node {0,1,L} (D)
      edge [bend left] node {1,0,R} (E)
      edge [loop below] node {1,1,R} (D)
      edge      node {0,1,R} (A)
      edge [bend left] node {1,0,R} (A);

\node [right=1cm,text width=8cm] at (C)
{
  The current candidate for the busy beaver for five states. It is
  presumed that this Turing machine writes a maximum number of
  $1$'s before halting among all Turing machines with five states
  and the tape alphabet $\{0, 1\}$. Proving this conjecture is an
  open research problem.
};

\end{tikzpicture}

```

8 Installation

There are different ways of installing PGF, depending on your system and needs, and you may need to install other packages as well, see below. Before installing, you may wish to review the licenses under which the package is distributed, see Section 9.

Typically, the package will already be installed on your system. Naturally, in this case you do not need to worry about the installation process at all and you can skip the rest of this section.

8.1 Package and Driver Versions

This documentation is part of version 3.1.5b of the PGF package. In order to run PGF, you need a reasonably recent \TeX installation. When using \LaTeX , you need the following packages installed (newer versions should also work):

- `xcolor` version 2.00.

With plain \TeX , `xcolor` is not needed, but you obviously do not get its (full) functionality.

Currently, PGF supports the following backend drivers:

- `luatex` version 0.76 or higher. Most earlier versions also work.
- `pdftex` version 0.14 or higher. Earlier versions do not work.
- `dvips` version 5.94a or higher. Earlier versions may also work.

For inter-picture connections, you need to process pictures using `pdftex` version 1.40 or higher running in DVI mode.

- `dvipdfm` version 0.13.2c or higher. Earlier versions may also work.

For inter-picture connections, you need to process pictures using `pdftex` version 1.40 or higher running in DVI mode.

- `dvipdfmx` version 0.13.2c or higher. Earlier versions may also work.
- `dvisvgm` version 1.2.2 or higher. Earlier versions may also work.
- `tex4ht` version 2003-05-05 or higher. Earlier versions may also work.
- `vtex` version 8.46a or higher. Earlier versions may also work.
- `textures` version 2.1 or higher. Earlier versions may also work.
- `xetex` version 0.996 or higher. Earlier versions may also work.

Currently, PGF supports the following formats:

- `latex` with complete functionality.
- `plain` with complete functionality, except for graphics inclusion, which works only for `pdf\TeX`.
- `context` with complete functionality, except for graphics inclusion, which works only for `pdf\TeX`.

For more details, see Section 10.

8.2 Installing Prebundled Packages

I do not create or manage prebundled packages of PGF, but, fortunately, nice other people do. I cannot give detailed instructions on how to install these packages, since I do not manage them, but I *can* tell you where to find them. If you have a problem with installing, you might wish to have a look at the Debian page or the MiK \TeX page first.

8.2.1 Debian

The command “`aptitude install pgf`” should do the trick. Sit back and relax.

8.2.2 MiKTeX

For MiKTeX, use the update wizard to install the (latest versions of the) packages called `pgf` and `xcolor`.

8.3 Installation in a `texmf` Tree

For a permanent installation, you place the files of the PGF package in an appropriate `texmf` tree.

When you ask TeX to use a certain class or package, it usually looks for the necessary files in so-called `texmf` trees. These trees are simply huge directories that contain these files. By default, TeX looks for files in three different `texmf` trees:

- The root `texmf` tree, which is usually located at `/usr/share/texmf/` or `c:\texmf\` or somewhere similar.
- The local `texmf` tree, which is usually located at `/usr/local/share/texmf/` or `c:\localtexmf\` or somewhere similar.
- Your personal `texmf` tree, which is usually located in your home directory at `~/texmf/` or `~/Library/texmf/`.

You should install the packages either in the local tree or in your personal tree, depending on whether you have write access to the local tree. Installation in the root tree can cause problems, since an update of the whole TeX installation will replace this whole tree.

8.3.1 Installation that Keeps Everything Together

Once you have located the right `texmf` tree, you must decide whether you want to install PGF in such a way that “all its files are kept in one place” or whether you want to be “TDS-compliant”, where TDS means “TeX directory structure”.

If you want to keep “everything in one place”, inside the `texmf` tree that you have chosen create a sub-sub-directory called `texmf/tex/generic/pgf` or `texmf/tex/generic/pgf-3.1.5b`, if you prefer. Then place all files of the `pgf` package in this directory. Finally, rebuild TeX’s filename database. This is done by running the command `texhash` or `mktexlsr` (they are the same). In MiKTeX, there is a menu option to do this.

8.3.2 Installation that is TDS-Compliant

While the above installation process is the most “natural” one and although I would like to recommend it since it makes updating and managing the PGF package easy, it is not TDS-compliant. If you want to be TDS-compliant, proceed as follows: (If you do not know what TDS-compliant means, you probably do not want to be TDS-compliant.)

The `.tar` file of the `pgf` package contains the following files and directories at its root: `README`, `doc`, `generic`, `plain`, and `latex`. You should “merge” each of the four directories with the following directories `texmf/doc`, `texmf/tex/generic`, `texmf/tex/plain`, and `texmf/tex/latex`. For example, in the `.tar` file the `doc` directory contains just the directory `pgf`, and this directory has to be moved to `texmf/doc/pgf`. The root `README` file can be ignored since it is reproduced in `doc/pgf/README`.

You may also consider keeping everything in one place and using symbolic links to point from the TDS-compliant directories to the central installation.

For a more detailed explanation of the standard installation process of packages, you might wish to consult <http://www.ctan.org/installationadvice/>. However, note that the PGF package does not come with a `.ins` file (simply skip that part).

8.4 Updating the Installation

To update your installation from a previous version, all you need to do is to replace everything in the directory `texmf/tex/generic/pgf` with the files of the new version (or in all the directories where `pgf` was installed, if you chose a TDS-compliant installation). The easiest way to do this is to first delete the old version and then proceed as described above. Sometimes, there are changes in the syntax of certain commands from version to version. If things no longer work that used to work, you may wish to have a look at the release notes and at the change log.

9 Licenses and Copyright

9.1 Which License Applies?

Different parts of the PGF package are distributed under different licenses:

1. The *code* of the package is dual-license. This means that you can decide which license you wish to use when using the PGF package. The two options are:
 - (a) You can use the GNU Public License, version 2.
 - (b) You can use the L^AT_EX Project Public License, version 1.3c.
2. The *documentation* of the package is also dual-license. Again, you can choose between two options:
 - (a) You can use the GNU Free Documentation License, version 1.2.
 - (b) You can use the L^AT_EX Project Public License, version 1.3c.

The “documentation of the package” refers to all files in the subdirectory `doc` of the `pgf` package. A detailed listing can be found in the file `doc/generic/pgf/licenses/manifest-documentation.txt`. All files in other directories are part of the “code of the package”. A detailed listing can be found in the file `doc/generic/pgf/licenses/manifest-code.txt`.

In the rest of this section, the licenses are presented. The following text is copyrighted, see the plain text versions of these licenses in the directory `doc/generic/pgf/licenses` for details.

The example picture used in this manual, the Brave GNU World logo, is taken from the Brave GNU World homepage, where it is copyrighted as follows: “Copyright (C) 1999, 2000, 2001, 2002, 2003, 2004 Georg C. F. Greve. Permission is granted to make and distribute verbatim copies of this transcript as long as the copyright and this permission notice appear.”

9.2 The GNU Public License, Version 2

9.2.1 Preamble

The licenses for most software are designed to take away your freedom to share and change it. By contrast, the GNU General Public License is intended to guarantee your freedom to share and change free software—to make sure the software is free for all its users. This General Public License applies to most of the Free Software Foundation’s software and to any other program whose authors commit to using it. (Some other Free Software Foundation software is covered by the GNU Library General Public License instead.) You can apply it to your programs, too.

When we speak of free software, we are referring to freedom, not price. Our General Public Licenses are designed to make sure that you have the freedom to distribute copies of free software (and charge for this service if you wish), that you receive source code or can get it if you want it, that you can change the software or use pieces of it in new free programs; and that you know you can do these things.

To protect your rights, we need to make restrictions that forbid anyone to deny you these rights or to ask you to surrender the rights. These restrictions translate to certain responsibilities for you if you distribute copies of the software, or if you modify it.

For example, if you distribute copies of such a program, whether gratis or for a fee, you must give the recipients all the rights that you have. You must make sure that they, too, receive or can get the source code. And you must show them these terms so they know their rights.

We protect your rights with two steps: (1) copyright the software, and (2) offer you this license which gives you legal permission to copy, distribute and/or modify the software.

Also, for each author’s protection and ours, we want to make certain that everyone understands that there is no warranty for this free software. If the software is modified by someone else and passed on, we want its recipients to know that what they have is not the original, so that any problems introduced by others will not reflect on the original authors’ reputations.

Finally, any free program is threatened constantly by software patents. We wish to avoid the danger that redistributors of a free program will individually obtain patent licenses, in effect making the program proprietary. To prevent this, we have made it clear that any patent must be licensed for everyone’s free use or not licensed at all.

The precise terms and conditions for copying, distribution and modification follow.

9.2.2 Terms and Conditions For Copying, Distribution and Modification

0. This License applies to any program or other work which contains a notice placed by the copyright holder saying it may be distributed under the terms of this General Public License. The “Program”, below, refers to any such program or work, and a “work based on the Program” means either the Program or any derivative work under copyright law: that is to say, a work containing the Program or a portion of it, either verbatim or with modifications and/or translated into another language. (Hereinafter, translation is included without limitation in the term “modification”.) Each licensee is addressed as “you”.

Activities other than copying, distribution and modification are not covered by this License; they are outside its scope. The act of running the Program is not restricted, and the output from the Program is covered only if its contents constitute a work based on the Program (independent of having been made by running the Program). Whether that is true depends on what the Program does.

1. You may copy and distribute verbatim copies of the Program’s source code as you receive it, in any medium, provided that you conspicuously and appropriately publish on each copy an appropriate copyright notice and disclaimer of warranty; keep intact all the notices that refer to this License and to the absence of any warranty; and give any other recipients of the Program a copy of this License along with the Program.

You may charge a fee for the physical act of transferring a copy, and you may at your option offer warranty protection in exchange for a fee.

2. You may modify your copy or copies of the Program or any portion of it, thus forming a work based on the Program, and copy and distribute such modifications or work under the terms of Section 1 above, provided that you also meet all of these conditions:

- (a) You must cause the modified files to carry prominent notices stating that you changed the files and the date of any change.
- (b) You must cause any work that you distribute or publish, that in whole or in part contains or is derived from the Program or any part thereof, to be licensed as a whole at no charge to all third parties under the terms of this License.
- (c) If the modified program normally reads commands interactively when run, you must cause it, when started running for such interactive use in the most ordinary way, to print or display an announcement including an appropriate copyright notice and a notice that there is no warranty (or else, saying that you provide a warranty) and that users may redistribute the program under these conditions, and telling the user how to view a copy of this License. (Exception: if the Program itself is interactive but does not normally print such an announcement, your work based on the Program is not required to print an announcement.)

These requirements apply to the modified work as a whole. If identifiable sections of that work are not derived from the Program, and can be reasonably considered independent and separate works in themselves, then this License, and its terms, do not apply to those sections when you distribute them as separate works. But when you distribute the same sections as part of a whole which is a work based on the Program, the distribution of the whole must be on the terms of this License, whose permissions for other licensees extend to the entire whole, and thus to each and every part regardless of who wrote it.

Thus, it is not the intent of this section to claim rights or contest your rights to work written entirely by you; rather, the intent is to exercise the right to control the distribution of derivative or collective works based on the Program.

In addition, mere aggregation of another work not based on the Program with the Program (or with a work based on the Program) on a volume of a storage or distribution medium does not bring the other work under the scope of this License.

3. You may copy and distribute the Program (or a work based on it, under Section 2) in object code or executable form under the terms of Sections 1 and 2 above provided that you also do one of the following:

- (a) Accompany it with the complete corresponding machine-readable source code, which must be distributed under the terms of Sections 1 and 2 above on a medium customarily used for software interchange; or,

- (b) Accompany it with a written offer, valid for at least three years, to give any third party, for a charge no more than your cost of physically performing source distribution, a complete machine-readable copy of the corresponding source code, to be distributed under the terms of Sections 1 and 2 above on a medium customarily used for software interchange; or,
- (c) Accompany it with the information you received as to the offer to distribute corresponding source code. (This alternative is allowed only for noncommercial distribution and only if you received the program in object code or executable form with such an offer, in accord with Subsubsection b above.)

The source code for a work means the preferred form of the work for making modifications to it. For an executable work, complete source code means all the source code for all modules it contains, plus any associated interface definition files, plus the scripts used to control compilation and installation of the executable. However, as a special exception, the source code distributed need not include anything that is normally distributed (in either source or binary form) with the major components (compiler, kernel, and so on) of the operating system on which the executable runs, unless that component itself accompanies the executable.

If distribution of executable or object code is made by offering access to copy from a designated place, then offering equivalent access to copy the source code from the same place counts as distribution of the source code, even though third parties are not compelled to copy the source along with the object code.

4. You may not copy, modify, sublicense, or distribute the Program except as expressly provided under this License. Any attempt otherwise to copy, modify, sublicense or distribute the Program is void, and will automatically terminate your rights under this License. However, parties who have received copies, or rights, from you under this License will not have their licenses terminated so long as such parties remain in full compliance.
5. You are not required to accept this License, since you have not signed it. However, nothing else grants you permission to modify or distribute the Program or its derivative works. These actions are prohibited by law if you do not accept this License. Therefore, by modifying or distributing the Program (or any work based on the Program), you indicate your acceptance of this License to do so, and all its terms and conditions for copying, distributing or modifying the Program or works based on it.
6. Each time you redistribute the Program (or any work based on the Program), the recipient automatically receives a license from the original licensor to copy, distribute or modify the Program subject to these terms and conditions. You may not impose any further restrictions on the recipients' exercise of the rights granted herein. You are not responsible for enforcing compliance by third parties to this License.
7. If, as a consequence of a court judgment or allegation of patent infringement or for any other reason (not limited to patent issues), conditions are imposed on you (whether by court order, agreement or otherwise) that contradict the conditions of this License, they do not excuse you from the conditions of this License. If you cannot distribute so as to satisfy simultaneously your obligations under this License and any other pertinent obligations, then as a consequence you may not distribute the Program at all. For example, if a patent license would not permit royalty-free redistribution of the Program by all those who receive copies directly or indirectly through you, then the only way you could satisfy both it and this License would be to refrain entirely from distribution of the Program.

If any portion of this section is held invalid or unenforceable under any particular circumstance, the balance of the section is intended to apply and the section as a whole is intended to apply in other circumstances.

It is not the purpose of this section to induce you to infringe any patents or other property right claims or to contest validity of any such claims; this section has the sole purpose of protecting the integrity of the free software distribution system, which is implemented by public license practices. Many people have made generous contributions to the wide range of software distributed through that system in reliance on consistent application of that system; it is up to the author/donor to decide if he or she is willing to distribute software through any other system and a licensee cannot impose that choice.

This section is intended to make thoroughly clear what is believed to be a consequence of the rest of this License.

8. If the distribution and/or use of the Program is restricted in certain countries either by patents or by copyrighted interfaces, the original copyright holder who places the Program under this License may add an explicit geographical distribution limitation excluding those countries, so that distribution is permitted only in or among countries not thus excluded. In such case, this License incorporates the limitation as if written in the body of this License.
9. The Free Software Foundation may publish revised and/or new versions of the General Public License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns.

Each version is given a distinguishing version number. If the Program specifies a version number of this License which applies to it and “any later version”, you have the option of following the terms and conditions either of that version or of any later version published by the Free Software Foundation. If the Program does not specify a version number of this License, you may choose any version ever published by the Free Software Foundation.

10. If you wish to incorporate parts of the Program into other free programs whose distribution conditions are different, write to the author to ask for permission. For software which is copyrighted by the Free Software Foundation, write to the Free Software Foundation; we sometimes make exceptions for this. Our decision will be guided by the two goals of preserving the free status of all derivatives of our free software and of promoting the sharing and reuse of software generally.

9.2.3 No Warranty

10. Because the program is licensed free of charge, there is no warranty for the program, to the extent permitted by applicable law. Except when otherwise stated in writing the copyright holders and/or other parties provide the program “as is” without warranty of any kind, either expressed or implied, including, but not limited to, the implied warranties of merchantability and fitness for a particular purpose. The entire risk as to the quality and performance of the program is with you. Should the program prove defective, you assume the cost of all necessary servicing, repair or correction.
11. In no event unless required by applicable law or agreed to in writing will any copyright holder, or any other party who may modify and/or redistribute the program as permitted above, be liable to you for damages, including any general, special, incidental or consequential damages arising out of the use or inability to use the program (including but not limited to loss of data or data being rendered inaccurate or losses sustained by you or third parties or a failure of the program to operate with any other programs), even if such holder or other party has been advised of the possibility of such damages.

9.3 The L^AT_EX Project Public License, Version 1.3c 2006-05-20

9.3.1 Preamble

The L^AT_EX Project Public License (LPPL) is the primary license under which the L^AT_EX kernel and the base L^AT_EX packages are distributed.

You may use this license for any work of which you hold the copyright and which you wish to distribute. This license may be particularly suitable if your work is T_EX-related (such as a L^AT_EX package), but it is written in such a way that you can use it even if your work is unrelated to T_EX.

The section ‘WHETHER AND HOW TO DISTRIBUTE WORKS UNDER THIS LICENSE’, below, gives instructions, examples, and recommendations for authors who are considering distributing their works under this license.

This license gives conditions under which a work may be distributed and modified, as well as conditions under which modified versions of that work may be distributed.

We, the L^AT_EX3 Project, believe that the conditions below give you the freedom to make and distribute modified versions of your work that conform with whatever technical specifications you wish while maintaining the availability, integrity, and reliability of that work. If you do not see how to achieve your goal while meeting these conditions, then read the document ‘cfgguide.tex’ and ‘modguide.tex’ in the base L^AT_EX distribution for suggestions.

9.3.2 Definitions

In this license document the following terms are used:

Work Any work being distributed under this License.

Derived Work Any work that under any applicable law is derived from the Work.

Modification Any procedure that produces a Derived Work under any applicable law – for example, the production of a file containing an original file associated with the Work or a significant portion of such a file, either verbatim or with modifications and/or translated into another language.

Modify To apply any procedure that produces a Derived Work under any applicable law.

Distribution Making copies of the Work available from one person to another, in whole or in part. Distribution includes (but is not limited to) making any electronic components of the Work accessible by file transfer protocols such as FTP or HTTP or by shared file systems such as Sun's Network File System (NFS).

Compiled Work A version of the Work that has been processed into a form where it is directly usable on a computer system. This processing may include using installation facilities provided by the Work, transformations of the Work, copying of components of the Work, or other activities. Note that modification of any installation facilities provided by the Work constitutes modification of the Work.

Current Maintainer A person or persons nominated as such within the Work. If there is no such explicit nomination then it is the 'Copyright Holder' under any applicable law.

Base Interpreter A program or process that is normally needed for running or interpreting a part or the whole of the Work.

A Base Interpreter may depend on external components but these are not considered part of the Base Interpreter provided that each external component clearly identifies itself whenever it is used interactively. Unless explicitly specified when applying the license to the Work, the only applicable Base Interpreter is a ' \LaTeX -Format' or in the case of files belonging to the ' \TeX -format' a program implementing the ' \TeX language'.

9.3.3 Conditions on Distribution and Modification

1. Activities other than distribution and/or modification of the Work are not covered by this license; they are outside its scope. In particular, the act of running the Work is not restricted and no requirements are made concerning any offers of support for the Work.
2. You may distribute a complete, unmodified copy of the Work as you received it. Distribution of only part of the Work is considered modification of the Work, and no right to distribute such a Derived Work may be assumed under the terms of this clause.
3. You may distribute a Compiled Work that has been generated from a complete, unmodified copy of the Work as distributed under Clause 2 above, as long as that Compiled Work is distributed in such a way that the recipients may install the Compiled Work on their system exactly as it would have been installed if they generated a Compiled Work directly from the Work.
4. If you are the Current Maintainer of the Work, you may, without restriction, modify the Work, thus creating a Derived Work. You may also distribute the Derived Work without restriction, including Compiled Works generated from the Derived Work. Derived Works distributed in this manner by the Current Maintainer are considered to be updated versions of the Work.
5. If you are not the Current Maintainer of the Work, you may modify your copy of the Work, thus creating a Derived Work based on the Work, and compile this Derived Work, thus creating a Compiled Work based on the Derived Work.
6. If you are not the Current Maintainer of the Work, you may distribute a Derived Work provided the following conditions are met for every component of the Work unless that component clearly states in the copyright notice that it is exempt from that condition. Only the Current Maintainer is allowed to add such statements of exemption to a component of the Work.

- (a) If a component of this Derived Work can be a direct replacement for a component of the Work when that component is used with the Base Interpreter, then, wherever this component of the Work identifies itself to the user when used interactively with that Base Interpreter, the replacement component of this Derived Work clearly and unambiguously identifies itself as a modified version of this component to the user when used interactively with that Base Interpreter.
 - (b) Every component of the Derived Work contains prominent notices detailing the nature of the changes to that component, or a prominent reference to another file that is distributed as part of the Derived Work and that contains a complete and accurate log of the changes.
 - (c) No information in the Derived Work implies that any persons, including (but not limited to) the authors of the original version of the Work, provide any support, including (but not limited to) the reporting and handling of errors, to recipients of the Derived Work unless those persons have stated explicitly that they do provide such support for the Derived Work.
 - (d) You distribute at least one of the following with the Derived Work:
 - i. A complete, unmodified copy of the Work; if your distribution of a modified component is made by offering access to copy the modified component from a designated place, then offering equivalent access to copy the Work from the same or some similar place meets this condition, even though third parties are not compelled to copy the Work along with the modified component;
 - ii. Information that is sufficient to obtain a complete, unmodified copy of the Work.
7. If you are not the Current Maintainer of the Work, you may distribute a Compiled Work generated from a Derived Work, as long as the Derived Work is distributed to all recipients of the Compiled Work, and as long as the conditions of Clause 6, above, are met with regard to the Derived Work.
8. The conditions above are not intended to prohibit, and hence do not apply to, the modification, by any method, of any component so that it becomes identical to an updated version of that component of the Work as it is distributed by the Current Maintainer under Clause 4, above.
9. Distribution of the Work or any Derived Work in an alternative format, where the Work or that Derived Work (in whole or in part) is then produced by applying some process to that format, does not relax or nullify any sections of this license as they pertain to the results of applying that process.
10. (a) A Derived Work may be distributed under a different license provided that license itself honors the conditions listed in Clause 6 above, in regard to the Work, though it does not have to honor the rest of the conditions in this license.
- (b) If a Derived Work is distributed under a different license, that Derived Work must provide sufficient documentation as part of itself to allow each recipient of that Derived Work to honor the restrictions in Clause 6 above, concerning changes from the Work.
11. This license places no restrictions on works that are unrelated to the Work, nor does this license place any restrictions on aggregating such works with the Work by any means.
12. Nothing in this license is intended to, or may be used to, prevent complete compliance by all parties with all applicable laws.

9.3.4 No Warranty

There is no warranty for the Work. Except when otherwise stated in writing, the Copyright Holder provides the Work ‘as is’, without warranty of any kind, either expressed or implied, including, but not limited to, the implied warranties of merchantability and fitness for a particular purpose. The entire risk as to the quality and performance of the Work is with you. Should the Work prove defective, you assume the cost of all necessary servicing, repair, or correction.

In no event unless required by applicable law or agreed to in writing will The Copyright Holder, or any author named in the components of the Work, or any other party who may distribute and/or modify the Work as permitted above, be liable to you for damages, including any general, special, incidental or consequential damages arising out of any use of the Work or out of inability to use the Work (including, but not limited to, loss of data, data being rendered inaccurate, or losses sustained by anyone as a result of any failure of the Work to operate with any other programs), even if the Copyright Holder or said author or said other party has been advised of the possibility of such damages.

9.3.5 Maintenance of The Work

The Work has the status ‘author-maintained’ if the Copyright Holder explicitly and prominently states near the primary copyright notice in the Work that the Work can only be maintained by the Copyright Holder or simply that it is ‘author-maintained’.

The Work has the status ‘maintained’ if there is a Current Maintainer who has indicated in the Work that they are willing to receive error reports for the Work (for example, by supplying a valid e-mail address). It is not required for the Current Maintainer to acknowledge or act upon these error reports.

The Work changes from status ‘maintained’ to ‘unmaintained’ if there is no Current Maintainer, or the person stated to be Current Maintainer of the work cannot be reached through the indicated means of communication for a period of six months, and there are no other significant signs of active maintenance.

You can become the Current Maintainer of the Work by agreement with any existing Current Maintainer to take over this role.

If the Work is unmaintained, you can become the Current Maintainer of the Work through the following steps:

1. Make a reasonable attempt to trace the Current Maintainer (and the Copyright Holder, if the two differ) through the means of an Internet or similar search.
2. If this search is successful, then enquire whether the Work is still maintained.
 - (a) If it is being maintained, then ask the Current Maintainer to update their communication data within one month.
 - (b) If the search is unsuccessful or no action to resume active maintenance is taken by the Current Maintainer, then announce within the pertinent community your intention to take over maintenance. (If the Work is a L^AT_EX work, this could be done, for example, by posting to `comp.text.tex`.)
3. (a) If the Current Maintainer is reachable and agrees to pass maintenance of the Work to you, then this takes effect immediately upon announcement.
(b) If the Current Maintainer is not reachable and the Copyright Holder agrees that maintenance of the Work be passed to you, then this takes effect immediately upon announcement.
4. If you make an ‘intention announcement’ as described in 2b above and after three months your intention is challenged neither by the Current Maintainer nor by the Copyright Holder nor by other people, then you may arrange for the Work to be changed so as to name you as the (new) Current Maintainer.
5. If the previously unreachable Current Maintainer becomes reachable once more within three months of a change completed under the terms of 3b or 4, then that Current Maintainer must become or remain the Current Maintainer upon request provided they then update their communication data within one month.

A change in the Current Maintainer does not, of itself, alter the fact that the Work is distributed under the LPPL license.

If you become the Current Maintainer of the Work, you should immediately provide, within the Work, a prominent and unambiguous statement of your status as Current Maintainer. You should also announce your new status to the same pertinent community as in 2b above.

9.3.6 Whether and How to Distribute Works under This License

This section contains important instructions, examples, and recommendations for authors who are considering distributing their works under this license. These authors are addressed as ‘you’ in this section.

9.3.7 Choosing This License or Another License

If for any part of your work you want or need to use *distribution* conditions that differ significantly from those in this license, then do not refer to this license anywhere in your work but, instead, distribute your work under a different license. You may use the text of this license as a model for your own license, but your license should not refer to the LPPL or otherwise give the impression that your work is distributed under the LPPL.

The document ‘modguide.tex’ in the base L^AT_EX distribution explains the motivation behind the conditions of this license. It explains, for example, why distributing L^AT_EX under the GNU General Public

License (GPL) was considered inappropriate. Even if your work is unrelated to L^AT_EX, the discussion in ‘modguide.tex’ may still be relevant, and authors intending to distribute their works under any license are encouraged to read it.

9.3.8 A Recommendation on Modification Without Distribution

It is wise never to modify a component of the Work, even for your own personal use, without also meeting the above conditions for distributing the modified component. While you might intend that such modifications will never be distributed, often this will happen by accident – you may forget that you have modified that component; or it may not occur to you when allowing others to access the modified version that you are thus distributing it and violating the conditions of this license in ways that could have legal implications and, worse, cause problems for the community. It is therefore usually in your best interest to keep your copy of the Work identical with the public one. Many works provide ways to control the behavior of that work without altering any of its licensed components.

9.3.9 How to Use This License

To use this license, place in each of the components of your work both an explicit copyright notice including your name and the year the work was authored and/or last substantially modified. Include also a statement that the distribution and/or modification of that component is constrained by the conditions in this license.

Here is an example of such a notice and statement:

```
%% pig.dtx
%% Copyright 2005 M. Y. Name
%
% This work may be distributed and/or modified under the
% conditions of the LaTeX Project Public License, either version 1.3
% of this license or (at your option) any later version.
% The latest version of this license is in
%   http://www.latex-project.org/lppl.txt
% and version 1.3 or later is part of all distributions of LaTeX
% version 2005/12/01 or later.
%
% This work has the LPPL maintenance status 'maintained'.
%
% The Current Maintainer of this work is M. Y. Name.
%
% This work consists of the files pig.dtx and pig.ins
% and the derived file pig.sty.
```

Given such a notice and statement in a file, the conditions given in this license document would apply, with the ‘Work’ referring to the three files ‘pig.dtx’, ‘pig.ins’, and ‘pig.sty’ (the last being generated from ‘pig.dtx’ using ‘pig.ins’), the ‘Base Interpreter’ referring to any ‘L^AT_EX-Format’, and both ‘Copyright Holder’ and ‘Current Maintainer’ referring to the person ‘M. Y. Name’.

If you do not want the Maintenance section of LPPL to apply to your Work, change ‘maintained’ above into ‘author-maintained’. However, we recommend that you use ‘maintained’ as the Maintenance section was added in order to ensure that your Work remains useful to the community even when you can no longer maintain and support it yourself.

9.3.10 Derived Works That Are Not Replacements

Several clauses of the LPPL specify means to provide reliability and stability for the user community. They therefore concern themselves with the case that a Derived Work is intended to be used as a (compatible or incompatible) replacement of the original Work. If this is not the case (e.g., if a few lines of code are reused for a completely different task), then clauses 6b and 6d shall not apply.

9.3.11 Important Recommendations

Defining What Constitutes the Work The LPPL requires that distributions of the Work contain all the files of the Work. It is therefore important that you provide a way for the licensee to determine which

files constitute the Work. This could, for example, be achieved by explicitly listing all the files of the Work near the copyright notice of each file or by using a line such as:

```
% This work consists of all files listed in manifest.txt.
```

in that place. In the absence of an unequivocal list it might be impossible for the licensee to determine what is considered by you to comprise the Work and, in such a case, the licensee would be entitled to make reasonable conjectures as to which files comprise the Work.

9.4 GNU Free Documentation License, Version 1.2, November 2002

9.4.1 Preamble

The purpose of this License is to make a manual, textbook, or other functional and useful document “free” in the sense of freedom: to assure everyone the effective freedom to copy and redistribute it, with or without modifying it, either commercially or noncommercially. Secondarily, this License preserves for the author and publisher a way to get credit for their work, while not being considered responsible for modifications made by others.

This License is a kind of “copyleft”, which means that derivative works of the document must themselves be free in the same sense. It complements the GNU General Public License, which is a copyleft license designed for free software.

We have designed this License in order to use it for manuals for free software, because free software needs free documentation: a free program should come with manuals providing the same freedoms that the software does. But this License is not limited to software manuals; it can be used for any textual work, regardless of subject matter or whether it is published as a printed book. We recommend this License principally for works whose purpose is instruction or reference.

9.4.2 Applicability and definitions

This License applies to any manual or other work, in any medium, that contains a notice placed by the copyright holder saying it can be distributed under the terms of this License. Such a notice grants a world-wide, royalty-free license, unlimited in duration, to use that work under the conditions stated herein. The “**Document**”, below, refers to any such manual or work. Any member of the public is a licensee, and is addressed as “**you**”. You accept the license if you copy, modify or distribute the work in a way requiring permission under copyright law.

A “**Modified Version**” of the Document means any work containing the Document or a portion of it, either copied verbatim, or with modifications and/or translated into another language.

A “**Secondary Section**” is a named appendix or a front-matter section of the Document that deals exclusively with the relationship of the publishers or authors of the Document to the Document’s overall subject (or to related matters) and contains nothing that could fall directly within that overall subject. (Thus, if the Document is in part a textbook of mathematics, a Secondary Section may not explain any mathematics.) The relationship could be a matter of historical connection with the subject or with related matters, or of legal, commercial, philosophical, ethical or political position regarding them.

The “**Invariant Sections**” are certain Secondary Sections whose titles are designated, as being those of Invariant Sections, in the notice that says that the Document is released under this License. If a section does not fit the above definition of Secondary then it is not allowed to be designated as Invariant. The Document may contain zero Invariant Sections. If the Document does not identify any Invariant Sections then there are none.

The “**Cover Texts**” are certain short passages of text that are listed, as Front-Cover Texts or Back-Cover Texts, in the notice that says that the Document is released under this License. A Front-Cover Text may be at most 5 words, and a Back-Cover Text may be at most 25 words.

A “**Transparent**” copy of the Document means a machine-readable copy, represented in a format whose specification is available to the general public, that is suitable for revising the document straightforwardly with generic text editors or (for images composed of pixels) generic paint programs or (for drawings) some widely available drawing editor, and that is suitable for input to text formatters or for automatic translation to a variety of formats suitable for input to text formatters. A copy made in an otherwise Transparent file format whose markup, or absence of markup, has been arranged to thwart or discourage subsequent modification by readers is not Transparent. An image format is not Transparent if used for any substantial amount of text. A copy that is not “Transparent” is called “**Opaque**”.

Examples of suitable formats for Transparent copies include plain ASCII without markup, Texinfo input format, LaTeX input format, SGML or XML using a publicly available DTD, and standard-conforming simple HTML, PostScript or PDF designed for human modification. Examples of transparent image formats include PNG, XCF and JPG. Opaque formats include proprietary formats that can be read and edited only by proprietary word processors, SGML or XML for which the DTD and/or processing tools are not generally available, and the machine-generated HTML, PostScript or PDF produced by some word processors for output purposes only.

The “**Title Page**” means, for a printed book, the title page itself, plus such following pages as are needed to hold, legibly, the material this License requires to appear in the title page. For works in formats which do not have any title page as such, “Title Page” means the text near the most prominent appearance of the work’s title, preceding the beginning of the body of the text.

A section “**Entitled XYZ**” means a named subunit of the Document whose title either is precisely XYZ or contains XYZ in parentheses following text that translates XYZ in another language. (Here XYZ stands for a specific section name mentioned below, such as “**Acknowledgements**”, “**Dedications**”, “**Endorsements**”, or “**History**”.) To “**Preserve the Title**” of such a section when you modify the Document means that it remains a section “Entitled XYZ” according to this definition.

The Document may include Warranty Disclaimers next to the notice which states that this License applies to the Document. These Warranty Disclaimers are considered to be included by reference in this License, but only as regards disclaiming warranties: any other implication that these Warranty Disclaimers may have is void and has no effect on the meaning of this License.

9.4.3 Verbatim Copying

You may copy and distribute the Document in any medium, either commercially or noncommercially, provided that this License, the copyright notices, and the license notice saying this License applies to the Document are reproduced in all copies, and that you add no other conditions whatsoever to those of this License. You may not use technical measures to obstruct or control the reading or further copying of the copies you make or distribute. However, you may accept compensation in exchange for copies. If you distribute a large enough number of copies you must also follow the conditions in section 3.

You may also lend copies, under the same conditions stated above, and you may publicly display copies.

9.4.4 Copying in Quantity

If you publish printed copies (or copies in media that commonly have printed covers) of the Document, numbering more than 100, and the Document’s license notice requires Cover Texts, you must enclose the copies in covers that carry, clearly and legibly, all these Cover Texts: Front-Cover Texts on the front cover, and Back-Cover Texts on the back cover. Both covers must also clearly and legibly identify you as the publisher of these copies. The front cover must present the full title with all words of the title equally prominent and visible. You may add other material on the covers in addition. Copying with changes limited to the covers, as long as they preserve the title of the Document and satisfy these conditions, can be treated as verbatim copying in other respects.

If the required texts for either cover are too voluminous to fit legibly, you should put the first ones listed (as many as fit reasonably) on the actual cover, and continue the rest onto adjacent pages.

If you publish or distribute Opaque copies of the Document numbering more than 100, you must either include a machine-readable Transparent copy along with each Opaque copy, or state in or with each Opaque copy a computer-network location from which the general network-using public has access to download using public-standard network protocols a complete Transparent copy of the Document, free of added material. If you use the latter option, you must take reasonably prudent steps, when you begin distribution of Opaque copies in quantity, to ensure that this Transparent copy will remain thus accessible at the stated location until at least one year after the last time you distribute an Opaque copy (directly or through your agents or retailers) of that edition to the public.

It is requested, but not required, that you contact the authors of the Document well before redistributing any large number of copies, to give them a chance to provide you with an updated version of the Document.

9.4.5 Modifications

You may copy and distribute a Modified Version of the Document under the conditions of sections 2 and 3 above, provided that you release the Modified Version under precisely this License, with the Modified

Version filling the role of the Document, thus licensing distribution and modification of the Modified Version to whoever possesses a copy of it. In addition, you must do these things in the Modified Version:

- A. Use in the Title Page (and on the covers, if any) a title distinct from that of the Document, and from those of previous versions (which should, if there were any, be listed in the History section of the Document). You may use the same title as a previous version if the original publisher of that version gives permission.
- B. List on the Title Page, as authors, one or more persons or entities responsible for authorship of the modifications in the Modified Version, together with at least five of the principal authors of the Document (all of its principal authors, if it has fewer than five), unless they release you from this requirement.
- C. State on the Title page the name of the publisher of the Modified Version, as the publisher.
- D. Preserve all the copyright notices of the Document.
- E. Add an appropriate copyright notice for your modifications adjacent to the other copyright notices.
- F. Include, immediately after the copyright notices, a license notice giving the public permission to use the Modified Version under the terms of this License, in the form shown in the Addendum below.
- G. Preserve in that license notice the full lists of Invariant Sections and required Cover Texts given in the Document's license notice.
- H. Include an unaltered copy of this License.
- I. Preserve the section Entitled "History", Preserve its Title, and add to it an item stating at least the title, year, new authors, and publisher of the Modified Version as given on the Title Page. If there is no section Entitled "History" in the Document, create one stating the title, year, authors, and publisher of the Document as given on its Title Page, then add an item describing the Modified Version as stated in the previous sentence.
- J. Preserve the network location, if any, given in the Document for public access to a Transparent copy of the Document, and likewise the network locations given in the Document for previous versions it was based on. These may be placed in the "History" section. You may omit a network location for a work that was published at least four years before the Document itself, or if the original publisher of the version it refers to gives permission.
- K. For any section Entitled "Acknowledgements" or "Dedications", Preserve the Title of the section, and preserve in the section all the substance and tone of each of the contributor acknowledgements and/or dedications given therein.
- L. Preserve all the Invariant Sections of the Document, unaltered in their text and in their titles. Section numbers or the equivalent are not considered part of the section titles.
- M. Delete any section Entitled "Endorsements". Such a section may not be included in the Modified Version.
- N. Do not retitle any existing section to be Entitled "Endorsements" or to conflict in title with any Invariant Section.
- O. Preserve any Warranty Disclaimers.

If the Modified Version includes new front-matter sections or appendices that qualify as Secondary Sections and contain no material copied from the Document, you may at your option designate some or all of these sections as invariant. To do this, add their titles to the list of Invariant Sections in the Modified Version's license notice. These titles must be distinct from any other section titles.

You may add a section Entitled "Endorsements", provided it contains nothing but endorsements of your Modified Version by various parties—for example, statements of peer review or that the text has been approved by an organization as the authoritative definition of a standard.

You may add a passage of up to five words as a Front-Cover Text, and a passage of up to 25 words as a Back-Cover Text, to the end of the list of Cover Texts in the Modified Version. Only one passage of Front-Cover Text and one of Back-Cover Text may be added by (or through arrangements made by) any

one entity. If the Document already includes a cover text for the same cover, previously added by you or by arrangement made by the same entity you are acting on behalf of, you may not add another; but you may replace the old one, on explicit permission from the previous publisher that added the old one.

The author(s) and publisher(s) of the Document do not by this License give permission to use their names for publicity for or to assert or imply endorsement of any Modified Version.

9.4.6 Combining Documents

You may combine the Document with other documents released under this License, under the terms defined in section 4 above for modified versions, provided that you include in the combination all of the Invariant Sections of all of the original documents, unmodified, and list them all as Invariant Sections of your combined work in its license notice, and that you preserve all their Warranty Disclaimers.

The combined work need only contain one copy of this License, and multiple identical Invariant Sections may be replaced with a single copy. If there are multiple Invariant Sections with the same name but different contents, make the title of each such section unique by adding at the end of it, in parentheses, the name of the original author or publisher of that section if known, or else a unique number. Make the same adjustment to the section titles in the list of Invariant Sections in the license notice of the combined work.

In the combination, you must combine any sections Entitled “History” in the various original documents, forming one section Entitled “History”; likewise combine any sections Entitled “Acknowledgements”, and any sections Entitled “Dedications”. You must delete all sections Entitled “Endorsements”.

9.4.7 Collection of Documents

You may make a collection consisting of the Document and other documents released under this License, and replace the individual copies of this License in the various documents with a single copy that is included in the collection, provided that you follow the rules of this License for verbatim copying of each of the documents in all other respects.

You may extract a single document from such a collection, and distribute it individually under this License, provided you insert a copy of this License into the extracted document, and follow this License in all other respects regarding verbatim copying of that document.

9.4.8 Aggregating with independent Works

A compilation of the Document or its derivatives with other separate and independent documents or works, in or on a volume of a storage or distribution medium, is called an “aggregate” if the copyright resulting from the compilation is not used to limit the legal rights of the compilation’s users beyond what the individual works permit. When the Document is included in an aggregate, this License does not apply to the other works in the aggregate which are not themselves derivative works of the Document.

If the Cover Text requirement of section 3 is applicable to these copies of the Document, then if the Document is less than one half of the entire aggregate, the Document’s Cover Texts may be placed on covers that bracket the Document within the aggregate, or the electronic equivalent of covers if the Document is in electronic form. Otherwise they must appear on printed covers that bracket the whole aggregate.

9.4.9 Translation

Translation is considered a kind of modification, so you may distribute translations of the Document under the terms of section 4. Replacing Invariant Sections with translations requires special permission from their copyright holders, but you may include translations of some or all Invariant Sections in addition to the original versions of these Invariant Sections. You may include a translation of this License, and all the license notices in the Document, and any Warranty Disclaimers, provided that you also include the original English version of this License and the original versions of those notices and disclaimers. In case of a disagreement between the translation and the original version of this License or a notice or disclaimer, the original version will prevail.

If a section in the Document is Entitled “Acknowledgements”, “Dedications”, or “History”, the requirement (section 4) to Preserve its Title (section 1) will typically require changing the actual title.

9.4.10 Termination

You may not copy, modify, sublicense, or distribute the Document except as expressly provided for under this License. Any other attempt to copy, modify, sublicense or distribute the Document is void, and

will automatically terminate your rights under this License. However, parties who have received copies, or rights, from you under this License will not have their licenses terminated so long as such parties remain in full compliance.

9.4.11 Future Revisions of this License

The Free Software Foundation may publish new, revised versions of the GNU Free Documentation License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns. See <http://www.gnu.org/copyleft/>.

Each version of the License is given a distinguishing version number. If the Document specifies that a particular numbered version of this License “or any later version” applies to it, you have the option of following the terms and conditions either of that specified version or of any later version that has been published (not as a draft) by the Free Software Foundation. If the Document does not specify a version number of this License, you may choose any version ever published (not as a draft) by the Free Software Foundation.

9.4.12 Addendum: How to use this License for your documents

To use this License in a document you have written, include a copy of the License in the document and put the following copyright and license notices just after the title page:

Copyright ©YEAR YOUR NAME. Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Free Documentation License, Version 1.2 or any later version published by the Free Software Foundation; with no Invariant Sections, no Front-Cover Texts, and no Back-Cover Texts. A copy of the license is included in the section entitled “GNU Free Documentation License”.

If you have Invariant Sections, Front-Cover Texts and Back-Cover Texts, replace the “with … Texts” line with this:

with the Invariant Sections being LIST THEIR TITLES, with the Front-Cover Texts being LIST, and with the Back-Cover Texts being LIST.

If you have Invariant Sections without Cover Texts, or some other combination of the three, merge those two alternatives to suit the situation.

If your document contains nontrivial examples of program code, we recommend releasing these examples in parallel under your choice of free software license, such as the GNU General Public License, to permit their use in free software.

10 Supported Formats

\TeX was designed to be a flexible system. This is true both for the *input* for \TeX as well as for the *output*. The present section explains which input formats there are and how they are supported by PGF. It also explains which different output formats can be produced.

10.1 Supported Input Formats: L^AT_EX, Plain T_EX, ConT_EXt

\TeX does not prescribe exactly how your input should be formatted. While it is *customary* that, say, an opening brace starts a scope in \TeX , this is by no means necessary. Likewise, it is *customary* that environments start with `\begin{}`, but \TeX could not really care less about the exact command name.

Even though \TeX can be reconfigured, users can not. For this reason, certain *input formats* specify a set of commands and conventions how input for \TeX should be formatted. There are currently three “major” formats: Donald Knuth’s original *plain T_EX* format, Leslie Lamport’s popular L^AT_EX format, and Hans Hagen’s ConT_EXt format.

10.1.1 Using the L^AT_EX Format

Using PGF and TikZ with the L^AT_EX format is easy: You say `\usepackage{pgf}` or `\usepackage{tikz}`. Usually, that is all you need to do, all configuration will be done automatically and (hopefully) correctly.

The style files used for the L^AT_EX format reside in the subdirectory `latex/pgf/` of the PGF-system. Mainly, what these files do is to include files in the directory `generic/pgf`. For example, here is the content of the file `latex/pgf/frontends/tikz.sty`:

```
% Copyright 2006 by Till Tantau
%
% This file may be distributed and/or modified
%
% 1. under the LaTeX Project Public License and/or
% 2. under the GNU Public License.
%
% See the file doc/generic/pgf/licenses/LICENSE for more details.

\RequirePackage{pgf,pgffor}

\input{tikz.code.tex}

\endinput
```

The files in the `generic/pgf` directory do the actual work.

10.1.2 Using the Plain T_EX Format

When using the plain T_EX format, you say `\input{pgf.tex}` or `\input{tikz.tex}`. Then, instead of `\begin{pgfpicture}` and `\end{pgfpicture}` you use `\pgfpicture` and `\endpgfpicture`.

Unlike for the L^AT_EX format, PGF is not as good at discerning the appropriate configuration for the plain T_EX format. In particular, it can only automatically determine the correct output format if you use `pdftex` or `tex` plus `dvips`. For all other output formats you need to set the macro `\pgfsysdriver` to the correct value. See the description of using output formats later on.

Like the L^AT_EX style files, the plain T_EX files like `tikz.tex` also just include the correct `tikz.code.tex` file.

10.1.3 Using the ConT_EXt Format

When using the ConT_EXt format, you say `\usemodule{pgf}` or `\usemodule{tikz}`. As for the plain T_EX format you also have to replace the start- and end-of-environment tags as follows: Instead of `\begin{pgfpicture}` and `\end{pgfpicture}` you use `\startpgfpicture` and `\stoppgfpicture`; similarly, instead of `\begin{tikzpicture}` and `\end{tikzpicture}` you use must now use `\starttikzpicture` and `\stoptikzpicture`; and so on for other environments.

The ConT_EXt support is very similar to the plain T_EX support, so the same restrictions apply: You may have to set the output format directly and graphics inclusion may be a problem.

In addition to `pgf` and `tikz` there also exist modules like `pgfcore` or `pgfmatrix`. To use them, you may need to include the module `pgfmod` first (the modules `pgf` and `tikz` both include `pgfmod` for you,

so typically you can skip this). This special module is necessary since old versions of ConTeXt MkII before 2005 satanically restricted the length of module names to 8 characters and PGF's long names are mapped to cryptic 6-letter-names for you by the module `pgfmod`. This restriction was never in place in ConTeXt MkIV and the `pgfmod` module can be safely ignored nowadays.

10.2 Supported Output Formats

An output format is a format in which TeX outputs the text it has typeset. Producing the output is (conceptually) a two-stage process:

1. TeX typesets your text and graphics. The result of this typesetting is mainly a long list of letter–coordinate pairs, plus (possibly) some “special” commands. This long list of pairs is written to something called a `.dvi`-file (informally known as “device-independent file”).
2. Some other program reads this `.dvi`-file and translates the letter–coordinate pairs into, say, PostScript commands for placing the given letter at the given coordinate.

The classical example of this process is the combination of `latex` and `dvips`. The `latex` program (which is just the `tex` program called with the L^AT_EX-macros preinstalled) produces a `.dvi`-file as its output. The `dvips` program takes this output and produces a `.ps`-file (a PostScript file). Possibly, this file is further converted using, say, `ps2pdf`, whose name is supposed to mean “PostScript to PDF”. Another example of programs using this process is the combination of `tex` and `dvipdfm`. The `dvipdfm` program takes a `.dvi`-file as input and translates the letter–coordinate pairs therein into PDF-commands, resulting in a `.pdf` file directly. Finally, the `tex4ht` is also a program that takes a `.dvi`-file and produces an output, this time it is a `.html` file. The programs `pdftex` and `pdflatex` are special: They directly produce a `.pdf`-file without the intermediate `.dvi`-stage. However, from the programmer's point of view they behave exactly as if there was an intermediate stage.

Normally, TeX only produces letter–coordinate pairs as its “output”. This obviously makes it difficult to draw, say, a curve. For this, “special” commands can be used. Unfortunately, these special commands are not the same for the different programs that process the `.dvi`-file. Indeed, every program that takes a `.dvi`-file as input has a totally different syntax for the special commands.

One of the main jobs of PGF is to “abstract away” the difference in the syntax of the different programs. However, this means that support for each program has to be “programmed”, which is a time-consuming and complicated process.

10.2.1 Selecting the Backend Driver

When TeX typesets your document, it does not know which program you are going to use to transform the `.dvi`-file. If your `.dvi`-file does not contain any special commands, this would be fine; but these days almost all `.dvi`-files contain lots of special commands. It is thus necessary to tell TeX which program you are going to use later on.

Unfortunately, there is no “standard” way of telling this to TeX. For the L^AT_EX format a sophisticated mechanism exists inside the `graphics` package and PGF plugs into this mechanism. For other formats and when this plugging does not work as expected, it is necessary to tell PGF directly which program you are going to use. This is done by redefining the macro `\pgfsysdriver` to an appropriate value *before* you load `pgf`. If you are going to use the `dvips` program, you set this macro to the value `pgfsys-dvips.def`; if you use `pdftex` or `pdflatex`, you set it to `pgfsys-pdftex.def`; and so on. In the following, details of the support of the different programs are discussed.

10.2.2 Producing PDF Output

PGF supports three programs that produce PDF output (PDF means “portable document format” and was invented by the Adobe company): `dvipdfm`, `pdftex`, and `vtex`. The `pdflatex` program is the same as the `pdftex` program: it uses a different input format, but the output is exactly the same.

File `pgfsys-pdftex.def`

This is the driver file for use with pdTeX, that is, with the `pdftex` or `pdflatex` command. It includes `pgfsys-common-pdf.def`.

This driver has a lot of functionality. (Almost) everything PGF “can do at all” is implemented in this driver.

File `pgfsys-dvipdfm.def`

This is a driver file for use with `(1a)tex` followed by `dvipdfm`. It includes `pgfsys-common-pdf.def`.

This driver supports most of PGF's features, but there are some restrictions:

1. In L^AT_EX mode it uses `graphicx` for the graphics inclusion and does not support masking.
2. In plain T_EX mode it does not support image inclusion.

File `pgfsys-xetex.def`

This is a driver file for use with `xe(1a)tex` followed by `xdvipdfmx`. This driver supports largely the same operations as the `dvipdfm` driver.

File `pgfsys-vtex.def`

This is the driver file for use with the commercial VTEX program. Even though it produces PDF output, it includes `pgfsys-common-postscript.def`. Note that the VTEX program can produce *both* Postscript and PDF output, depending on the command line parameters. However, whether you produce Postscript or PDF output does not change anything with respect to the driver.

This driver supports most of PGF's features, except for the following restrictions:

1. In L^AT_EX mode it uses `graphicx` for the graphics inclusion and does not support masking.
2. In plain T_EX mode it does not support image inclusion.
3. Shading is fully implemented, but yields the same quality as the implementation for `dvips`.
4. Opacity is not supported.
5. Remembering of pictures (inter-picture connections) is not supported.

It is also possible to produce a `.pdf`-file by first producing a PostScript file (see below) and then using a PostScript-to-PDF conversion program like `ps2pdf` or the Acrobat Distiller.

10.2.3 Producing PostScript Output

File `pgfsys-dvips.def`

This is a driver file for use with `(1a)tex` followed by `dvips`. It includes `pgfsys-common-postscript.def`.

This driver also supports most of PGF's features, except for the following restrictions:

1. In L^AT_EX mode it uses `graphicx` for the graphics inclusion and does not support masking.
2. In plain T_EX mode it does not support image inclusion.
3. Shading is fully implemented, but the results will not be as good as with a driver producing `.pdf` as output.
4. Opacity works only in conjunction with newer versions of Ghostscript.
5. For remembering of pictures (inter-picture connections) you need to use a recent version of `pdftex` running in DVI-mode.

File `pgfsys-textures.def`

This is a driver file for use with the TEXTURES program. It includes `pgfsys-common-postscript.def`.

This driver has exactly the same restrictions as the driver for `dvips`.

You can also use the `vtex` program together with `pgfsys-vtex.def` to produce Postscript output.

10.2.4 Producing SVG Output

File `pgfsys-dvisvgm.def`

This driver converts DVI files to SVG file, including text and fonts. When you select this driver, PGF will output the required raw SVG code for the pictures it produces.

Since the `graphics` package does not (yet?) support this driver directly, there is special rule for this driver in L^AT_EX: If the option `dvisvgm` is given to the `tikz` package, this driver gets selected (normally, the driver selected by `graphics` would be used). For packages like `beamer` that load PGF themselves, this means that the option `dvisvgm` should be given to the document class.

```
% example.tex
\documentclass[dvisvgm]{minimal}

\usepackage{tikz}

\begin{document}
Hello \tikz [baseline] \fill [fill=blue!80!black] (0,.75ex) circle[radius=.75ex];
\end{document}
```

And then run

```
latex example
dvisvgm example
```

or better

```
lualatex --output-format=dvi example
dvisvgm example
```

(This is “better” since it gives you access to the full power of LuaTeX inside your TeX-file. In particular, TikZ is able to run graph drawing algorithms in this case.)

Unlike the `tex4ht` driver below, this driver has full support of text nodes.

File `pgfsys-tex4ht.def`

This is a driver file for use with the `tex4ht` program. It is selected automatically when the `tex4ht` style or command is loaded. It includes `pgfsys-common-svg.def`.

The `tex4ht` program converts `.dvi`-files to `.html`-files. While the HTML-format cannot be used to draw graphics, the SVG-format can. This driver will ask PGF to produce an SVG-picture for each PGF graphic in your text.

When using this driver you should be aware of the following restrictions:

1. In L^AT_EX mode it uses `graphicx` for the graphics inclusion.
2. In plain TeX mode it does not support image inclusion.
3. Remembering of pictures (inter-picture connections) is not supported.
4. Text inside `pgfpicture`s is not supported very well. The reason is that the SVG specification currently does not support text very well and, although it is possible to “escape back” to HTML, TikZ has then to guess what size the text rendered by the browser would have.
5. Unlike for other output formats, the bounding box of a picture “really crops” the picture.
6. Matrices do not work.
7. Functional shadings are not supported.

The driver basically works as follows: When a `{pgfpicture}` is started, appropriate `\special` commands are used to direct the output of `tex4ht` to a new file called `\jobname-xxx.svg`, where `xxx` is a number that is increased for each graphic. Then, till the end of the picture, each (system layer) graphic command creates a special that inserts appropriate SVG literal text into the output file. The exact details are a bit complicated since the imaging model and the processing model of PostScript/PDF and SVG are not quite the same; but they are “close enough” for PGF’s purposes.

Because text is not supported very well in the SVG standard, you may wish to use the following options to modify the way text is handled:

`/pgf/tex4ht node/escape=<boolean>` (default `false`)

Selects the rendering method for a text node with the `tex4ht` driver.

When this key is set to `false`, text is translated into SVG text, which is somewhat limited: simple characters (letters, numerals, punctuation, \sum , \int , ...), subscripts and superscripts (but not sub-subscripts) will display but everything else will be filtered out, ignored or will produce invalid HTML code (in the worst case). This means that two kind of texts render reasonably well:

1. First, plain text without math mode, special characters or anything else special.
2. Second, *very* simple mathematical text that contains subscripts or superscripts. Even then, variables are not correctly set in italics and, in general, text simple does not look very nice.

If you use text that contains anything special, even something as simple as `\alpha`, this may corrupt the graphic.

```
\tikz \node[draw,/pgf/tex4ht node/escape=false] {Example : $(a+b)^2=a^2+2ab+b^2$};
```

When you write `node[/pgf/tex4ht node/escape=true]` $\{\langle text \rangle\}$, PGF escapes back to HTML to render the $\langle text \rangle$. This method produces valid HTML code in most cases and the support for complicated text nodes is much better since code that renders well outside a `{pgfpicture}`, should also render well inside a text node. Another advantage is that inside text nodes with fixed width, HTML will produce line breaks for long lines. On the other hand, you need a browser with good SVG support to display the picture. Also, the text will display differently, depending on your browsers, the fonts you have on your system and your settings. Finally, PGF has to guess the size of the text rendered by the browser to scale it and prevent it from sticking from the node. When it fails, the text will be either cropped or too small.

```
\tikz \node[draw,/pgf/tex4ht node/escape=true]
{Example : $\int_0^\infty \frac{1}{1+t^2} dt = \frac{\pi}{2}$};
```

`/pgf/tex4ht node/css=(filename)` (default `\jobname`)

This option allows you to tell the browser what CSS file it should use to style the display of the node (only with `tex4ht node/escape=true`).

`/pgf/tex4ht node/class=(class name)` (default `foreignobject`)

This option allows you to give a class name to the node, allowing it to be styled by a CSS file (only with `tex4ht node/escape=true`).

`/pgf/tex4ht node/id=(id name)` (default `\jobname picturenumber-nodenumber`)

This option allows you to give a unique id to the node, allowing it to be styled by a CSS file (only with `tex4ht node/escape=true`).

10.2.5 Producing Perfectly Portable DVI Output

File `pgfsys-dvi.def`

This is a driver file that can be used with any output driver, except for `tex4ht`.

The driver will produce perfectly portable `.dvi` files by composing all pictures entirely of black rectangles, the basic and only graphic shape supported by the T_EX core. Even straight, but slanted lines are tricky to get right in this model (they need to be composed of lots of little squares).

Naturally, *very little* is possible with this driver. In fact, so little is possible that it is easier to list what is possible:

- Text boxes can be placed in the normal way.
- Lines and curves can be drawn (stroked). If they are not horizontal or vertical, they are composed of hundreds of small rectangles.
- Lines of different width are supported.
- Transformations are supported.

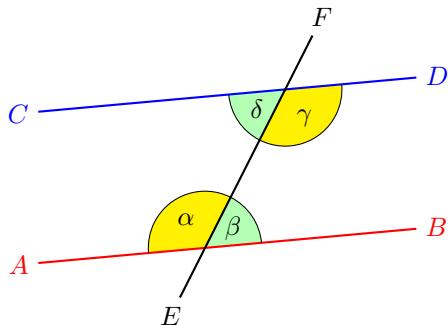
Note that, say, even filling is not supported! (Let alone color or anything fancy.)

This driver has only one real application: It might be useful when you only need horizontal or vertical lines in a picture. Then, the results are quite satisfactory.

Part III

TikZ ist *kein* Zeichenprogramm

by Till Tantau



When we assume that AB and CD are parallel, i.e., $AB \parallel CD$, then $\alpha = \gamma$ and $\beta = \delta$.

```
\usetikzlibrary {angles,calc,quotes}
\begin{tikzpicture}[angle radius=.75cm]

\node (A) at (-2,0) [red,left] {$A$};
\node (B) at ( 3,.5) [red,right] {$B$};
\node (C) at (-2,2) [blue,left] {$C$};
\node (D) at ( 3,2.5) [blue,right] {$D$};
\node (E) at (60:-5mm) [below] {$E$};
\node (F) at (60:3.5cm) [above] {$F$};

\coordinate (X) at (intersection cs:first line={(A)--(B)}, second line={(E)--(F)});
\coordinate (Y) at (intersection cs:first line={(C)--(D)}, second line={(E)--(F)});

\path
(A) edge [red, thick] (B)
(C) edge [blue, thick] (D)
(E) edge [thick] (F)
pic ["$\alpha$]", draw, fill=yellow] {angle = F--X--A}
pic ["$\beta$]", draw, fill=green!30] {angle = B--X--F}
pic ["$\gamma$]", draw, fill=yellow] {angle = E--Y--D}
pic ["$\delta$]", draw, fill=green!30] {angle = C--Y--E};

\node at ($ (D)! .5! (B) $) [right=1cm, text width=6cm, rounded corners, fill=red!20, inner sep=1ex]
{
    When we assume that $\color{red}AB$ and $\color{blue}CD$ are parallel, i.\,e., $\color{red}\mathbf{AB} \parallel \mathbf{CD}$,
    then $\alpha = \gamma$ and $\beta = \delta$.
};

\end{tikzpicture}
```

11 Design Principles

This section describes the design principles behind the *TikZ* frontend, where *TikZ* means “*TikZ ist kein Zeichenprogramm*”. To use *TikZ*, as a L^AT_EX user say `\usepackage{tikz}` somewhere in the preamble, as a plain T_EX user say `\input tikz.tex`. *TikZ*’s job is to make your life easier by providing an easy-to-learn and easy-to-use syntax for describing graphics.

The commands and syntax of *TikZ* were influenced by several sources. The basic command names and the notion of path operations is taken from METAFONT, the option mechanism comes from PSTricks, the notion of styles is reminiscent of SVG, the graph syntax is taken from GRAPHVIZ. To make it all work together, some compromises were necessary. I also added some ideas of my own, like coordinate transformations.

The following basic design principles underlie *TikZ*:

1. Special syntax for specifying points.
2. Special syntax for path specifications.
3. Actions on paths.
4. Key–value syntax for graphic parameters.
5. Special syntax for nodes.
6. Special syntax for trees.
7. Special syntax for graphs.
8. Grouping of graphic parameters.
9. Coordinate transformation system.

11.1 Special Syntax For Specifying Points

TikZ provides a special syntax for specifying points and coordinates. In the simplest case, you provide two T_EX dimensions, separated by commas, in round brackets as in `(1cm,2pt)`.

You can also specify a point in polar coordinates by using a colon instead of a comma as in `(30:1cm)`, which means “1cm in a 30 degrees direction”.

If you do not provide a unit, as in `(2,1)`, you specify a point in PGF’s *xy*-coordinate system. By default, the unit *x*-vector goes 1cm to the right and the unit *y*-vector goes 1cm upward.

By specifying three numbers as in `(1,1,1)` you specify a point in PGF’s *xyz*-coordinate system.

It is also possible to use an anchor of a previously defined shape as in `(first node.south)`.

You can add two plus signs before a coordinate as in `++(1cm,0pt)`. This means “1cm to the right of the last point used”. This allows you to easily specify relative movements. For example, `(1,0) ++(1,0) ++(0,1)` specifies the three coordinates `(1,0)`, then `(2,0)`, and `(2,1)`.

Finally, instead of two plus signs, you can also add a single one. This also specifies a point in a relative manner, but it does not “change” the current point used in subsequent relative commands. For example, `(1,0) +(1,0) +(0,1)` specifies the three coordinates `(1,0)`, then `(2,0)`, and `(1,1)`.

11.2 Special Syntax For Path Specifications

When creating a picture using *TikZ*, your main job is the specification of *paths*. A path is a series of straight or curved lines, which need not be connected. *TikZ* makes it easy to specify paths, partly using the syntax of METAPOST. For example, to specify a triangular path you use

```
(5pt,0pt) -- (0pt,0pt) -- (0pt,5pt) -- cycle
```

and you get \triangle when you draw this path.

11.3 Actions on Paths

A path is just a series of straight and curved lines, but it is not yet specified what should happen with it. One can *draw* a path, *fill* a path, *shade* it, *clip* it, or do any combination of these. Drawing (also known as *stroking*) can be thought of as taking a pen of a certain thickness and moving it along the path, thereby drawing on the canvas. Filling means that the interior of the path is filled with a uniform color. Obviously, filling makes sense only for *closed* paths and a path is automatically closed prior to filling, if necessary.

Given a path as in `\path (0,0) rectangle (2ex,1ex);`, you can draw it by adding the `draw` option as in `\path[draw] (0,0) rectangle (2ex,1ex);`, which yields . The `\draw` command is just an abbreviation for `\path[draw]`. To fill a path, use the `fill` option or the `\fill` command, which is an abbreviation for `\path[fill]`. The `\filldraw` command is an abbreviation for `\path[fill,draw]`. Shading is caused by the `shade` option (there are `\shade` and `\shadedraw` abbreviations) and clipping by the `clip` option. There is also a `\clip` command, which does the same as `\path[clip]`, but not commands like `\drawclip`. Use, say, `\draw[clip]` or `\path[draw,clip]` instead.

All of these commands can only be used inside `{tikzpicture}` environments.

TikZ allows you to use different colors for filling and stroking.

11.4 Key–Value Syntax for Graphic Parameters

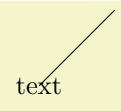
Whenever TikZ draws or fills a path, a large number of graphic parameters influences the rendering. Examples include the colors used, the dashing pattern, the clipping area, the line width, and many others. In TikZ, all these options are specified as lists of so called key–value pairs, as in `color=red`, that are passed as optional parameters to the path drawing and filling commands. This usage is similar to PSTRICKS. For example, the following will draw a thick, red triangle;



```
\tikz \draw[line width=2pt,color=red] (1,0) -- (0,0) -- (0,1) -- cycle;
```

11.5 Special Syntax for Specifying Nodes

TikZ introduces a special syntax for adding text or, more generally, nodes to a graphic. When you specify a path, add nodes as in the following example:



```
\tikz \draw (1,1) node {text} -- (2,2);
```

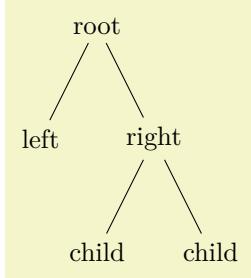
Nodes are inserted at the current position of the path, but either *after* (the default) or *before* the complete path is rendered. When special options are given, as in `\draw (1,1) node[circle,draw] {text};`, the text is not just put at the current position. Rather, it is surrounded by a circle and this circle is “drawn”.

You can add a name to a node for later reference either by using the option `name=<node name>` or by stating the node name in parentheses outside the text as in `node[circle](name){text}`.

Predefined shapes include `rectangle`, `circle`, and `ellipse`, but it is possible (though a bit challenging) to define new shapes.

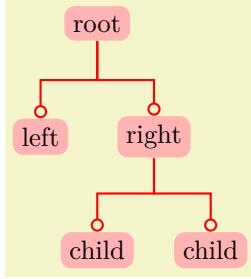
11.6 Special Syntax for Specifying Trees

The “node syntax” can also be used to draw trees: A `node` can be followed by any number of children, each introduced by the keyword `child`. The children are nodes themselves, each of which may have children in turn.

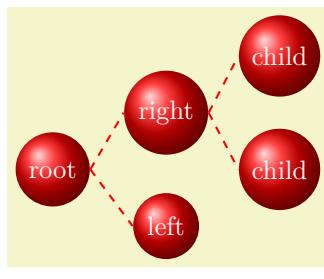


```
\begin{tikzpicture}
\node {root}
    child {node {left}}
    child {node {right}
        child {node {child}}
        child {node {child}}
    };
\end{tikzpicture}
```

Since trees are made up from nodes, it is possible to use options to modify the way trees are drawn. Here are two examples of the above tree, redrawn with different options:



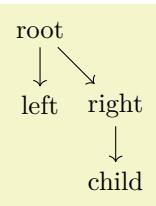
```
\usetikzlibrary {arrows.meta,trees}
\begin{tikzpicture}
[edge from parent fork down, sibling distance=15mm, level distance=15mm,
every node/.style={fill=red!30,rounded corners},
edge from parent/.style={red,-{Circle[open]},thick,draw}]
\node {root}
    child {node {left}}
    child {node {right}
        child {node {child}}
        child {node {child}}
    };
\end{tikzpicture}
```



```
\begin{tikzpicture}
[parent anchor=east,child anchor=west,grow=east,
sibling distance=15mm, level distance=15mm,
every node/.style={ball color=red,circle,text=white},
edge from parent/.style={draw,dashed,thick,red}]
\node {root}
    child {node {left}}
    child {node {right}
        child {node {child}}
        child {node {child}}
    };
\end{tikzpicture}
```

11.7 Special Syntax for Graphs

The `\node` command gives you fine control over where nodes should be placed, what text they should use, and what they should look like. However, when you draw a graph, you typically need to create numerous fairly similar nodes that only differ with respect to the name they show. In these cases, the `graph` syntax can be used, which is another syntax layer build “on top” of the node syntax.



```
\usetikzlibrary {graphs}
\tikz \graph [grow down, branch right] {
    root -> { left, right -> {child, child} }
};
```

The syntax of the `graph` command extends the so-called DOT-notation used in the popular GRAPHVIZ program.

Depending on the version of \TeX you use (it must allow you to call Lua code, which is the case for LuaTeX), you can also ask TikZ to do automatically compute good positions for the nodes of a graph using one of several integrated *graph drawing algorithms*.

11.8 Grouping of Graphic Parameters

Graphic parameters should often apply to several path drawing or filling commands. For example, we may wish to draw numerous lines all with the same line width of 1pt. For this, we put these commands

in a `{scope}` environment that takes the desired graphic options as an optional parameter. Naturally, the specified graphic parameters apply only to the drawing and filling commands inside the environment. Furthermore, nested `{scope}` environments or individual drawing commands can override the graphic parameters of outer `{scope}` environments. In the following example, three red lines, two green lines, and one blue line are drawn:



```
\begin{tikzpicture}
\begin{scope}[color=red]
\draw (0mm,10mm) -- (10mm,10mm);
\draw (0mm, 8mm) -- (10mm, 8mm);
\draw (0mm, 6mm) -- (10mm, 6mm);
\end{scope}
\begin{scope}[color=green]
\draw (0mm, 4mm) -- (10mm, 4mm);
\draw (0mm, 2mm) -- (10mm, 2mm);
\draw[color=blue] (0mm, 0mm) -- (10mm, 0mm);
\end{scope}
\end{tikzpicture}
```

The `{tikzpicture}` environment itself also behaves like a `{scope}` environment, that is, you can specify graphic parameters using an optional argument. These optional apply to all commands in the picture.

11.9 Coordinate Transformation System

TikZ supports both PGF's *coordinate* transformation system to perform transformations as well as *canvas* transformations, a more low-level transformation system. (For details on the difference between coordinate transformations and canvas transformations see Section ??.)

The syntax is set up in such a way that it is harder to use canvas transformations than coordinate transformations. There are two reasons for this: First, the canvas transformation must be used with great care and often results in "bad" graphics with changing line width and text in wrong sizes. Second, PGF loses track of where nodes and shapes are positioned when canvas transformations are used. So, in almost all circumstances, you should use coordinate transformations rather than canvas transformations.

12 Hierarchical Structures: Package, Environments, Scopes, and Styles

The present section explains how your files should be structured when you use TikZ. On the top level, you need to include the `tikz` package. In the main text, each graphic needs to be put in a `{tikzpicture}` environment. Inside these environments, you can use `{scope}` environments to create internal groups. Inside the scopes you use `\path` commands to actually draw something. On all levels (except for the package level), graphic options can be given that apply to everything within the environment.

12.1 Loading the Package and the Libraries

```
\usepackage{tikz} % \TeX
\input tikz.tex % plain \TeX
\usemodule[tikz] % Con\TeX t
```

This package does not have any options.

This will automatically load the PGF and the pgffor package.

PGF needs to know what \TeX driver you are intending to use. In most cases PGF is clever enough to determine the correct driver for you; this is true in particular if you use \LaTeX. One situation where PGF cannot know the driver “by itself” is when you use plain \TeX or Con \TeX t together with dvipdfm. In this case, you have to write `\def\pgfsysdriver{pgfsys-dvipdfm.def}` before you input `tikz.tex`.

```
\usetikzlibrary{\langle list of libraries \rangle}
```

Once TikZ has been loaded, you can use this command to load further libraries. The list of libraries should contain the names of libraries separated by commas. Instead of curly braces, you can also use square brackets, which is something Con \TeX t users will like. If you try to load a library a second time, nothing will happen.

Example: `\usetikzlibrary{arrows.meta}`

The above command will load a whole bunch of extra arrow tip definitions.

What this command does is to load the file `tikzlibrary<library>.code.tex` for each `<library>` in the `\langle list of libraries \rangle`. If this file does not exist, the file `pgflibrary<library>.code.tex` is loaded instead. If this file also does not exist, an error message is printed. Thus, to write your own library file, all you need to do is to place a file of the appropriate name somewhere where \TeX can find it. \LaTeX, plain \TeX, and Con \TeX t users can then use your library.

12.2 Creating a Picture

12.2.1 Creating a Picture Using an Environment

The “outermost” scope of TikZ is the `{tikzpicture}` environment. You may give drawing commands only inside this environment, giving them outside (as is possible in many other packages) will result in chaos.

In TikZ, the way graphics are rendered is strongly influenced by graphic options. For example, there is an option for setting the color used for drawing, another for setting the color used for filling, and also more obscure ones like the option for setting the prefix used in the filenames of temporary files written while plotting functions using an external program. The graphic options are specified in *key lists*, see Section 12.4 below for details. All graphic options are local to the `{tikzpicture}` to which they apply.

```
\begin{tikzpicture} \langle animations spec \rangle [\langle options \rangle]
  <environment contents>
\end{tikzpicture}
```

All TikZ commands should be given inside this environment, except for the `\tikzset` command. You cannot use graphics commands like the low-level command `\pgfpathmoveto` outside this environment and doing so will result in chaos. For TikZ, commands like `\path` are only defined inside this environment, so there is little chance that you will do something wrong here.

When this environment is encountered, the `\langle options \rangle` are parsed, see Section 12.4. All options given here will apply to the whole picture. Before the options you can specify animation commands, provided that the `animations` library is loaded, see Section 26 for details.

Next, the contents of the environment is processed and the graphic commands therein are put into a box. Non-graphic text is suppressed as well as possible, but non-PGF commands inside a `{tikzpicture}` environment should not produce any “output” since this may totally scramble the positioning system of the backend drivers. The suppressing of normal text, by the way, is done by temporarily switching the font to `\nullfont`. You can, however, “escape back” to normal TeX typesetting. This happens, for example, when you specify a node.

At the end of the environment, PGF tries to make a good guess at the size of a bounding box of the graphic and then resizes the picture box such that the box has this size. To “make its guess”, every time PGF encounters a coordinate, it updates the bounding box’s size such that it encompasses all these coordinates. This will usually give a good approximation of the bounding box, but will not always be accurate. First, the line thickness of diagonal lines is not taken into account correctly. Second, control points of a curve often lie far “outside” the curve and make the bounding box too large. In this case, you should use the `[use as bounding box]` option.

The following key influences the baseline of the resulting picture:

`/tikz/baseline=(dimension or coordinate or default)` (default `0pt`)

Normally, the lower end of the picture is put on the baseline of the surrounding text. For example, when you give the code `\tikz\draw(0,0)circle(.5ex);`, PGF will find out that the lower end of the picture is at $-.5\text{ex} - 0.2\text{pt}$ (the 0.2pt are half the line width, which is 0.4pt) and that the upper end is at $.5\text{ex} + .5\text{pt}$. Then, the lower end will be put on the baseline, resulting in the following: 

Using this option, you can specify that the picture should be raised or lowered such that the height `(dimension)` is on the baseline. For example, `\tikz[baseline=0pt]\draw(0,0)circle(.5ex);` yields  since, now, the baseline is on the height of the x -axis.

This options is often useful for “inlined” graphics as in

```
A → B $A \mathbin{\tikz[baseline] \draw[->} (0pt,.5ex) -- (3ex,.5ex);} B$
```

Instead of a `(dimension)` you can also provide a coordinate in parentheses. Then the effect is to put the baseline on the y -coordinate that the given `(coordinate)` has *at the end of the picture*. This means that, at the end of the picture, the `(coordinate)` is evaluated and then the baseline is set to the y -coordinate of the resulting point. This makes it easy to reference the y -coordinate of, say, the baseline of nodes.

```
\usetikzlibrary {shapes.misc}
Hello
\tikz [baseline=(X.base)]
\node [cross out,draw] (X) {world.};
```

```
Top align: 
Top align:
\tikz [baseline=(current bounding box.north)]
\draw (0,0) rectangle (1cm,1ex);
```

Use `baseline=default` to reset the `baseline` option to its initial configuration.

`/tikz/execute at begin picture=(code)` (no default)

This option causes `(code)` to be executed at the beginning of the picture. This option must be given in the argument of the `{tikzpicture}` environment itself since this option will not have an effect otherwise. After all, the picture has already “started” later on. The effect of multiply setting this option accumulates.

This option is mainly used in styles like the `every picture` style to execute certain code at the start of a picture.

`/tikz/execute at end picture=(code)` (no default)

This option installs `(code)` that will be executed at the end of the picture. Using this option multiple times will cause the code to accumulate. This option must also be given in the optional argument of the `{tikzpicture}` environment.



```
\usetikzlibrary {backgrounds}
\begin{tikzpicture}[execute at end picture=%
{
  \begin{pgfonlayer}{background}
    \path[fill=yellow,rounded corners]
      (current bounding box.south west) rectangle
      (current bounding box.north east);
  \end{pgfonlayer}
}
]
\node at (0,0) {X};
\node at (2,1) {Y};
\end{tikzpicture}
```

All options “end” at the end of the picture. To set an option “globally” change the following style:

`/tikz/every picture`

(style, initially empty)

This style is installed at the beginning of each picture.

```
\tikzset{every picture/.style=semithick}
```

Note that you should not use `\tikzset` to set options directly. For instance, if you want to use a line width of `1pt` by default, do not try to say `\tikzset{line width=1pt}` at the beginning of your document. This will not work since the line width is changed in many places. Instead, say

```
\tikzset{every picture/.style={line width=1pt}}
```

This will have the desired effect.

In other TeX formats, you should use the following commands instead:

```
\tikzpicture[<options>]
  <environment contents>
\endtikzpicture
```

This is the plain TeX version of the environment.

```
\starttikzpicture[<options>]
  <environment contents>
\stoptikzpicture
```

This is the ConTeXt version of the environment.

12.2.2 Creating a Picture Using a Command

The following command is an alternative to `{tikzpicture}` that is particular useful for graphics consisting of a single or few commands.

`\tikz<animations spec>[<options>]{<path commands>}`

This command places the `<path commands>` inside a `{tikzpicture}` environment. The `<path commands>` may contain paragraphs and fragile material (like verbatim text).

If there is only one path command, it need not be surrounded by curly braces, if there are several, you need to add them (this is similar to the `\foreach` statement and also to the rules in programming languages like Java or C concerning the placement of curly braces).

Example: `\tikz{\draw (0,0) rectangle (2ex,1ex);}` yields □

Example: `\tikz \draw (0,0) rectangle (2ex,1ex);` yields □

12.2.3 Handling Catcodes and the babel Package

Inside a TikZ picture, most symbols need to have the category code 12 (normal text) in order to ensure that the parser works properly. This is typically not the case when packages like `babel` are used, which change catcodes aggressively.

To solve this problem, TikZ provides a small library also called `babel` (which can, however, also be used together with any other package that globally changes category codes). What it does is to reset the

category codes at the beginning of every `{tikzpicture}` and to restore them at the beginning of every node. In almost all cases, this is exactly what you would expect and need, so I recommend to always load this library by saying `\usetikzlibrary{babel}`. For details on what, exactly, happens with the category codes, see Section 44.

12.2.4 Adding a Background

By default, pictures do not have any background, that is, they are “transparent” on all parts on which you do not draw anything. You may instead wish to have a colored background behind your picture or a black frame around it or lines above and below it or some other kind of decoration.

Since backgrounds are often not needed at all, the definition of styles for adding backgrounds has been put in the library package `backgrounds`. This package is documented in Section 45.

12.3 Using Scopes to Structure a Picture

Inside a `{tikzpicture}` environment you can create scopes using the `{scope}` environment. This environment is available only inside the `{tikzpicture}` environment, so once more, there is little chance of doing anything wrong.

12.3.1 The Scope Environment

```
\begin{scope}<animations spec>[<options>]
  <environment contents>
\end{scope}
```

All `<options>` are local to the `<environment contents>`. Furthermore, the clipping path is also local to the environment, that is, any clipping done inside the environment “ends” at its end.



```
\begin{tikzpicture}[ultra thick]
\begin{scope}[red]
  \draw (0mm,10mm) -- (10mm,10mm);
  \draw (0mm,8mm) -- (10mm,8mm);
\end{scope}
\draw (0mm,6mm) -- (10mm,6mm);
\begin{scope}[green]
  \draw (0mm,4mm) -- (10mm,4mm);
  \draw (0mm,2mm) -- (10mm,2mm);
  \draw[blue] (0mm,0mm) -- (10mm,0mm);
\end{scope}
\end{tikzpicture}
```

`/tikz/name=<scope name>` (no default)

Assigns a name to a scope reference in animations. The name is a “high-level” name that drivers do not see, so you can use spaces, number, letters, in a name, but you should *not* use any punctuation like a dot, a comma, or a colon.

The following style influences scopes:

`/tikz/every scope` (style, initially empty)

This style is installed at the beginning of every scope.

The following options are useful for scopes:

`/tikz/execute at begin scope=<code>` (no default)

This option install some code that will be executed at the beginning of the scope. This option must be given in the argument of the `{scope}` environment.

The effect applies only to the current scope, not to subscopes.

`/tikz/execute at end scope=<code>` (no default)

This option installs some code that will be executed at the end of the current scope. Using this option multiple times will cause the code to accumulate. This option must also be given in the optional argument of the `{scope}` environment.

Again, the effect applies only to the current scope, not to subscopes.

```
\scope<animations spec> [<options>]
  <environment contents>
\endscope
```

Plain TEX version of the environment.

```
\startscope<animations spec> [<options>]
  <environment contents>
\stopscope
```

ConTeXt version of the environment.

12.3.2 Shorthand for Scope Environments

There is a small library that makes using scopes a bit easier:

TikZ Library `scopes`

```
\usetikzlibrary{scopes} % LATEX and plain TEX
\usetikzlibrary[scopes] % ConTeXt
```

This library defines a shorthand for starting and ending `{scope}` environments.

When this library is loaded, the following happens: At certain places inside a TikZ picture, it is allowed to start a scope just using a single brace, provided the single brace is followed by options in square brackets:



```
\usetikzlibrary {scopes}
\begin{tikzpicture}
  [ultra thick]
  [red]
    \draw (0mm,10mm) -- (10mm,10mm);
    \draw (0mm,8mm) -- (10mm,8mm);
  }
  \draw (0mm,6mm) -- (10mm,6mm);
}
[green]
\draw (0mm,4mm) -- (10mm,4mm);
\draw (0mm,2mm) -- (10mm,2mm);
\draw[blue] (0mm,0mm) -- (10mm,0mm);
\end{tikzpicture}
```

In the above example, `{ [thick]}` actually causes a `\begin{scope}[thick]` to be inserted, and the corresponding closing `}` causes an `\end{scope}` to be inserted.

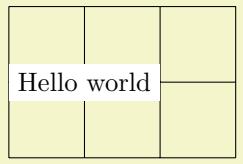
The “certain places” where an opening brace has this special meaning are the following: First, right after the semicolon that ends a path. Second, right after the end of a scope. Third, right at the beginning of a scope, which includes the beginning of a picture. Also note that some square bracket must follow, otherwise the brace is treated as a normal TEX scope.

12.3.3 Single Command Scopes

In some situations it is useful to create a scope for a single command. For instance, when you wish to use algorithm graph drawing in order to layout a tree, the path of the tree needs to be surrounded by a scope whose only purpose is to take a key that selects a layout for the scope. Similarly, in order to put something on a background layer, a scope needs to be created. In such cases, where it will be cumbersome to create a `\begin{scope}` and `\end{scope}` pair just for a single command, the `\scopeds` command may be useful:

```
\scopeds<animations spec> [<options>] <path command>
```

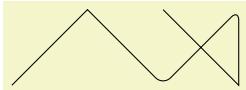
This command works like `\tikz`, only you can use it inside a `{tikzpicture}`. It will take the following `<path command>` and put it inside a `{scope}` with the `<options>` set. The `<path command>` may either be a single command ended by a semicolon or it may contain multiple commands, but then they must be surrounded by curly braces.



```
\usetikzlibrary {backgrounds}
\begin{tikzpicture}
  \node [fill=white] at (1,1) {Hello world};
  \scoped [on background layer]
    \draw (0,0) grid (3,2);
\end{tikzpicture}
```

12.3.4 Using Scopes Inside Paths

The `\path` command, which is described in much more detail in later sections, also takes graphic options. These options are local to the path. Furthermore, it is possible to create local scopes within a path simply by using curly braces as in



```
\tikz \draw (0,0) -- (1,1)
  {[rounded corners] -- (2,0) -- (3,1)}
  -- (3,0) -- (2,1);
```

Note that many options apply only to the path as a whole and cannot be scoped in this way. For example, it is not possible to scope the `color` of the path. See the explanations in the section on paths for more details.

Finally, certain elements that you specify in the argument to the `\path` command also take local options. For example, a node specification takes options. In this case, the options apply only to the node, not to the surrounding path.

12.4 Using Graphic Options

12.4.1 How Graphic Options Are Processed

Many commands and environments of TikZ accept *options*. These options are so-called *key lists*. To process the options, the following command is used, which you can also call yourself. Note that it is usually better not to call this command directly, since this will ensure that the effect of options are local to a well-defined scope.

`\tikzset{\langle options\rangle}`

This command will process the *⟨options⟩* using the `\pgfkeys` command, documented in detail in Section ??, with the default path set to `/tikz`. Under normal circumstances, the *⟨options⟩* will be lists of comma-separated pairs of the form *⟨key⟩=⟨value⟩*, but more fancy things can happen when you use the power of the `pgfkeys` mechanism, see Section ?? once more.

When a pair *⟨key⟩=⟨value⟩* is processed, the following happens:

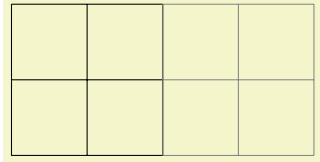
1. If the *⟨key⟩* is a full key (starts with a slash) it is handled directly as described in Section ??.
2. Otherwise (which is usually the case), it is checked whether `/tikz/⟨key⟩` is a key and, if so, it is executed.
3. Otherwise, it is checked whether `/pgf/⟨key⟩` is a key and, if so, it is executed.
4. Otherwise, it is checked whether *⟨key⟩* is a color and, if so, `color=⟨key⟩` is executed.
5. Otherwise, it is checked whether *⟨key⟩* contains a dash and, if so, `arrows=⟨key⟩` is executed.
6. Otherwise, it is checked whether *⟨key⟩* is the name of a shape and, if so, `shape=⟨key⟩` is executed.
7. Otherwise, an error message is printed.

Note that by the above description, all keys starting with `/tikz` and also all keys starting with `/pgf` can be used as *⟨key⟩*s in an *⟨options⟩* list.

12.4.2 Using Styles to Manage How Pictures Look

There is a way of organizing sets of graphic options “orthogonally” to the normal scoping mechanism. For example, you might wish all your “help lines” to be drawn in a certain way like, say, gray and thin (do *not* dash them, that distracts). For this, you can use *styles*.

A style is a key that, when used, causes a set of graphic options to be processed. Once a style has been defined, it can be used like any other key. For example, the predefined `help lines` style, which you should use for lines in the background like grid lines or construction lines.



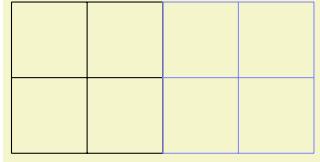
```
\begin{tikzpicture}
  \draw (0,0) grid +(2,2);
  \draw[help lines] (2,0) grid +(2,2);
\end{tikzpicture}
```

Defining styles is also done using options. Suppose we wish to define a style called `my style` and when this style is used, we want the draw color to be set to `red` and the fill color be set to `red!20`. To achieve this, we use the following option:

```
my style/.style={draw=red,fill=red!20}
```

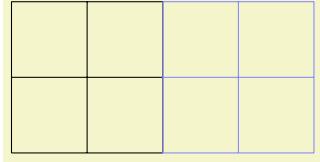
The meaning of the curious `/ .style` is the following: “The key `my style` should not be used here but, rather, be defined. So, set up things such that using the key `my style` will, in the following, have the same effect as if we had written `draw=red,fill=red!20` instead.”

Returning to the help lines example, suppose we prefer blue help lines. This could be achieved as follows:



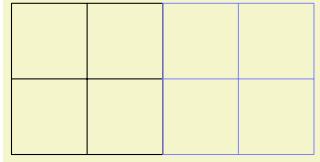
```
\begin{tikzpicture}[help lines/.style={blue!50,very thin}]
  \draw (0,0) grid +(2,2);
  \draw[help lines] (2,0) grid +(2,2);
\end{tikzpicture}
```

Naturally, one of the main ideas behind styles is that they can be used in different pictures. In this case, we have to use the `\tikzset` command somewhere at the beginning.



```
\tikzset{help lines/.style={blue!50,very thin}}
%
\begin{tikzpicture}
  \draw (0,0) grid +(2,2);
  \draw[help lines] (2,0) grid +(2,2);
\end{tikzpicture}
```

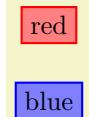
Since styles are just special cases of `pgfkeys`'s general style facility, you can actually do quite a bit more. Let us start with adding options to an already existing style. This is done using `/ .append style` instead of `/ .style`:



```
\begin{tikzpicture}[help lines/.append style=blue!50]
  \draw (0,0) grid +(2,2);
  \draw[help lines] (2,0) grid +(2,2);
\end{tikzpicture}
```

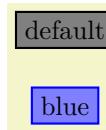
In the above example, the option `blue!50` is appended to the style `help lines`, which now has the same effect as `black!50,very thin,blue!50`. Note that two colors are set, so the last one will “win”. There also exists a handler called `/ .prefix style` that adds something at the beginning of the style.

Just as normal keys, styles can be parameterized. This means that you write `<style>=<value>` when you use the style instead of just `<style>`. In this case, all occurrences of `#1` in `<style>` are replaced by `<value>`. Here is an example that shows how this can be used.



```
\begin{tikzpicture}[outline/.style={draw=#1,thick,fill=#1!50}]
  \node [outline=red] at (0,1) {red};
  \node [outline=blue] at (0,0) {blue};
\end{tikzpicture}
```

For parameterized styles you can also set a *default* value using the `/.default` handler:



```
\begin{tikzpicture}[outline/.style={draw=#1,thick,fill=#1!50},
                  outline/.default=black]
  \node [outline]      at (0,1) {default};
  \node [outline=blue] at (0,0) {blue};
\end{tikzpicture}
```

For more details on using and setting styles, see also Section ??.

13 Specifying Coordinates

13.1 Overview

A *coordinate* is a position on the canvas on which your picture is drawn. TikZ uses a special syntax for specifying coordinates. Coordinates are always put in round brackets. The general syntax is `([<options>] <coordinate specification>)`.

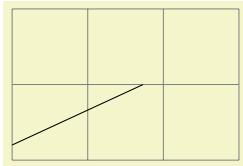
The *<coordinate specification>* specifies coordinates using one of many different possible *coordinate systems*. Examples are the Cartesian coordinate system or polar coordinates or spherical coordinates. No matter which coordinate system is used, in the end, a specific point on the canvas is represented by the coordinate.

There are two ways of specifying which coordinate system should be used:

Explicitly You can specify the coordinate system explicitly. To do so, you give the name of the coordinate system at the beginning, followed by `cs:`, which stands for “coordinate system”, followed by a specification of the coordinate using the key–value syntax. Thus, the general syntax for *<coordinate specification>* in the explicit case is `(<coordinate system> cs:<list of key–value pairs specific to the coordinate system>)`.

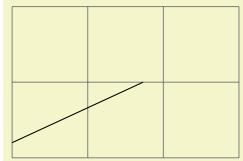
Implicitly The explicit specification is often too verbose when numerous coordinates should be given. Because of this, for the coordinate systems that you are likely to use often a special syntax is provided. TikZ will notice when you use a coordinate specified in a special syntax and will choose the correct coordinate system automatically.

Here is an example in which explicit the coordinate systems are specified explicitly:



```
\begin{tikzpicture}
  \draw[help lines] (0,0) grid (3,2);
  \draw (canvas cs:x=0cm,y=2mm)
    -- (canvas polar cs:radius=2cm,angle=30);
\end{tikzpicture}
```

In the next example, the coordinate systems are implicit:



```
\begin{tikzpicture}
  \draw[help lines] (0,0) grid (3,2);
  \draw (0cm,2mm) -- (30:2cm);
\end{tikzpicture}
```

It is possible to give options that apply only to a single coordinate, although this makes sense for transformation options only. To give transformation options for a single coordinate, give these options at the beginning in brackets:



```
\begin{tikzpicture}
  \draw[help lines] (0,0) grid (3,2);
  \draw (0,0) -- (1,1);
  \draw[red] (0,0) -- ([xshift=3pt] 1,1);
  \draw (1,0) -- +(30:2cm);
  \draw[red] (1,0) -- +([shift=(135:5pt)] 30:2cm);
\end{tikzpicture}
```

13.2 Coordinate Systems

13.2.1 Canvas, XYZ, and Polar Coordinate Systems

Let us start with the basic coordinate systems.

Coordinate system `canvas`

The simplest way of specifying a coordinate is to use the `canvas` coordinate system. You provide a dimension d_x using the `x=` option and another dimension d_y using the `y=` option. The position on the canvas is located at the position that is d_x to the right and d_y above the origin.

`/tikz/cs/x=<dimension>`

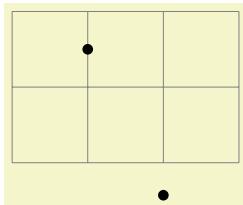
(no default, initially 0pt)

Distance by which the coordinate is to the right of the origin. You can also write things like `1cm+2pt` since the mathematical engine is used to evaluate the `<dimension>`.

`/tikz/cs/y=<dimension>`

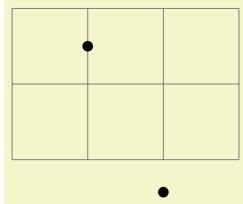
(no default, initially 0pt)

Distance by which the coordinate is above the origin.



```
\begin{tikzpicture}
  \draw[help lines] (0,0) grid (3,2);
  \fill (canvas cs:x=1cm,y=1.5cm) circle (2pt);
  \fill (canvas cs:x=2cm,y=-5mm+2pt) circle (2pt);
\end{tikzpicture}
```

To specify a coordinate in the coordinate system implicitly, you use two dimensions that are separated by a comma as in `(0cm,3pt)` or `(2cm,\textheight)`.



```
\begin{tikzpicture}
  \draw[help lines] (0,0) grid (3,2);
  \fill (1cm,1.5cm) circle (2pt);
  \fill (2cm,-5mm+2pt) circle (2pt);
\end{tikzpicture}
```

Coordinate system xyz

The `xyz` coordinate system allows you to specify a point as a multiple of three vectors called the x -, y -, and z -vectors. By default, the x -vector points 1cm to the right, the y -vector points 1cm upwards, but this can be changed arbitrarily as explained in Section 25.2. The default z -vector points to $(-3.85\text{mm}, -3.85\text{mm})$.

To specify the factors by which the vectors should be multiplied before being added, you use the following three options:

`/tikz/cs/x=<factor>`

(no default, initially 0)

Factor by which the x -vector is multiplied.

`/tikz/cs/y=<factor>`

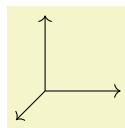
(no default, initially 0)

Works like `x`.

`/tikz/cs/z=<factor>`

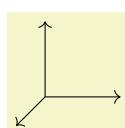
(no default, initially 0)

Works like `z`.



```
\begin{tikzpicture}[->]
  \draw (0,0) -- (xyz cs:x=1);
  \draw (0,0) -- (xyz cs:y=1);
  \draw (0,0) -- (xyz cs:z=1);
\end{tikzpicture}
```

This coordinate system can also be selected implicitly. To do so, you just provide two or three comma-separated factors (not dimensions).



```
\begin{tikzpicture}[->]
  \draw (0,0) -- (1,0);
  \draw (0,0) -- (0,1,0);
  \draw (0,0) -- (0,0,1);
\end{tikzpicture}
```

Note: It is possible to use coordinates like `(1,2cm)`, which are neither `canvas` coordinates nor `xyz` coordinates. The rule is the following: If a coordinate is of the implicit form `(⟨x⟩,⟨y⟩)`, then `⟨x⟩` and `⟨y⟩` are checked, independently, whether they have a dimension or whether they are dimensionless. If both have a dimension, the `canvas` coordinate system is used. If both lack a dimension, the `xyz` coordinate system is used. If `⟨x⟩` has a dimension and `⟨y⟩` has not, then the sum of two coordinate `(⟨x⟩,0pt)` and `(0,⟨y⟩)` is used. If `⟨y⟩` has a dimension and `⟨x⟩` has not, then the sum of two coordinate `(⟨x⟩,0)` and `(0pt,⟨y⟩)` is used.

Note furthermore: An expression like `(2+3cm,0)` does *not* mean the same as `(2cm+3cm,0)`. Instead, if `⟨x⟩` or `⟨y⟩` internally uses a mixture of dimensions and dimensionless values, then all dimensionless values are “upgraded” to dimensions by interpreting them as `pt`. So, `2+3cm` is the same dimension as `2pt+3cm`.

Coordinate system `canvas polar`

The `canvas polar` coordinate system allows you to specify polar coordinates. You provide an angle using the `angle=` option and a radius using the `radius=` option. This yields the point on the canvas that is at the given radius distance from the origin at the given degree. An angle of zero degrees to the right, a degree of 90 upward.

`/tikz/cs/angle=(degrees)` (no default)

The angle of the coordinate. The angle must always be given in degrees and should be between -360 and 720 .

`/tikz/cs/radius=(dimension)` (no default)

The distance from the origin.

`/tikz/cs/x radius=(dimension)` (no default)

A polar coordinate is, after all, just a point on a circle of the given `⟨radius⟩`. When you provide an `x-radius` and also a `y-radius`, you specify an ellipse instead of a circle. The `radius` option has the same effect as specifying identical `x radius` and `y radius` options.

`/tikz/cs/y radius=(dimension)` (no default)

Works like `x radius`.



```
\tikz \draw (0,0) -- (canvas polar cs:angle=30,radius=1cm);
```

The implicit form for canvas polar coordinates is the following: you specify the angle and the distance, separated by a colon as in `(30:1cm)`.



```
\tikz \draw (0cm,0cm) -- (30:1cm) -- (60:1cm) -- (90:1cm)
          -- (120:1cm) -- (150:1cm) -- (180:1cm);
```

Two different radii are specified by writing `(30:1cm and 2cm)`.

For the implicit form, instead of an angle given as a number you can also use certain words. For example, `up` is the same as `90`, so that you can write `\tikz \draw (0,0) --(2ex,0pt) --+(up:1ex);` and get \rightarrow . Apart from `up` you can use `down`, `left`, `right`, `north`, `south`, `west`, `east`, `north east`, `north west`, `south east`, `south west`, all of which have their natural meaning.

Coordinate system `xyz polar`

This coordinate system work similarly to the `canvas polar` system. However, the radius and the angle are interpreted in the xy -coordinate system, not in the `canvas` system. More detailed, consider the circle or ellipse whose half axes are given by the current x -vector and the current y -vector. Then, consider the point that lies at a given angle on this ellipse, where an angle of zero is the same as the x -vector and an angle of 90 is the y -vector. Finally, multiply the resulting vector by the given radius factor. Voilà.

`/tikz/cs/angle=(degrees)` (no default)

The angle of the coordinate interpreted in the ellipse whose axes are the x -vector and the y -vector.

`/tikz/cs/radius=(factor)` (no default)

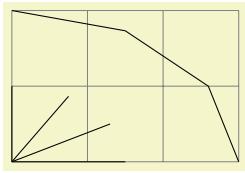
A factor by which the x -vector and y -vector are multiplied prior to forming the ellipse.

`/tikz/cs/x radius=(dimension)` (no default)

A specific factor by which only the x -vector is multiplied.

`/tikz/cs/y radius=(dimension)` (no default)

Works like `x radius`.



```
\begin{tikzpicture}[x=1.5cm,y=1cm]
\draw[help lines] (0cm,0cm) grid (3cm,2cm);

\draw (0,0) -- (xyz polar cs:angle=0,radius=1);
\draw (0,0) -- (xyz polar cs:angle=30,radius=1);
\draw (0,0) -- (xyz polar cs:angle=60,radius=1);
\draw (0,0) -- (xyz polar cs:angle=90,radius=1);

\draw (xyz polar cs:angle=0,radius=2)
-- (xyz polar cs:angle=30,radius=2)
-- (xyz polar cs:angle=60,radius=2)
-- (xyz polar cs:angle=90,radius=2);
\end{tikzpicture}
```

The implicit version of this option is the same as the implicit version of `canvas polar`, only you do not provide a unit.



```
\tikz[x={(0cm,1cm)},y={(-1cm,0cm)}]
\draw (0,0) -- (30:1) -- (60:1) -- (90:1)
-- (120:1) -- (150:1) -- (180:1);
```

Coordinate system `xy polar`

This is just an alias for `xyz polar`, which some people might prefer as there is no z -coordinate involved in the `xyz polar` coordinates.

13.2.2 Barycentric Systems

In the barycentric coordinate system a point is expressed as the linear combination of multiple vectors. The idea is that you specify vectors v_1, v_2, \dots, v_n and numbers $\alpha_1, \alpha_2, \dots, \alpha_n$. Then the barycentric coordinate specified by these vectors and numbers is

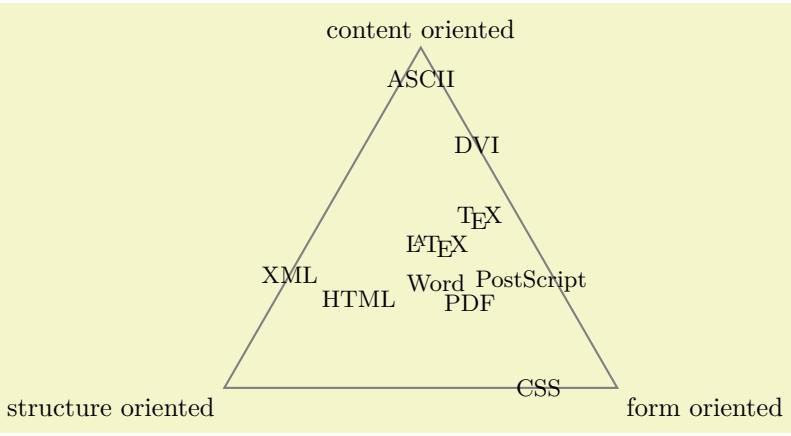
$$\frac{\alpha_1 v_1 + \alpha_2 v_2 + \cdots + \alpha_n v_n}{\alpha_1 + \alpha_2 + \cdots + \alpha_n}$$

The `barycentric cs` allows you to specify such coordinates easily.

Coordinate system `barycentric`

For this coordinate system, the `<coordinate specification>` should be a comma-separated list of expressions of the form `<node name>=<number>`. Note that (currently) the list should not contain any spaces before or after the `<node name>` (unlike normal key–value pairs).

The specified coordinate is now computed as follows: Each pair provides one vector and a number. The vector is the `center` anchor of the `<node name>`. The number is the `<number>`. Note that (currently) you cannot specify a different anchor, so that in order to use, say, the `north` anchor of a node you first have to create a new coordinate at this north anchor. (Using for instance `\coordinate(mynorth) at (mynode.north);`)



```
\begin{tikzpicture}
  \coordinate (content) at (90:3cm);
  \coordinate (structure) at (210:3cm);
  \coordinate (form) at (-30:3cm);

  \node [above] at (content) {content oriented};
  \node [below left] at (structure) {structure oriented};
  \node [below right] at (form) {form oriented};

  \draw [thick,gray] (content.south) -- (structure.north east) -- (form.north west) -- cycle;

  \small
  \node at (barycentric cs:content=0.5,structure=0.1 ,form=1) {PostScript};
  \node at (barycentric cs:content=1 ,structure=0 ,form=0.4) {DVI};
  \node at (barycentric cs:content=0.5,structure=0.5 ,form=1) {PDF};
  \node at (barycentric cs:content=0 ,structure=0.25,form=1) {CSS};
  \node at (barycentric cs:content=0.5,structure=1 ,form=0) {XML};
  \node at (barycentric cs:content=0.5,structure=1 ,form=0.4) {HTML};
  \node at (barycentric cs:content=1 ,structure=0.2 ,form=0.8) {\TeX};
  \node at (barycentric cs:content=1 ,structure=0.6 ,form=0.8) {\LaTeX};
  \node at (barycentric cs:content=0.8,structure=0.8 ,form=1) {Word};
  \node at (barycentric cs:content=1 ,structure=0.05,form=0.05) {ASCII};
\end{tikzpicture}
```

13.2.3 Node Coordinate System

In PGF and in TikZ it is quite easy to define a node that you wish to reference at a later point. Once you have defined a node, there are different ways of referencing points of the node. To do so, you use the following coordinate system:

Coordinate system `node`

This coordinate system is used to reference a specific point inside or on the border of a previously defined node. It can be used in different ways, so let us go over them one by one.

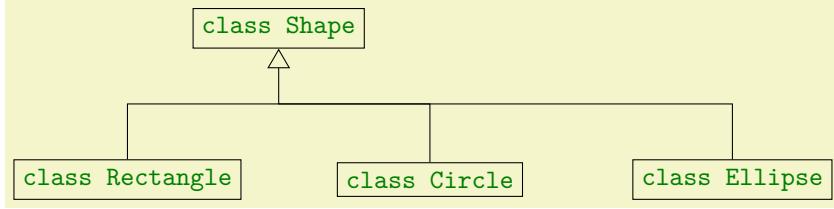
You can use three options to specify which coordinate you mean:

`/tikz/cs/name=node name` (no default)

Specifies the node that you wish to use to specify a coordinate. The *<node name>* is the name that was previously used to name the node using the `name=<node name>` option or the special node name syntax.

`/tikz/anchor=anchor` (no default)

Specifies an anchor of the node. Here is an example:



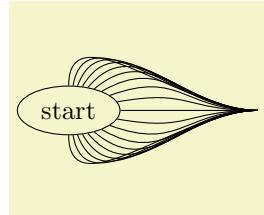
```
\usetikzlibrary {arrows.meta}
\begin{tikzpicture}
\node (shape) at (0,2) [draw] {|class Shape|};
\node (rect) at (-2,0) [draw] {|class Rectangle|};
\node (circle) at (2,0) [draw] {|class Circle|};
\node (ellipse) at (6,0) [draw] {|class Ellipse|};

\draw (node cs:name=circle,anchor=north) |- (0,1);
\draw (node cs:name=ellipse,anchor=north) |- (0,1);
\draw [arrows = {-{Triangle[open, angle=60:3mm]}}
      (node cs:name=rect,anchor=north)
      |- (0,1) -| (node cs:name=shape,anchor=south);
\end{tikzpicture}
```

`/tikz/cs/angle=<degrees>`

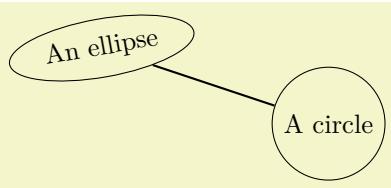
(no default)

It is also possible to provide an angle *instead* of an anchor. This coordinate refers to a point of the node's border where a ray shot from the center in the given angle hits the border. Here is an example:



```
\usetikzlibrary {shapes.geometric}
\begin{tikzpicture}
\node (start) [draw,shape=ellipse] {start};
\foreach \angle in {-90, -80, ..., 90}
  \draw (node cs:name=start,angle=\angle)
    .. controls +(\angle:1cm) and +(-1,0) .. (2.5,0);
\end{tikzpicture}
```

It is possible to provide *neither* the `anchor=` option nor the `angle=` option. In this case, TikZ will calculate an appropriate border position for you. Here is an example:



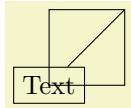
```
\usetikzlibrary {shapes.geometric}
\begin{tikzpicture}
\path (0,0) node(a) [ellipse,rotate=10,draw] {An ellipse}
      (3,-1) node(b) [circle,draw]           {A circle};
\draw[thick] (node cs:name=a) -- (node cs:name=b);
\end{tikzpicture}
```

TikZ will be reasonably clever at determining the border points that you "mean", but, naturally, this may fail in some situations. If TikZ fails to determine an appropriate border point, the center will be used instead.

Automatic computation of anchors works only with the line-to operations `--`, the vertical/horizontal versions `|-` and `-|`, and with the curve-to operation `...`. For other path commands, such as `parabola` or `plot`, the center will be used. If this is not desired, you should give a named anchor or an angle anchor.

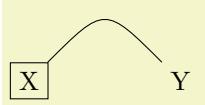
Note that if you use an automatic coordinate for both the start and the end of a line-to, as in `--(node cs:name=b)--`, then *two* border coordinates are computed with a move-to between them. This is usually exactly what you want.

If you use relative coordinates together with automatic anchor coordinates, the relative coordinates are computed relative to the node's center, not relative to the border point. Here is an example:



```
\tikz \draw (0,0) node(x) [draw] {Text}
            rectangle (1,1)
            (node cs:name=x) -- +(1,1);
```

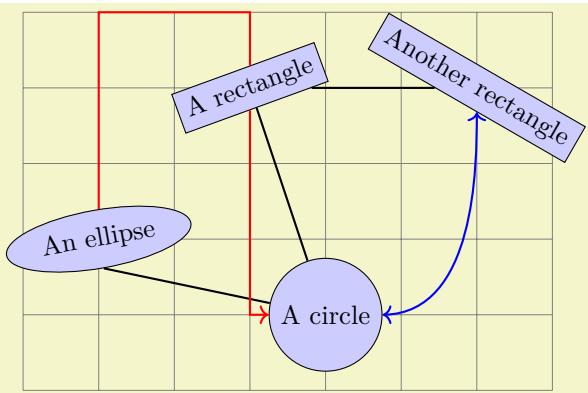
Similarly, in the following examples both control points are (1,1):



```
\tikz \draw (0,0) node(x) [draw] {X}
            (2,0) node(y) {Y}
            (node cs:name=x) .. controls +(1,1) and +(-1,1) ..
            (node cs:name=y);
```

The implicit way of specifying the node coordinate system is to simply use the name of the node in parentheses as in (a) or to specify a name together with an anchor or an angle separated by a dot as in (a.north) or (a.10).

Here is a more complete example:



```
\usetikzlibrary {shapes.geometric}
\begin{tikzpicture}[fill=blue!20]
\begin{help lines} (-1,-2) grid (6,3);
\draw[help lines] (-1,-2) grid (6,3);
\path (0,0) node(a) [ellipse,rotate=10,draw,fill] {An ellipse}
      (3,-1) node(b) [circle,draw,fill] {A circle}
      (2,2) node(c) [rectangle,rotate=20,draw,fill] {A rectangle}
      (5,2) node(d) [rectangle,rotate=-30,draw,fill] {Another rectangle};
\draw[thick] (a.south) -- (b) -- (c) -- (d);
\draw[thick,red,->] (a) |- +(1,3) -| (c) |- (b);
\draw[thick,blue,<->] (b) .. controls +(right:2cm) and +(down:1cm) .. (d);
\end{tikzpicture}
```

13.2.4 Tangent Coordinate Systems

Coordinate system `tangent`

This coordinate system, which is available only when the TikZ library `calc` is loaded, allows you to compute the point that lies tangent to a shape. In detail, consider a `<node>` and a `<point>`. Now, draw a straight line from the `<point>` so that it “touches” the `<node>` (more formally, so that it is *tangent* to this `<node>`). The point where the line touches the shape is the point referred to by the `tangent` coordinate system.

The following options may be given:

`/tikz/cs/node=<node>`

(no default)

This key specifies the node on whose border the tangent should lie.

`/tikz/cs/point=<point>`

(no default)

This key specifies the point through which the tangent should go.

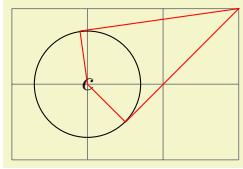
`/tikz/cs/solution=⟨number⟩`

(no default)

Specifies which solution should be used if there are more than one.

A special algorithm is needed in order to compute the tangent for a given shape. Currently, tangents can be computed for nodes whose shape is one of the following:

- `coordinate`
- `circle`



```
\usetikzlibrary {calc}
\begin{tikzpicture}
  \draw[help lines] (0,0) grid (3,2);
  \coordinate (a) at (3,2);

  \node [circle,draw] (c) at (1,1) [minimum size=40pt] {$c$};

  \draw[red] (a) -- (tangent cs:node=c,point={(a)},solution=1) --
             (c.center) -- (tangent cs:node=c,point={(a)},solution=2) -- cycle;
\end{tikzpicture}
```

There is no implicit syntax for this coordinate system.

13.2.5 Defining New Coordinate Systems

While the set of coordinate systems that TikZ can parse via their special syntax is fixed, it is possible and quite easy to define new explicitly named coordinate systems. For this, the following commands are used:

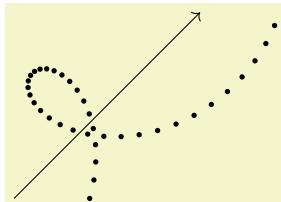
`\tikzdeclarecoordinatesystem{⟨name⟩}{⟨code⟩}`

This command declares a new coordinate system named `⟨name⟩` that can later on be used by writing `(⟨name⟩ cs:⟨arguments⟩)`. When TikZ encounters a coordinate specified in this way, the `⟨arguments⟩` are passed to `⟨code⟩` as argument #1.

It is now the job of `⟨code⟩` to make sense of the `⟨arguments⟩`. At the end of `⟨code⟩`, the two TeX dimensions `\pgf@x` and `\pgf@y` should be have the *x*- and *y*-canvas coordinate of the coordinate.

It is not necessary, but customary, to parse `⟨arguments⟩` using the key–value syntax. However, you can also parse it in any way you like.

In the following example, a coordinate system `cylindrical` is defined.



```
\makeatletter
\define@key{cylindricalkeys}{angle}{\def\myangle{#1}}
\define@key{cylindricalkeys}{radius}{\def\myradius{#1}}
\define@key{cylindricalkeys}{z}{\def\myz{#1}}
\tikzdeclarecoordinatesystem[cylindrical]{%
  %
  \setkeys{cylindricalkeys}{#1}%
  \pgfpointadd{\pgfpointxyz{0}{0}{\myz}}{\pgfpointpolarxy{\myangle}{\myradius}}%
}
\begin{tikzpicture}[z=0.2pt]
  \draw [->] (0,0,0) -- (0,0,350);
  \foreach \num in {0,10,\dots,350}
    \fill (cylindrical cs:angle=\num,radius=1,z=\num) circle (1pt);
\end{tikzpicture}
```

`\tikzaliascoordinatesystem{⟨new name⟩}{⟨old name⟩}`

Creates an alias of `⟨old name⟩`.

13.3 Coordinates at Intersections

You will wish to compute the intersection of two paths. For the special and frequent case of two perpendicular lines, a special coordinate system called `perpendicular` is available. For more general cases, the `intersection` library can be used.

13.3.1 Intersections of Perpendicular Lines

A frequent special case of path intersections is the intersection of a vertical line going through a point p and a horizontal line going through some other point q . For this situation there is a useful coordinate system.

Coordinate system `perpendicular`

You can specify the two lines using the following keys:

`/tikz/cs/horizontal line through={⟨coordinate⟩}` (no default)

Specifies that one line is a horizontal line that goes through the given coordinate.

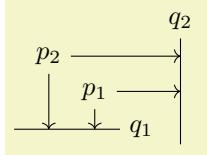
`/tikz/cs/vertical line through={⟨coordinate⟩}` (no default)

Specifies that the other line is vertical and goes through the given coordinate.

However, in almost all cases you should, instead, use the implicit syntax. Here, you write $(\langle p \rangle |- \langle q \rangle)$ or $(\langle q \rangle -| \langle p \rangle)$.

For example, $(2,1 |- 3,4)$ and $(3,4 -| 2,1)$ both yield the same as $(2,4)$ (provided the xy -coordinate system has not been modified).

The most useful application of the syntax is to draw a line up to some point on a vertical or horizontal line. Here is an example:

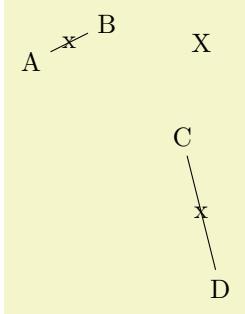


```
\begin{tikzpicture}
  \path (30:1cm) node(p1) {$p_1$} (75:1cm) node(p2) {$p_2$};
  \draw (-0.2,0) -- (1.2,0) node(xline)[right] {$q_1$};
  \draw (2,-0.2) -- (2,1.2) node(yline)[above] {$q_2$};

  \draw[->] (p1) -- (p1 |- xline);
  \draw[->] (p2) -- (p2 |- xline);
  \draw[->] (p1) -- (p1 -| yline);
  \draw[->] (p2) -- (p2 -| yline);
\end{tikzpicture}
```

Note that in $(\langle c \rangle |- \langle d \rangle)$ the coordinates $\langle c \rangle$ and $\langle d \rangle$ are *not* surrounded by parentheses. If they need to be complicated expressions (like a computation using the \$-syntax), you must surround them with braces; parentheses will then be added around them.

As an example, let us specify a point that lies horizontally at the middle of the line from A to B and vertically at the middle of the line from C to D :



```
\usetikzlibrary{calc}
\begin{tikzpicture}
  \node(A) at (0,1) {A};
  \node(B) at (1,1.5) {B};
  \node(C) at (2,0) {C};
  \node(D) at (2.5,-2) {D};

  \draw(A) -- (B) node[midway]{x};
  \draw(C) -- (D) node[midway]{x};

  \node at ({(A)!0.5!(B)} -| {(C)!0.5!(D)}) {X};
\end{tikzpicture}
```

13.3.2 Intersections of Arbitrary Paths

TikZ Library `intersections`

```
\usetikzlibrary{intersections} % LEX and plain TEX
\usetikzlibrary[intersections] % ConTEXt
```

This library enables the calculation of intersections of two arbitrary paths. However, due to the low accuracy of T^EX, the paths should not be “too complicated”. In particular, you should not try to intersect paths consisting of lots of very small segments such as plots or decorated paths.

To find the intersections of two paths in TikZ, they must be “named”. A “named path” is, quite simply, a path that has been named using the following key (note that this is a *different* key from the `name` key, which only attaches a hyperlink target to a path, but does not store the path in a way that is useful for the intersection computation):

```
/tikz/name path=<name>                                         (no default)
/tikz/name path global=<name>                                    (no default)
```

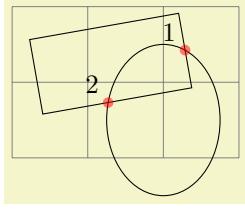
The effect of this key is that, after the path has been constructed, just before it is used, it is associated with `<name>`. For `name path`, this association survives beyond the final semi-colon of the path but not the end of the surrounding scope. For `name path global`, the association will survive beyond any scope as well. Handle with care.

Any paths created by nodes on the (main) path are ignored, unless this key is explicitly used. If the same `<name>` is used for the main path and the node path(s), then the paths will be added together and then associated with `<name>`.

To find the intersection of named paths, the following key is used:

```
/tikz/name intersections={<options>}                                     (no default)
```

This key changes the key path to `/tikz/intersection` and processes `<options>`. These options determine, among other things, which paths to use for the intersection. Having processed the options, any intersections are then found. A coordinate is created at each intersection, which by default, will be named `intersection-1`, `intersection-2`, and so on. Optionally, the prefix `intersection` can be changed, and the total number of intersections stored in a TeX-macro.



```
\usetikzlibrary {intersections}
\begin{tikzpicture} [every node/.style={opacity=1, black, above left}]
  \draw [help lines] grid (3,2);
  \draw [name path=ellipse] (2,0.5) ellipse (0.75cm and 1cm);
  \draw [name path=rectangle, rotate=10] (0.5,0.5) rectangle +(2,1);
  \fill [red, opacity=0.5, name intersections={of=ellipse and rectangle}]
    (intersection-1) circle (2pt) node {1};
  \fill [red, opacity=0.5, name intersections={of=ellipse and rectangle}]
    (intersection-2) circle (2pt) node {2};
\end{tikzpicture}
```

The following keys can be used in `<options>`:

```
/tikz/intersection/of=<name path 1>and<name path 2>                         (no default)
```

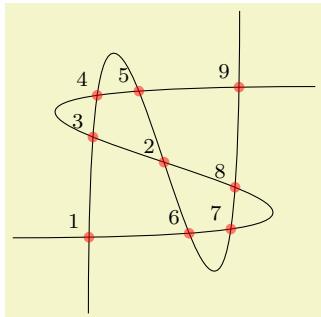
This key is used to specify the names of the paths to use for the intersection.

```
/tikz/intersection/name=<prefix>                                              (no default, initially intersection)
```

This key specifies the prefix name for the coordinate nodes placed at each intersection.

```
/tikz/intersection/total=<macro>                                            (no default)
```

This key means that the total number of intersections found will be stored in `<macro>`.



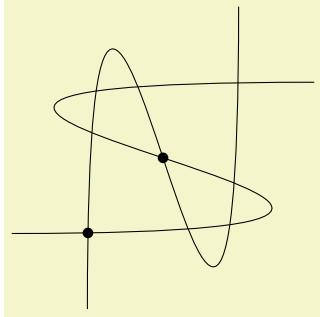
```
\usetikzlibrary {intersections}
\begin{tikzpicture}
  \clip (-2,-2) rectangle (2,2);
  \draw [name path=curve 1] (-2,-1) .. controls (8,-1) and (-8,1) .. (2,1);
  \draw [name path=curve 2] (-1,-2) .. controls (-1,8) and (1,-8) .. (1,2);

  \fill [name intersections={of=curve 1 and curve 2, name=i, total=\t}]
    [red, opacity=0.5, every node/.style={above left, black, opacity=1}]
    \foreach \s in {1,...,\t}{(i-\s) circle (2pt) node {\footnotesize\s}};
\end{tikzpicture}
```

```
/tikz/intersection/by=<comma-separated list>                                      (no default)
```

This key allows you to specify a list of names for the intersection coordinates. The intersection coordinates will still be named `<prefix>-<number>`, but additionally the first coordinate will

also be named by the first element of the `<comma-separated list>`. What happens is that the `<comma-separated list>` is passed to the `\foreach` statement and for `<list member>` a coordinate is created at the already-named intersection.

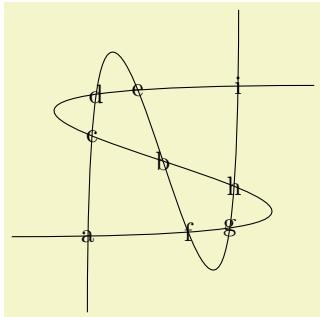


```
\usetikzlibrary {intersections}
\begin{tikzpicture}
\clip (-2,-2) rectangle (2,2);
\draw [name path=curve 1] (-2,-1) .. controls (8,-1) and (-8,1) .. (2,1);
\draw [name path=curve 2] (-1,-2) .. controls (-1,8) and (1,-8) .. (1,2);

\fill [name intersections={of=curve 1 and curve 2, by={a,b}}]
(a) circle (2pt)
(b) circle (2pt);
\end{tikzpicture}
```

You can also use the `...` notation of the `\foreach` statement inside the `<comma-separated list>`.

In case an element of the `<comma-separated list>` starts with options in square brackets, these options are used when the coordinate is created. A coordinate name can still, but need not, follow the options. This makes it easy to add labels to intersections:



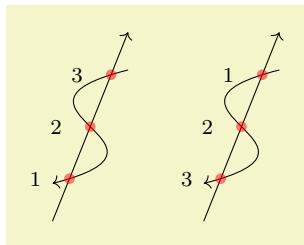
```
\usetikzlibrary {intersections}
\begin{tikzpicture}
\clip (-2,-2) rectangle (2,2);
\draw [name path=curve 1] (-2,-1) .. controls (8,-1) and (-8,1) .. (2,1);
\draw [name path=curve 2] (-1,-2) .. controls (-1,8) and (1,-8) .. (1,2);

\fill [name intersections={
    of=curve 1 and curve 2,
    by={[label=center:a],[label=center:b],[label=center:c],[label=center:d],
          [label=center:e],[label=center:f],[label=center:g],[label=center:h]}];
\end{tikzpicture}
```

`/tikz/intersection/sort by=<path name>`

(no default)

By default, the intersections are simply returned in the order that the intersection algorithm finds them. Unfortunately, this is not necessarily a “helpful” ordering. This key can be used to sort the intersections along the path specified by `<path name>`, which should be one of the paths mentioned in the `/tikz/intersection/of` key.

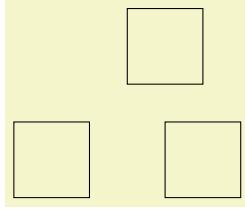


```
\usetikzlibrary {intersections}
\begin{tikzpicture}
\clip (-0.5,-0.75) rectangle (3.25,2.25);
\foreach \pathname/\shift in {line/0cm, curve/2cm} {
    \tikzset{xshift=\shift}
    \draw [->, name path=curve] (1,1.5) .. controls (-1,1) and (2,0.5) .. (0,0);
    \draw [->, name path=line] (0,-.5) -- (1,2);
    \fill [name intersections={of=line and curve,sort by=\pathname, name=i}]
        [red, opacity=0.5, every node/.style={left=.25cm, black, opacity=1}]
        \foreach \s in {1,2,3}{(i-\s) circle (2pt) node {\footnotesize\s}};
}
\end{tikzpicture}
```

13.4 Relative and Incremental Coordinates

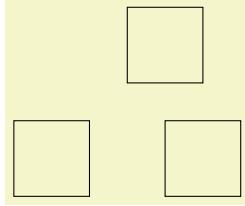
13.4.1 Specifying Relative Coordinates

You can prefix coordinates by `++` to make them “relative”. A coordinate such as `++(1cm,0pt)` means “1cm to the right of the previous position, making this the new current position”. Relative coordinates are often useful in “local” contexts:



```
\begin{tikzpicture}
\draw (0,0) -- ++(1,0) -- +(0,1) -- +(-1,0) -- cycle;
\draw (2,0) -- ++(1,0) -- +(0,1) -- +(-1,0) -- cycle;
\draw (1.5,1.5) -- ++(1,0) -- +(0,1) -- +(-1,0) -- cycle;
\end{tikzpicture}
```

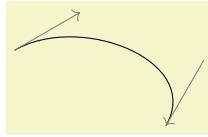
Instead of `++` you can also use a single `+`. This also specifies a relative coordinate, but it does not “update” the current point for subsequent usages of relative coordinates. Thus, you can use this notation to specify numerous points, all relative to the same “initial” point:



```
\begin{tikzpicture}
\draw (0,0) -- +(1,0) -- +(1,1) -- +(0,1) -- cycle;
\draw (2,0) -- +(1,0) -- +(1,1) -- +(0,1) -- cycle;
\draw (1.5,1.5) -- +(1,0) -- +(1,1) -- +(0,1) -- cycle;
\end{tikzpicture}
```

There is a special situation, where relative coordinates are interpreted differently. If you use a relative coordinate as a control point of a Bézier curve, the following rule applies: First, a relative first control point is taken relative to the beginning of the curve. Second, a relative second control point is taken relative to the end of the curve. Third, a relative end point of a curve is taken relative to the start of the curve.

This special behavior makes it easy to specify that a curve should “leave or arrive from a certain direction” at the start or end. In the following example, the curve “leaves” at 30° and “arrives” at 60° :



```
\begin{tikzpicture}
\draw (1,0) .. controls +(30:1cm) and +(60:1cm) .. (3,-1);
\draw[gray,>] (1,0) -- +(30:1cm);
\draw[gray,<-] (3,-1) -- +(60:1cm);
\end{tikzpicture}
```

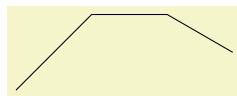
13.4.2 Rotational Relative Coordinates

You may sometimes wish to specify points relative not only to the previous point, but additionally relative to the tangent entering the previous point. For this, the following key is useful:

`/tikz/turn`

(no value)

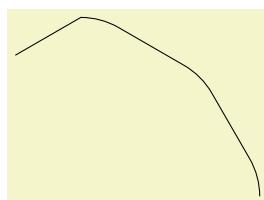
This key can be given as an option to a `<coordinate>` as in the following example:



```
\tikz \draw (0,0) -- (1,1) -- ([turn]-45:1cm) -- ([turn]-30:1cm);
```

The effect of this key is to locally shift the coordinate system so that the last point reached is at the origin and the coordinate system is “turned” so that the x -axis points in the direction of a tangent entering the last point. This means, in effect, that when you use polar coordinates of the form `<relative angle>:<distance>` together with the `turn` option, you specify a point that lies at `<distance>` from the last point in the direction of the last tangent entering the last point, but with a rotation of `<relative angle>`.

This key also works with curves ...

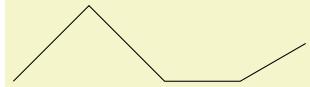


```
\tikz [delta angle=30, radius=1cm]
\draw (0,0) arc [start angle=0] -- ([turn]0:1cm)
      arc [start angle=30] -- ([turn]0:1cm)
      arc [start angle=60] -- ([turn]30:1cm);
```



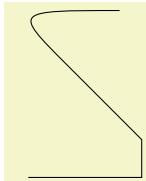
```
\tikz \draw (0,0) to [bend left] (2,1) -- ([turn]0:1cm);
```

... and with plots ...



```
\tikz \draw plot coordinates {(0,0) (1,1) (2,0) (3,0)} -- ([turn]30:1cm);
```

Although the above examples use polar coordinates with `turn`, you can also use any normal coordinate. For instance, `([turn]1,1)` will append a line of length $\sqrt{2}$ that is turns by 45° relative to the tangent to the last point.



```
\tikz \draw (0.5,0.5) -| (2,1) -- ([turn]1,1)
            .. controls ([turn]0:1cm) .. ([turn]-90:1cm);
```

13.4.3 Relative Coordinates and Scopes

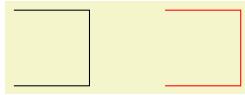
An interesting question is, how do relative coordinates behave in the presence of scopes? That is, suppose we use curly braces in a path to make part of it “local”, how does that affect the current position? On the one hand, the current position certainly changes since the scope only affects options, not the path itself. On the other hand, it may be useful to “temporarily escape” from the updating of the current point.

Since both interpretations of how the current point and scopes should “interact” are useful, there is a (`local!`) option that allows you to decide which you need.

`/tikz/current point is local=<boolean>`

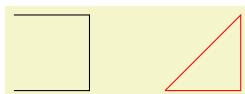
(no default, initially `false`)

Normally, the scope path operation has no effect on the current point. That is, curly braces on a path have no effect on the current position:



```
\begin{tikzpicture}
  \draw (0,0) -- +(1,0) -- +(0,1) -- +(-1,0);
  \draw[red] (2,0) -- +(1,0) { -- +(0,1) } -- +(-1,0);
\end{tikzpicture}
```

If you set this key to `true`, this behaviour changes. In this case, at the end of a group created on a path, the last current position reverts to whatever value it had at the beginning of the scope. More precisely, when TikZ encounters `}` on a path, it checks whether at this particular moment the key is set to `true`. If so, the current position reverts to the value it had when the matching `{` was read.



```
\begin{tikzpicture}
  \draw (0,0) -- +(1,0) -- +(0,1) -- +(-1,0);
  \draw[red] (2,0) -- +(1,0)
    { [current point is local] -- +(0,1) } -- +(-1,0);
\end{tikzpicture}
```

In the above example, we could also have given the option outside the scope, for instance as a parameter to the whole scope.

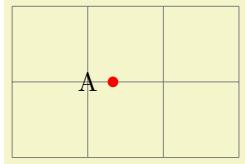
13.5 Coordinate Calculations

TikZ Library `calc`

```
\usetikzlibrary{calc} % LATEX and plain TEX
\usetikzlibrary[calc] % ConTEXt
```

You need to load this library in order to use the coordinate calculation functions described in the present section.

It is possible to do some basic calculations that involve coordinates. In essence, you can add and subtract coordinates, scale them, compute midpoints, and do projections. For instance, $(\$a) + 1/3*(1cm,0)$ is the coordinate that is 1/3cm to the right of the point a :



```
\usetikzlibrary {calc}
\begin{tikzpicture}
  \draw [help lines] (0,0) grid (3,2);
  \node (a) at (1,1) {A};
  \fill [red] ($(a) + 1/3*(1cm,0)$) circle (2pt);
\end{tikzpicture}
```

13.5.1 The General Syntax

The general syntax is the following:

$([\langle options \rangle] \$(\langle coordinate computation \rangle))$.

As you can see, the syntax uses the TeX math symbol \$ to indicate that a “mathematical computation” is involved. However, the \$ has no other effect, in particular, no mathematical text is typeset.

The $\langle coordinate computation \rangle$ has the following structure:

1. It starts with

$\langle factor \rangle * \langle coordinate \rangle \langle modifiers \rangle$

2. This is optionally followed by + or - and then another

$\langle factor \rangle * \langle coordinate \rangle \langle modifiers \rangle$

3. This is once more followed by + or - and another of the above modified coordinate; and so on.

In the following, the syntax of factors and of the different modifiers is explained in detail.

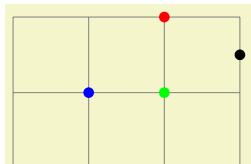
13.5.2 The Syntax of Factors

The $\langle factor \rangle$ s are optional and detected by checking whether the $\langle coordinate computation \rangle$ starts with a (. Also, after each \pm a $\langle factor \rangle$ is present if, and only if, the + or - sign is not directly followed by (.

If a $\langle factor \rangle$ is present, it is evaluated using the `\pgfmathparse` macro. This means that you can use pretty complicated computations inside a factor. A $\langle factor \rangle$ may even contain opening parentheses, which creates a complication: How does TikZ know where a $\langle factor \rangle$ ends and where a coordinate starts? For instance, if the beginning of a $\langle coordinate computation \rangle$ is $2*(3+4\dots$, it is not clear whether $3+4$ is part of a $\langle coordinate \rangle$ or part of a $\langle factor \rangle$. Because of this, the following rule is used: Once it has been determined, that a $\langle factor \rangle$ is present, in principle, the $\langle factor \rangle$ contains everything up to the next occurrence of *. Note that there is no space between the asterisk and the parenthesis.

It is permissible to put the $\langle factor \rangle$ in curly braces. This can be used whenever it is unclear where the $\langle factor \rangle$ would end.

Here are some examples of coordinate specifications that consist of exactly one $\langle factor \rangle$ and one $\langle coordinate \rangle$:



```
\usetikzlibrary {calc}
\begin{tikzpicture}
  \draw [help lines] (0,0) grid (3,2);
  \fill [red] ($2*(1,1)$) circle (2pt);
  \fill [green] (${1+1}*(1,.5)$) circle (2pt);
  \fill [blue] ($\cos(0)*\sin(90)*(1,1)$) circle (2pt);
  \fill [black] (${3*(4-3)}*(1,0.5)$) circle (2pt);
\end{tikzpicture}
```

13.5.3 The Syntax of Partway Modifiers

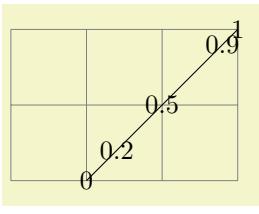
A $\langle coordinate \rangle$ can be followed by different $\langle modifiers \rangle$. The first kind of modifier is the *partway modifier*. The syntax (which is loosely inspired by Uwe Kern's `xcolor` package) is the following:

$\langle coordinate \rangle ! \langle number \rangle ! \langle angle \rangle : \langle second coordinate \rangle$

One could write for instance

$(1,2)! .75!(3,4)$

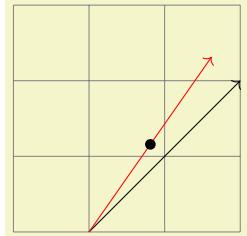
The meaning of this is: “Use the coordinate that is three quarters on the way from $(1,2)$ to $(3,4)$.” In general, $\langle coordinate x \rangle ! \langle number \rangle ! \langle coordinate y \rangle$ yields the coordinate $(1 - \langle number \rangle) \langle coordinate x \rangle + \langle number \rangle \langle coordinate y \rangle$. Note that this is a bit different from the way the $\langle number \rangle$ is interpreted in the `xcolor` package: First, you use a factor between 0 and 1, not a percentage, and, second, as the $\langle number \rangle$ approaches 1, we approach the second coordinate, not the first. It is permissible to use a $\langle number \rangle$ that is smaller than 0 or larger than 1. The $\langle number \rangle$ is evaluated using the `\pgfmathparse` command and, thus, it can involve complicated computations.



```
\usetikzlibrary {calc}
\begin{tikzpicture}
  \draw [help lines] (0,0) grid (3,2);
  \draw (1,0) -- (3,2);
  \foreach \i in {0,0.2,0.5,0.9,1}
    \node at ($(1,0)! \i!(3,2)$) {\i};
\end{tikzpicture}
```

The $\langle second coordinate \rangle$ may be prefixed by an $\langle angle \rangle$, separated with a colon, as in $(1,1)! .5!60:(2,2)$. The general meaning of $\langle a \rangle ! \langle factor \rangle ! \langle angle \rangle : \langle b \rangle$ is: “First, consider the line from $\langle a \rangle$ to $\langle b \rangle$. Then rotate this line by $\langle angle \rangle$ around the point $\langle a \rangle$. Then the two endpoints of this line will be $\langle a \rangle$ and some point $\langle c \rangle$. Use this point $\langle c \rangle$ for the subsequent computation, namely the partway computation.”

Here are two examples:



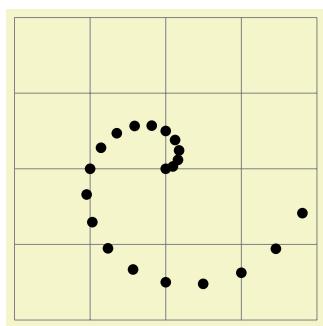
```
\usetikzlibrary {calc}
\begin{tikzpicture}
  \draw [help lines] (0,0) grid (3,3);
  \coordinate (a) at (1,0);
  \coordinate (b) at (3,2);

  \draw[->] (a) -- (b);

  \coordinate (c) at ($ (a)!1! 10:(b) $);

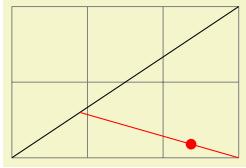
  \draw[->,red] (a) -- (c);

  \fill ($ (a)! .5! 10:(b) $) circle (2pt);
\end{tikzpicture}
```



```
\usetikzlibrary {calc}
\begin{tikzpicture}
  \draw [help lines] (0,0) grid (4,4);
  \foreach \i in {0,0.1,...,2}
    \fill ($ (2,2) ! \i! \i*180:(3,2) $) circle (2pt);
\end{tikzpicture}
```

You can repeatedly apply modifiers. That is, after any modifier you can add another (possibly different) modifier.



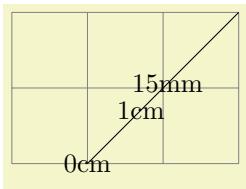
```
\usetikzlibrary {calc}
\begin{tikzpicture}
  \draw [help lines] (0,0) grid (3,2);
  \draw (0,0) -- (3,2);
  \draw[red] ($ (0,0) ! .3! (3,2) $) -- (3,0);
  \fill[red] ($ (0,0) ! .3! (3,2) ! .7! (3,0) $) circle (2pt);
\end{tikzpicture}
```

13.5.4 The Syntax of Distance Modifiers

A *distance modifier* has nearly the same syntax as a partway modifier, only you use a $\langle dimension \rangle$ (something like 1cm) instead of a $\langle factor \rangle$ (something like 0.5):

$$\langle coordinate \rangle ! \langle dimension \rangle ! \langle angle \rangle : \langle second coordinate \rangle$$

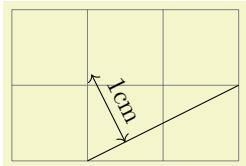
When you write $\langle a \rangle ! \langle dimension \rangle ! \langle b \rangle$, this means the following: Use the point that is distanced $\langle dimension \rangle$ from $\langle a \rangle$ on the straight line from $\langle a \rangle$ to $\langle b \rangle$. Here is an example:



```
\usetikzlibrary {calc}
\begin{tikzpicture}
  \draw [help lines] (0,0) grid (3,2);
  \draw (1,0) -- (3,2);
  \foreach \i in {0cm,1cm,15mm}
    \node at ($ (1,0) ! \i ! (3,2) $) {\i};
\end{tikzpicture}
```

As before, if you use a $\langle angle \rangle$, the $\langle second coordinate \rangle$ is rotated by this much around the $\langle coordinate \rangle$ before it is used.

The combination of an $\langle angle \rangle$ of 90 degrees with a distance can be used to “offset” a point relative to a line. Suppose, for instance, that you have computed a point (c) that lies somewhere on a line from (a) to (b) and you now wish to offset this point by 1cm so that the distance from this offset point to the line is 1cm . This can be achieved as follows:



```
\usetikzlibrary {calc}
\begin{tikzpicture}
  \draw [help lines] (0,0) grid (3,2);
  \coordinate (a) at (1,0);
  \coordinate (b) at (3,1);

  \draw (a) -- (b);

  \coordinate (c) at ($ (a) ! .25! (b) $);
  \coordinate (d) at ($ (c) ! 1cm! 90:(b) $);

  \draw [<->] (c) -- (d) node [sloped,midway,above] {1cm};
\end{tikzpicture}
```

13.5.5 The Syntax of Projection Modifiers

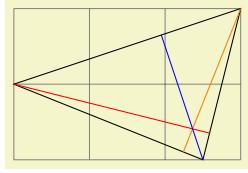
The projection modifier is also similar to the above modifiers: It also gives a point on a line from the $\langle coordinate \rangle$ to the $\langle second coordinate \rangle$. However, the $\langle number \rangle$ or $\langle dimension \rangle$ is replaced by a $\langle projection coordinate \rangle$:

$$\langle coordinate \rangle ! \langle projection coordinate \rangle ! \langle angle \rangle : \langle second coordinate \rangle$$

Here is an example:

```
(1,2) ! (0,5) ! (3,4)
```

The effect is the following: We project the $\langle projection coordinate \rangle$ orthogonally onto the line from $\langle coordinate \rangle$ to $\langle second coordinate \rangle$. This makes it easy to compute projected points:



```
\usetikzlibrary {calc}
\begin{tikzpicture}[help lines]
\draw [help lines] (0,0) grid (3,2);
\coordinate (a) at (0,1);
\coordinate (b) at (3,2);
\coordinate (c) at (2.5,0);

\draw (a) -- (b) -- (c) -- cycle;
\draw[red] (a) -- ($(b)!(a)!(c)$);
\draw[orange] (b) -- ($(a)!(b)!(c)$);
\draw[blue] (c) -- ($(a)!(c)!(b)$);
\end{tikzpicture}
```

14 Syntax for Path Specifications

A *path* is a series of straight and curved line segments. It is specified following a `\path` command and the specification must follow a special syntax, which is described in the subsections of the present section.

`\path<specification>;`

This command is available only inside a `{tikzpicture}` environment.

The *<specification>* is a long stream of *path operations*. Most of these path operations tell TikZ how the path is built. For example, when you write `--(0,0)`, you use a *line-to operation* and it means “continue the path from wherever you are to the origin”.

At any point where TikZ expects a path operation, you can also give some graphic options, which is a list of options in brackets, such as `[rounded corners]`. These options can have different effects:

1. Some options take “immediate” effect and apply to all subsequent path operations on the path. For example, the `rounded corners` option will round all following corners, but not the corners “before” and if the `sharp corners` is given later on the path (in a new set of brackets), the rounding effect will end.



```
\tikz \draw (0,0) -- (1,1)
           [rounded corners] -- (2,0) -- (3,1)
           [sharp corners] -- (3,0) -- (2,1);
```

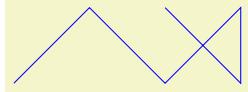
Another example are the transformation options, which also apply only to subsequent coordinates.

2. The options that have immediate effect can be “scoped” by putting part of a path in curly braces. For example, the above example could also be written as follows:



```
\tikz \draw (0,0) -- (1,1)
           {[rounded corners]} -- (2,0) -- (3,1)
           -- (3,0) -- (2,1);
```

3. Some options only apply to the path as a whole. For example, the `color=` option for determining the color used for, say, drawing the path always applies to all parts of the path. If several different colors are given for different parts of the path, only the last one (on the outermost scope) “wins”:



```
\tikz \draw (0,0) -- (1,1)
           [color=red] -- (2,0) -- (3,1)
           [color=blue] -- (3,0) -- (2,1);
```

Most options are of this type. In the above example, we would have had to “split up” the path into several `\path` commands:



```
\tikz \draw (0,0) -- (1,1);
       \draw [color:red] (2,0) -- (3,1);
       \draw [color=blue] (3,0) -- (2,1);
```

By default, the `\path` command does “nothing” with the path, it just “throws it away”. Thus, if you write `\path(0,0)--(1,1);`, nothing is drawn in your picture. The only effect is that the area occupied by the picture is (possibly) enlarged so that the path fits inside the area. To actually “do” something with the path, an option like `draw` or `fill` must be given somewhere on the path. Commands like `\draw` do this implicitly.

Finally, it is also possible to give *node specifications* on a path. Such specifications can come at different locations, but they are always allowed when a normal path operation could follow. A node specification starts with `node`. Basically, the effect is to typeset the node’s text as normal TeX text and to place it at the “current location” on the path. The details are explained in Section 17.

Note, however, that the nodes are *not* part of the path in any way. Rather, after everything has been done with the path what is specified by the path options (like filling and drawing the path due to a `fill` and a `draw` option somewhere in the *<specification>*), the nodes are added in a post-processing step.

`/tikz/name=<path name>` (no default)

Assigns a name to the path for reference (specifically, for reference in animations; for reference in intersections, use the `name path` command, which has a different purpose, see the `intersections` library for details). Since the name is a “high-level” name (drivers never know of it), you can use spaces, number, letters, or whatever you like when naming a path, but the name may *not* contain any punctuation like a dot, a comma, or a colon.

The following style influences scopes:

`/tikz/every path` (style, initially empty)

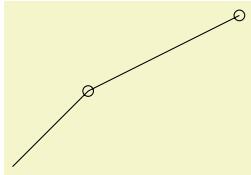
This style is installed at the beginning of every path. This can be useful for (temporarily) adding, say, the `draw` option to everything in a scope.



```
\begin{tikzpicture}
  [fill=yellow!80!black,           % only sets the color
   every path/.style={draw}]    % all paths are drawn
  \fill (0,0) rectangle +(1,1);
  \shade (2,0) rectangle +(1,1);
\end{tikzpicture}
```

`/tikz/insert path=<path>` (no default)

This key can be used inside an option to add something to the current path. This is mostly useful for defining styles that create graphic contents. This option should be used with care, for instance it should not be used as an argument of, say, a `node`. In the following example, we use a style to add little circles to a path.



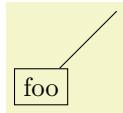
```
\tikz [c/.style={insert path={circle[radius=2pt]}}]
  \draw (0,0) -- (1,1) [c] -- (3,2) [c];
```

The effect is the same as of `(0,0) --(1,1) circle[radius=2pt] --(3,2) circle[radius=2pt]`.

The following options are for experts only:

`/tikz/append after command=<path>` (no default)

Some of the path commands described in the following sections take optional arguments. For these commands, when you use this key inside these options, the `<path>` will be inserted *after* the path command is done. For instance, when you give this command in the option list of a node, the `<path>` will be added after the node. This is used by, for instance, the `label` option to allow you to specify a label in the option list of a node, but have this `label` cause a node to be added after another node.



```
\tikz \draw node [append after command={(foo)--(1,1)},draw] (foo){foo};
```

If this key is called multiple times, the effects accumulate, that is, all of the paths are added in the order to keys were found.

`/tikz/prefix after command=<path>` (no default)

Works like `append after command`, only the accumulation order is inverse: The `<path>` is added before any earlier paths added using either `append after command` or `prefix after command`.

14.1 The Move-To Operation

The perhaps simplest operation is the move-to operation, which is specified by just giving a coordinate where a path operation is expected.

```
\path ... <coordinate> ...;
```

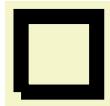
The move-to operation normally starts a path at a certain point. This does not cause a line segment to be created, but it specifies the starting point of the next segment. If a path is already under construction, that is, if several segments have already been created, a move-to operation will start a new part of the path that is not connected to any of the previous segments.



```
\begin{tikzpicture}
  \draw (0,0) --(2,0) (0,1) --(2,1);
\end{tikzpicture}
```

In the specification `(0,0) --(2,0) (0,1) --(2,1)` two move-to operations are specified: `(0,0)` and `(0,1)`. The other two operations, namely `--(2,0)` and `--(2,1)` are line-to operations, described next.

There is special coordinate called `current subpath start` that is always at the position of the last move-to operation on the current path.



```
\tikz [line width=2mm]
\draw (0,0) -- (1,0) -- (1,1)
  -- (0,1) -- (current subpath start);
```

Note how in the above example the path is not closed (as `--cycle` would do). Rather, the line just starts and ends at the origin without being a closed path.

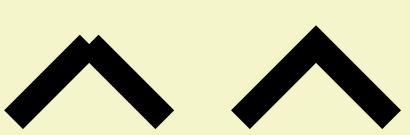
14.2 The Line-To Operation

14.2.1 Straight Lines

```
\path ... --<coordinate or cycle> ...;
```

The line-to operation extends the current path from the current point in a straight line to the given `<coordinate>` (the “or cycle” part is explained in a moment). The “current point” is the endpoint of the previous drawing operation or the point specified by a prior move-to operation.

When a line-to operation is used and some path segment has just been constructed, for example by another line-to operation, the two line segments become joined. This means that if they are drawn, the point where they meet is “joined” smoothly. To appreciate the difference, consider the following two examples: In the left example, the path consists of two path segments that are not joined, but they happen to share a point, while in the right example a smooth join is shown.



```
\begin{tikzpicture}[line width=10pt]
\draw (0,0) --(1,1) (1,1) --(2,0);
\draw (3,0) -- (4,1) -- (5,0);
\useasboundingbox (0,1.5); % make bounding box higher
\end{tikzpicture}
```

Instead of a coordinate following the two minus signs, you can also use the text `cycle`. This causes the straight line from the current point to go to the last point specified by a move-to operation. Note that this need not be the beginning of the path. Furthermore, a smooth join is created between the first segment created after the last move-to operation and the straight line appended by the cycle operation.

Consider the following example. In the left example, two triangles are created using three straight lines, but they are not joined at the ends. In the second example cycle operations are used.



```
\begin{tikzpicture}[line width=10pt]
\draw (0,0) -- (1,1) -- (1,0) -- (0,0) (2,0) -- (3,1) -- (3,0) -- (2,0);
\draw (5,0) -- (6,1) -- (6,0) -- cycle (7,0) -- (8,1) -- (8,0) -- cycle;
\useasboundingbox (0,1.5); % make bounding box higher
\end{tikzpicture}
```

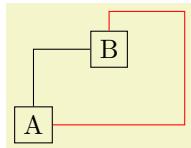
Writing `cycle` instead of a coordinate at the end of a path operation is possible with all path operations that end with a coordinate (such as `--` or `..` or `sin` or `grid`, but not `graph` or `plot`). In all cases, the effect is that the coordinate of the last `moveto` is used as the coordinate expected by the path operation and that a smooth join is added. (What actually happens is that the text `cycle` used with any path operation other than `--` gets replaced by `(current subpath start)--cycle`.)

14.2.2 Horizontal and Vertical Lines

Sometimes you want to connect two points via straight lines that are only horizontal and vertical. For this, you can use two path construction operations.

```
\path ... -|<coordinate or cycle> ...;
```

This operation means “first horizontal, then vertical”.



```
\begin{tikzpicture}
\draw (0,0) node(a) [draw] {A} (1,1) node(b) [draw] {B};
\draw (a.north) |- (b.west);
\draw[color=red] (a.east) -| (2,1.5) -| (b.north);
\end{tikzpicture}
```

Instead of a coordinate you can also write `cycle` to close the path:



```
\begin{tikzpicture}[ultra thick]
\draw (0,0) -- (1,1) -| cycle;
\end{tikzpicture}
```

```
\path ... |-<coordinate or cycle> ...;
```

This operation means “first vertical, then horizontal”.

14.3 The Curve-To Operation

The curve-to operation allows you to extend a path using a Bézier curve.

```
\path ... .. controls<c>and<d>..<y or cycle> ...;
```

This operation extends the current path from the current point, let us call it x , via a curve to a point y (if, instead of a coordinate you say `cycle` at the end, y will be the coordinate of the last move-to operation). The curve is a cubic Bézier curve. For such a curve, apart from y , you also specify two control points c and d . The idea is that the curve starts at x , “heading” in the direction of c . Mathematically spoken, the tangent of the curve at x goes through c . Similarly, the curve ends at y , “coming from” the other control point, d . The larger the distance between x and c and between d and y , the larger the curve will be.

If the “`and<d>`” part is not given, d is assumed to be equal to c .



```
\begin{tikzpicture}
  \draw[line width=10pt] (0,0) .. controls (1,1) .. (4,0)
    .. controls (5,0) and (5,1) .. (4,1);
  \draw[color=gray] (0,0) -- (1,1) -- (4,0) -- (5,0) -- (5,1) -- (4,1);
\end{tikzpicture}
```



```
\begin{tikzpicture}
  \draw[line width=10pt] (0,0) -- (2,0) .. controls (1,1) .. cycle;
\end{tikzpicture}
```

As with the line-to operation, it makes a difference whether two curves are joined because they resulted from consecutive curve-to or line-to operations, or whether they just happen to have a common (end) point:



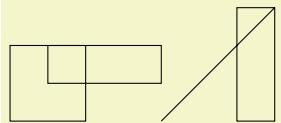
```
\begin{tikzpicture}[line width=10pt]
  \draw (0,0) -- (1,1) (1,1) .. controls (1,0) and (2,0) .. (2,0);
  \draw [yshift=-1.5cm]
    (0,0) -- (1,1) .. controls (1,0) and (2,0) .. (2,0);
\end{tikzpicture}
```

14.4 The Rectangle Operation

A rectangle can obviously be created using four straight lines and a cycle operation. However, since rectangles are needed so often, a special syntax is available for them.

```
\path ... rectangle<corner or cycle> ...;
```

When this operation is used, one corner will be the current point, another corner is given by *<corner>*, which becomes the new current point.



```
\begin{tikzpicture}
  \draw (0,0) rectangle (1,1);
  \draw (.5,1) rectangle (2,0.5) (3,0) rectangle (3.5,1.5) -- (2,0);
\end{tikzpicture}
```

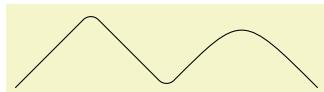
Just for consistency, you can also use *cycle* instead of a coordinate, but it is a bit unclear what use this might have.

14.5 Rounding Corners

All of the path construction operations mentioned up to now are influenced by the following option:

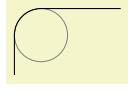
*/tikz/rounded corners=(*inset*)* (default 4pt)

When this option is in force, all corners (places where a line is continued either via line-to or a curve-to operation) are replaced by little arcs so that the corner becomes smooth.



```
\tikz \draw [rounded corners] (0,0) -- (1,1)
  -- (2,0) .. controls (3,1) .. (4,0);
```

The *<inset>* describes how big the corner is. Note that the *<inset>* is *not* scaled along if you use a scaling option like `scale=2`.



```
\begin{tikzpicture}
  \draw [color=gray,very thin] (10pt,15pt) circle [radius=10pt];
  \draw [rounded corners=10pt] (0,0) -- (0pt,25pt) -- (40pt,25pt);
\end{tikzpicture}
```

You can switch the rounded corners on and off “in the middle of path” and different corners in the same path can have different corner radii:



```
\begin{tikzpicture}
  \draw (0,0) [rounded corners=10pt] -- (1,1) -- (2,1)
        [sharp corners] -- (2,0)
        [rounded corners=5pt] -- cycle;
\end{tikzpicture}
```

Here is a rectangle with rounded corners:



```
\tikz \draw[rounded corners=1ex] (0,0) rectangle (20pt,2ex);
```

You should be aware, that there are several pitfalls when using this option. First, the rounded corner will only be an arc (part of a circle) if the angle is 90° . In other cases, the rounded corner will still be round, but “not as nice”.

Second, if there are very short line segments in a path, the “rounding” may cause inadvertent effects. In such case it may be necessary to temporarily switch off the rounding using `sharp corners`.

`/tikz/sharp corners`

(no value)

This options switches off any rounding on subsequent corners of the path.

14.6 The Circle and Ellipse Operations

Circles and ellipses are common path elements for which there is a special path operation.

```
\path ... circle[\langle options\rangle] ...;
```

This command adds a circle to the current path where the center of the circle is the current point by default, but you can use the `at` option to change this. The new current point of the path will be (typically just remain) the center of the circle.

The radius of the circle is specified using the following options:

`/tikz/x radius=<value>`

(no default)

Sets the horizontal radius of the circle (which, when this value is different form the vertical radius, is actually an ellipse). The `<value>` may either be a dimension or a dimensionless number. In the latter case, the number is interpreted in the *xy*-coordinate system (if the *x*-unit is set to, say, `2cm`, then `x radius=3` will have the same effect as `x radius=6cm`).

`/tikz/y radius=<value>`

(no default)

Works like the `x radius`.

`/tikz/radius=<value>`

(no default)

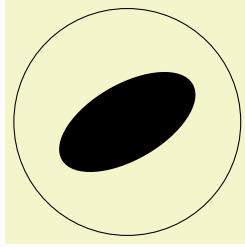
Sets the `x radius` and `y radius` simultaneously.

`/tikz/at=<coordinate>`

(no default)

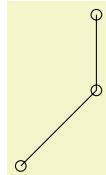
If this option is explicitly set inside the `<options>` (or indirectly via the `every circle` style), the `<coordinate>` is used as the center of the circle instead of the current point. Setting `at` to some value in an enclosing scope has no effect.

The `<options>` may also contain additional options like, say, a `rotate` or `scale`, that will only have an effect on the circle.



```
\begin{tikzpicture}
  \draw (1,0) circle [radius=1.5];
  \fill (1,0) circle [x radius=1cm, y radius=5mm, rotate=30];
\end{tikzpicture}
```

It is possible to set the `radius` also in some enclosing scope, in this case the options can be left out (but see the note below on what may follow):



```
\begin{tikzpicture}[radius=2pt]
  \draw (0,0) circle -- (1,1) circle -- ++(0,1) circle;
\end{tikzpicture}
```

The following style is used with every circle:

`/tikz/every circle` (style, no value)

You can use this key to set up, say, a default radius for every circle. The key will also be used with the `ellipse` operation.

In case you feel that the names `radius` and `x radius` are too long for your taste, you can easily created shorter aliases:

```
\tikzset{r/.style={radius=#1},rx/.style={x radius=#1},ry/.style={y radius=#1}}
```

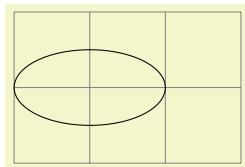
You can then say `circle [r=1cm` or `circle [rx=1,ry=1.5`. The reason TikZ uses the longer names by default is that it encourages people to write more readable code.

Note: There also exists an older syntax for circles, where the radius of the circle is given in parentheses right after the `circle` command as in `circle (1pt)`. Although this syntax is a bit more succinct, it is harder to understand for readers of the code and the use of parentheses for something other than a coordinate is ill-chosen.

TikZ will use the following rule to determine whether the old or the normal syntax is used: If `circle` is directly followed by something that (expands to) an opening parenthesis, then the old syntax is used and inside these following parentheses there must be a single number or dimension representing a radius. In all other cases the new syntax is used.

```
\path ... ellipse[<options>] ...;
```

This command has exactly the same effect as `circle`. The older syntax for this command is `ellipse (<x radius> and <y radius>)`. As for the `circle` command, this syntax is not as good as the standard syntax.



```
\begin{tikzpicture}
  \draw [help lines] (0,0) grid (3,2);
  \draw (1,1) ellipse [x radius=1cm,y radius=.5cm];
\end{tikzpicture}
```

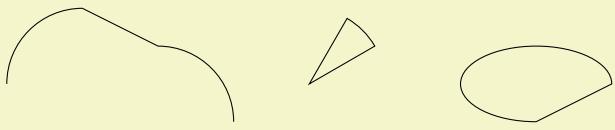
14.7 The Arc Operation

The *arc operation* allows you to add an arc to the current path.

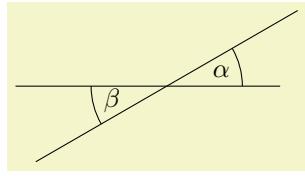
```
\path ... arc [<options>] ...;
```

The `arc` operation adds a part of an ellipse to the current path. The radii of the ellipse are given by the values of `x radius` and `y radius`, which should be set in the `\langle options \rangle`. The arc will start at the current point and will end at the end of the arc. The arc will start and end at angles computed from the three keys `start angle`, `end angle`, and `delta angle`. Normally, the first two keys specify the start and end angle. However, in case one of them is empty, it is computed from the other key plus or minus the `delta angle`. In detail, if `end angle` is empty, it is set to the start angle plus the delta angle. If the start angle is missing, it is set to the end angle minus the delta angle. If all three keys are set, the delta angle is ignored.

<code>/tikz/start angle=<degrees></code>	(no default)
Sets the start angle.	
<code>/tikz/end angle=<degrees></code>	(no default)
Sets the end angle.	
<code>/tikz/delta angle=<degrees></code>	(no default)
Sets the delta angle.	



```
\begin{tikzpicture}[radius=1cm]
\draw (0,0) arc [start angle=180, end angle=90]
      -- (2,.5) arc [start angle=90, delta angle=-90];
\draw (4,0) -- +(30:1cm)
      arc [start angle=30, delta angle=30] -- cycle;
\draw (8,0) arc [start angle=0, end angle=270,
                 x radius=1cm, y radius=5mm] -- cycle;
\end{tikzpicture}
```



```
\begin{tikzpicture}[radius=1cm,delta angle=30]
\draw (-1,0) -- +(3.5,0);
\draw (1,0) ++(210:2cm) -- +(30:4cm);
\draw (1,0)+(0:1cm) arc [start angle=0];
\draw (1,0)+(180:1cm) arc [start angle=180];
\path (1,0) ++(15:.75cm) node{$\alpha$};
\path (1,0) ++(15:-.75cm) node{$\beta$};
\end{tikzpicture}
```

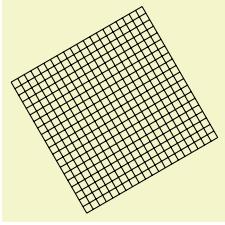
There also exists a shorter syntax for the arc operation, namely `arc` begin directly followed by `(<start angle>:<end angle>:<radius>)`. However, this syntax is harder to read, so the normal syntax should be preferred in general.

14.8 The Grid Operation

You can add a grid to the current path using the `grid` path operation.

```
\path ... grid[\langle options \rangle] \langle corner or cycle \rangle ...;
```

This operation adds a grid filling a rectangle whose two corners are given by `\langle corner \rangle` and by the previous coordinate. (Instead of a coordinate you can also say `cycle` to use the position of the last move-to as the corner coordinate, but it is not very natural to do so.) `corner` Thus, the typical way in which a grid is drawn is `\draw (1,1) grid (3,3);`, which yields a grid filling the rectangle whose corners are at $(1,1)$ and $(3,3)$. All coordinate transformations apply to the grid.

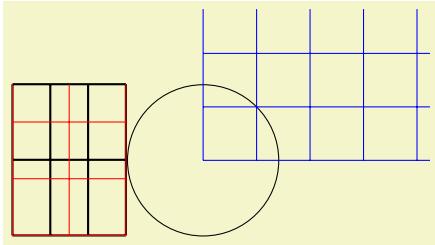


```
\tikz [rotate=30] \draw [step=1mm] (0,0) grid (2,2);
```

The `<options>`, which are local to the `grid` operation, can be used to influence the appearance of the grid. The stepping of the grid is governed by the following options:

/tikz/step=(number or dimension or coordinate) (no default, initially `1cm`)

Sets the stepping in both the *x* and *y*-direction. If a dimension is provided, this is used directly. If a number is provided, this number is interpreted in the *xy*-coordinate system. For example, if you provide the number 2, then the *x*-step is twice the *x*-vector and the *y*-step is twice the *y*-vector set by the `x=` and `y=` options. Finally, if you provide a coordinate, then the *x*-part of this coordinate will be used as the *x*-step and the *y*-part will be used as the *y*-coordinate.



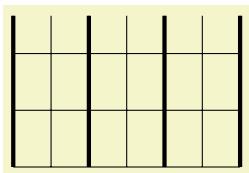
```
\begin{tikzpicture} [x=.5cm]
\draw[thick] (0,0) grid [step=1] (3,2);
\draw[red] (0,0) grid [step=.75cm] (3,2);
\end{tikzpicture}
\begin{tikzpicture}
\draw (0,0) circle [radius=1];
\draw[blue] (0,0) grid [step=(45:1)] (3,2);
\end{tikzpicture}
```

A complication arises when the *x*- and/or *y*-vector do not point along the axes. Because of this, the actual rule for computing the *x*-step and the *y*-step is the following: As the *x*- and *y*-steps we use the *x*- and *y*-components of the following two vectors: The first vector is either $(\langle x\text{-grid-step-number}\rangle, 0)$ or $(\langle x\text{-grid-step-dimension}\rangle, 0\text{pt})$, the second vector is $(0, \langle y\text{-grid-step-number}\rangle)$ or $(0\text{pt}, \langle y\text{-grid-step-dimension}\rangle)$.

If the *x*-step or *y*-step is 0 or negative the corresponding lines are not drawn.

/tikz/xstep=(dimension or number) (no default, initially `1cm`)

Sets the stepping in the *x*-direction.



```
\begin{tikzpicture}
\draw (0,0) grid [xstep=.5,ystep=.75] (3,2);
\draw[ultra thick] (0,0) grid [ystep=0] (3,2);
\end{tikzpicture}
```

/tikz/ystep=(dimension or number) (no default, initially `1cm`)

Sets the stepping in the *y*-direction.

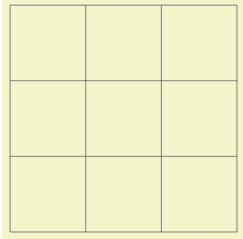
It is important to note that the grid is always “phased” such that it contains the point $(0, 0)$ if that point happens to be inside the rectangle. Thus, the grid does *not* always have an intersection at the corner points; this occurs only if the corner points are multiples of the stepping. Note that due to rounding

errors, the “last” lines of a grid may be omitted. In this case, you have to add an epsilon to the corner points.

The following style is useful for drawing grids:

`/tikz/help lines` (style, initially `line width=0.2pt,gray`)

This style makes lines “subdued” by using thin gray lines for them. However, this style is not installed automatically and you have to say for example:



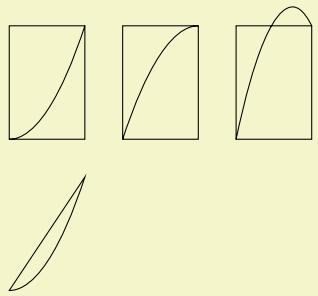
```
\tikz \draw[help lines] (0,0) grid (3,3);
```

14.9 The Parabola Operation

The `parabola` path operation continues the current path with a parabola. A parabola is a (shifted and scaled) curve defined by the equation $f(x) = x^2$ and looks like this: \cup .

```
\path ... parabola[<options>] bend <bend coordinate> <coordinate or cycle> ...;
```

This operation adds a parabola through the current point and the given `<coordinate>` or, if `cycle` is used instead of coordinate at the end, the `<coordinate>` is set to the position of the last move-to and the path gets closed after the parabola. If the `bend` is given, it specifies where the bend should go; the `<options>` can also be used to specify where the bend is. By default, the bend is at the old current point.



```
\begin{tikzpicture}
  \draw      (0,0) rectangle (1,1.5);
  \draw[parabola] (0,0) parabola (1,1.5);
  \draw[xshift=1.5cm] (0,0) rectangle (1,1.5);
  \draw[xshift=1.5cm] (0,0) parabola[bend at end] (1,1.5);
  \draw[xshift=3cm] (0,0) rectangle (1,1.5);
  \draw[xshift=3cm] (0,0) parabola bend (.75,1.75) (1,1.5);

  \draw[yshift=-2cm] (1,1.5) -- (0,0) parabola cycle;
\end{tikzpicture}
```

The following options influence parabolas:

`/tikz/bend=<coordinate>` (no default)

Has the same effect as saying `bend<coordinate>` outside the `<options>`. The option specifies that the bend of the parabola should be at the given `<coordinate>`. You have to take care yourself that the bend position is a “valid” position; which means that if there is no parabola of the form $f(x) = ax^2 + bx + c$ that goes through the old current point, the given bend, and the new current point, the result will not be a parabola.

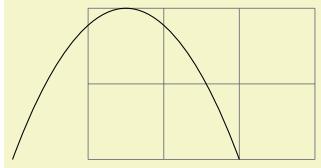
There is one special property of the `<coordinate>`: When a relative coordinate is given like `+(0,0)`, the position relative to this coordinate is “flexible”. More precisely, this position lies somewhere on a line from the old current point to the new current point. The exact position depends on the next option.

`/tikz/bend pos=<fraction>` (no default)

Specifies where the “previous” point is relative to which the bend is calculated. The previous point will be at the `<fraction>`th part of the line from the old current point to the new current point.

The idea is the following: If you say `bend pos=0` and `bend +(0,0)`, the bend will be at the old current point. If you say `bend pos=1` and `bend +(0,0)`, the bend will be at the new current point.

If you say `bend pos=0.5` and `bend +(0,2cm)` the bend will be 2cm above the middle of the line between the start and end point. This is most useful in situations such as the following:

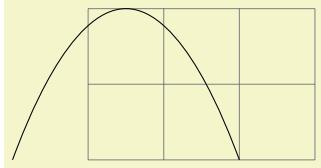


```
\begin{tikzpicture}
  \draw[help lines] (0,0) grid (3,2);
  \draw (-1,0) parabola[bend pos=0.5] bend +(0,2) +(3,0);
\end{tikzpicture}
```

In the above example, the `bend +(0,2)` essentially means “a parabola that is 2cm high” and `+(3,0)` means “and 3cm wide”. Since this situation arises often, there is a special shortcut option:

`/tikz/parabola height=<dimension>` (no default)

This option has the same effect as `[bend pos=0.5,bend={+(0pt,<dimension>)}]`.



```
\begin{tikzpicture}
  \draw[help lines] (0,0) grid (3,2);
  \draw (-1,0) parabola[parabola height=2cm] +(3,0);
\end{tikzpicture}
```

The following styles are useful shortcuts:

`/tikz/bend at start` (style, no value)

This places the bend at the start of a parabola. It is a shortcut for the following options: `bend pos=0,bend={+(0,0)}`.

`/tikz/bend at end` (style, no value)

This places the bend at the end of a parabola.

14.10 The Sine and Cosine Operation

The `sin` and `cos` operations are similar to the `parabola` operation. They, too, can be used to draw (parts of) a sine or cosine curve.

```
\path ... sin<coordinate or cycle> ...;
```

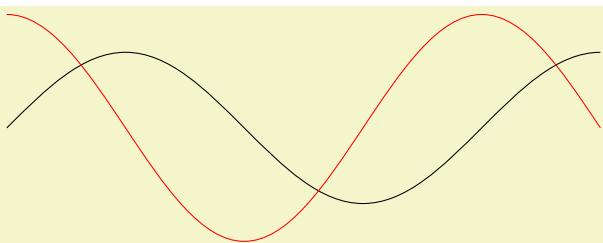
The effect of `sin` is to draw a scaled and shifted version of a sine curve in the interval $[0, \pi/2]$. The scaling and shifting is done in such a way that the start of the sine curve in the interval is at the old current point and that the end of the curve in the interval is at `<coordinate>`. Here is an example that should clarify this:



```
\tikz \draw (0,0) rectangle (1,1) (0,0) sin (1,1)
          (2,0) rectangle +(1.57,1) (2,0) sin +(1.57,1);
```

```
\path ... cos<coordinate or cycle> ...;
```

This operation works similarly, only a cosine in the interval $[0, \pi/2]$ is drawn. By correctly alternating `sin` and `cos` operations, you can create a complete sine or cosine curve:



```
\begin{tikzpicture}[xscale=1.57]
  \draw (0,0) sin (1,1) cos (2,0) sin (3,-1) cos (4,0) sin (5,1);
  \draw[color=red] (0,1.5) cos (1,0) sin (2,-1.5) cos (3,0) sin (4,1.5) cos (5,0);
\end{tikzpicture}
```

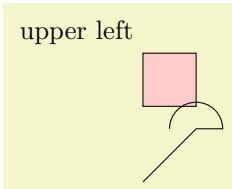
Note that there is no way to (conveniently) draw an interval on a sine or cosine curve whose end points are not multiples of $\pi/2$.

14.11 The SVG Operation

The `svg` operation can be used to extend the current path by a path given in the SVG path data syntax. This syntax is described in detail in Section 8.3 of the SVG 1.1 specification, please consult this specification for details.

```
\path ... svg[\langle options\rangle]{\langle path data\rangle} ...;
```

This operation adds the path specified in the `\langle path data\rangle` in SVG 1.1 PATH DATA syntax to the current path. Unlike the SVG-specification, it *is* permissible that the path data does not start with a move-to command (`m` or `M`), in which case the last point of the current path is used as start point. The optional `\langle options\rangle` apply locally to this path operation, typically you will use them to set up, say, some transformations.



```
\usetikzlibrary {svg.path}
\begin{tikzpicture}
  \filldraw [fill=red!20] (0,1) svg[scale=2] {h 10 v 10 h -10}
    node [above left] {upper left} -- cycle;

  \draw svg {M 0 0 L 20 20 h 10 a 10 10 0 0 -20 0};
\end{tikzpicture}
```

An SVG coordinate like `10 20` is always interpreted as `(10pt,20pt)`, so the basic unit is always points (`pt`). The *xy*-coordinate system is not used. However, you can use scaling to (locally) change the basic unit. For instance, `svg[scale=1cm]` (yes, this works, although some rather evil magic is involved) will cause `1cm` to be the basic unit.

Instead of curly braces, you can also use quotation marks to indicate the start and end of the SVG path.

Warning: The arc operations (`a` and `A`) are numerically unstable. This means that they will be quite imprecise, except when the angle is a multiple of 90° (as is, fortunately, most often the case).

14.12 The Plot Operation

The `plot` operation can be used to append a line or curve to the path that goes through a large number of coordinates. These coordinates are either given in a simple list of coordinates, read from some file, or they are computed on the fly.

Since the syntax and the behaviour of this command are a bit complex, they are described in the separated Section 22.

14.13 The To Path Operation

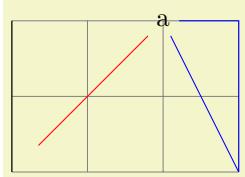
The `to` operation is used to add a user-defined path from the previous coordinate to the following coordinate. When you write `(a) to (b)`, a straight line is added from `a` to `b`, exactly as if you had written `(a) --(b)`. However, if you write `(a) to [out=135,in=45] (b)` a curve is added to the path, which leaves at an angle of 135° at `a` and arrives at an angle of 45° at `b`. This is because the options `in` and `out` trigger a special path to be used instead of the straight line.

```
\path ... to[\langle options\rangle] \langle nodes\rangle \langle coordinate or cycle\rangle ...;
```

This path operation inserts the path currently set via the `to path` option at the current position. The `\langle options\rangle` can be used to modify (perhaps implicitly) the `to path` and to set up how the path will be rendered.

Before the `to path` is inserted, a number of macros are set up that can “help” the `to path`. These are `\tikztostart`, `\tikztotarget`, and `\tikztonodes`; they are explained in the following.

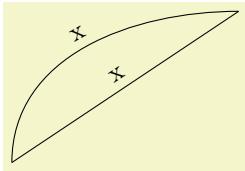
Start and Target Coordinates. The `to` operation is always followed by a `(coordinate)`, called the target coordinate, or the text cycle, in which case the last move-to is used as a coordinate and the path gets closed. The macro `\tikztotarget` is set to this coordinate (without its parentheses). There is also a *start coordinate*, which is the coordinate preceding the `to` operation. This coordinate can be accessed via the macro `\tikztostart`. In the following example, for the first `to`, the macro `\tikztostart` is `0pt,0pt` and the `\tikztotarget` is `0,2`. For the second `to`, the macro `\tikztostart` is `10pt,10pt` and `\tikztotarget` is `a`. For the third, they are set to `a` and current subpath start.



```
\begin{tikzpicture}
\draw[help lines] (0,0) grid (3,2);
\node (a) at (2,2) {a};

\draw (0,0) to (0,2);
\draw[red] (10pt,10pt) to (a);
\draw[blue] (3,0) -- (3,2) -- (a) to cycle;
\end{tikzpicture}
```

Nodes on to-paths. It is possible to add nodes to the paths constructed by a `to` operation. To do so, you specify the nodes between the `to` keyword and the coordinate (if there are options to the `to` operation, these come first). The effect of `(a) to node {x} (b)` (typically) is the same as if you had written `(a) --node {x} (b)`, namely that the node is placed on the `to`. This can be used to add labels to `tos`:



```
\begin{tikzpicture}
\draw (0,0) to node [sloped,above] {x} (3,2);

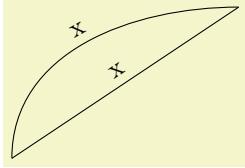
\draw (0,0) to[out=90,in=180] node [sloped,above] {x} (3,2);
\end{tikzpicture}
```

Instead of writing the node between the `to` keyword and the target coordinate, you may also use the following keys to create such nodes:

`/tikz/edge node=<node specification>`

(no default)

This key can be used inside the `<options>` of a `to` path command. It will add the `<node specification>` to the list of nodes to be placed on the connecting line, just as if you had written the `<node specification>` directly after the `to` keyword:



```
\begin{tikzpicture}
\draw (0,0) to [edge node={node [sloped,above] {x}}] (3,2);

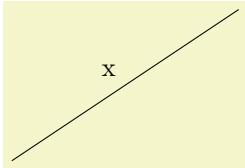
\draw (0,0) to [out=90,in=180,
edge node={node [sloped,above] {x}}] (3,2);
\end{tikzpicture}
```

This key is mostly useful to create labels automatically using other keys.

`/tikz/edge label=<text>`

(no default)

A shorthand for `edge node={node [auto] {<text>}}`.

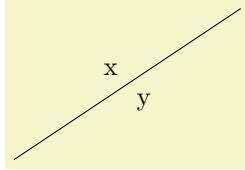


```
\tikz \draw (0,0) to [edge label=x] (3,2);
```

`/tikz/edge label'=<text>`

(no default)

A shorthand for `edge node={node [auto, swap] {<text>}}`.



```
\tikz \draw (0,0) to [edge label=x, edge label'=y] (3,2);
```

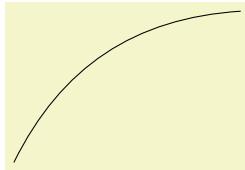
When the `quotes` library is loaded, additional ways of specifying nodes on `to`-paths become available, see Section 17.12.2.

Styles for `to`-paths. In addition to the `<options>` given after the `to` operation, the following style is also set at the beginning of the `to` path:

`/tikz/every to`

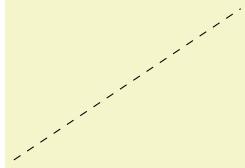
(style, initially empty)

This style is installed at the beginning of every `to`.



```
\tikz[every to/.style={bend left}]
\draw (0,0) to (3,2);
```

Note that, as explained below, every `to` path is implicitly surrounded by curly braces. This means that options like `draw` given in an `every to` do not actually influence the path. You can fix this by using the `append after command` option:



```
\tikz[every to/.style={append after command={[draw,dashed]}}]
\draw (0,0) to (3,2);
```

Options. The `<options>` given with the `to` allow you to influence the appearance of the `to path`. Mostly, these options are used to change the `to path`. This can be used to change the path from a straight line to, say, a curve.

The path used is set using the following option:

`/tikz/to path=<path>`

(no default)

Whenever a `to` operation is used, the `<path>` is inserted. More precisely, the following path is added:

`{[every to,<options>] <path>}`

The `<options>` are the options given to the `to` operation, the `<path>` is the path set by this option `to path`.

Inside the `<path>`, different macros are used to reference the from- and to-coordinates. In detail, these are:

- `\tikztostart` will expand to the from-coordinate (without the parentheses).
- `\tikztotarget` will expand to the to-coordinate.
- `\tikztonodes` will expand to the nodes between the `to` operation and the coordinate. Furthermore, these nodes will have the `pos` option set implicitly.

Let us have a look at a simple example. The standard straight line for a `to` is achieved by the following `<path>`:

`--(\tikztotarget) \tikztonodes`

Indeed, this is the default setting for the path. When we write `(a) to (b)`, the `<path>` will expand to `(a) --(b)`, when we write

```
(a) to[red] node {x} (b)
```

the $\langle path \rangle$ will expand to

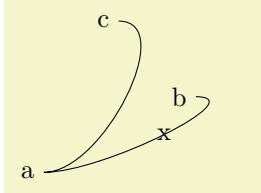
```
(a) --(b) node[red] {x}
```

It is not possible to specify the path

```
--\tikztonodes (\tikztotarget)
```

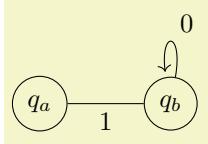
since TikZ does not allow one to have a macro after `--` that expands to a node.

Now let us have a look at how we can modify the $\langle path \rangle$ sensibly. The simplest way is to use a curve.



```
\begin{tikzpicture}[to path={%
    .. controls +(1,0) and +(1,0) .. (\tikztotarget) \tikztonodes}]
\node (a) at (0,0) {a};
\node (b) at (2,1) {b};
\node (c) at (1,2) {c};
\draw (a) to node {x} (b)
      (a) to           (c);
\end{tikzpicture}
```

Here is another example:



```
\tikzset{
  my loop/.style={to path={%
    .. controls +(80:1) and +(100:1) .. (\tikztotarget) \tikztonodes},
  my state/.style={circle,draw}}
\begin{tikzpicture}[shorten >=2pt]
\node [my state] (a) at (210:1) {$q_a$};
\node [my state] (b) at (330:1) {$q_b$};

\draw[->] (a) to [below] {1} (b)
            to [my loop] node[above right] {0} (b);
\end{tikzpicture}
```

/tikz/execute at begin to= $\langle code \rangle$ (no default)

The $\langle code \rangle$ is executed prior to the `to`. This can be used to draw one or more additional paths or to do additional computations.

/tikz/execute at end to= $\langle code \rangle$ (no default)

Works like the previous option, only this code is executed after the `to` path has been added.

/tikz/every to (style, initially empty)

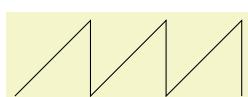
This style is installed at the beginning of every `to`.

There are a number of predefined `to` paths, see Section 75 for a reference.

14.14 The Foreach Operation

```
\path ... foreach<variables>[<options>] in {<path commands>} ...;
```

The `foreach` operation can be used to repeatedly insert the $\langle path commands \rangle$ into the current path. Naturally, the $\langle path commands \rangle$ should internally reference some of the $\langle variables \rangle$ so that you do not insert exactly the same path repeatedly, but rather variations. For historical reasons, you can also write `\foreach` instead of `foreach`.



```
\tikz \draw (0,0) foreach \x in {1,...,3} { -- (\x,1) -- (\x,0) };
```

See Section ?? for more details on the for-each-command.

14.15 The Let Operation

The *let operation* is the first of a number of path operations that do not actually extend that path, but have different, mostly local, effects.

```
\path ... let<assignment> ,<assignment>,<assignment>... in ... ;
```

When this path operation is encountered, the *<assignment>*s are evaluated, one by one. This will store coordinate and number in special *registers* (which are local to TikZ, they have nothing to do with TeX registers). Subsequently, one can access the contents of these registers using the macros `\p`, `\x`, `\y`, and `\n`.

The first kind of permissible *<assignment>*s have the following form:

```
\n<number register>={<formula>}
```

When an assignment has this form, the *<formula>* is evaluated using the `\pgfmathparse` operation. The result is stored in the *<number register>*. If the *<formula>* involves a dimension anywhere (as in $2*3\text{cm}/2$), then the *<number register>* stores the resulting dimension with a trailing `pt`. A *<number register>* can be named arbitrarily and is a normal TeX parameter to the `\n` macro. Possible names are `{left corner}`, but also just a single digit like `5`.

Let us call the path that follows a let operation its *body*. Inside the body, the `\n` macro can be used to access the register.

```
\n{<number register>}
```

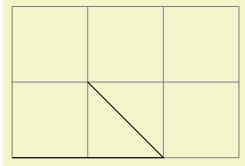
When this macro is used on the left-hand side of an `=`-sign in a let operation, it has no effect and is just there for readability. When the macro is used on the right-hand side of an `=`-sign or in the body of the let operation, then it expands to the value stored in the *<number register>*. This will either be a dimensionless number like `2.0` or a dimension like `5.6pt`.

For instance, if we say `let \n1={1pt+2pt}, \n2={1+2} in ...`, then inside the `...` part the macro `\n1` will expand to `3pt` and `\n2` expands to `3`.

The second kind of *<assignments>* have the following form:

```
\p<point register>={<formula>}
```

Point position registers store a single point, consisting of an *x*-part and a *y*-part measured in TeX points (`pt`). In particular, point registers do not store nodes or node names. Here is an example:



```
\usetikzlibrary {calc}
\begin{tikzpicture}
  \draw [help lines] (0,0) grid (3,2);

  \draw let \p{foo} = (1,1), \p2 = (2,0) in
    (0,0) -- (\p2) -- (\p{foo});
\end{tikzpicture}
```

```
\p{<point register>}
```

When this macro is used on the left-hand side of an `=`-sign in a let operation, it has no effect and is just there for readability. When the macro is used on the right-hand side of an `=`-sign or in the body of the let operation, then it expands to the *x*-part (measured in TeX points) of the coordinate stored in the *<register>*, followed, by a comma, followed by the *y*-part.

For instance, if we say `let \p1=(1pt,1pt+2pt) in ...`, then inside the `...` part the macro `\p1` will expand to exactly the seven characters “`1pt,3pt`”. This means that you when you write `(\p1)`, this expands to `(1pt,3pt)`, which is presumably exactly what you intended.

```
\x{<point register>}
```

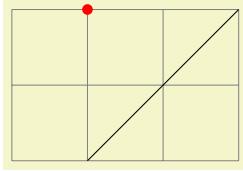
This macro expands just to the *x*-part of the point register. If we say as above, as we did above, `let \p1=(1pt,1pt+2pt) in ...`, then inside the `...` part the macro `\x1` expands to `1pt`.

```
\y{<point register>}
```

Works like `\x`, only for the *y*-part.

Note that the above macros are available only inside a let operation.

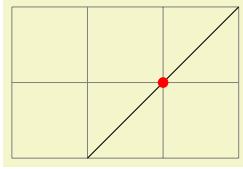
Here is an example where let clauses are used to assemble a coordinate from the x -coordinate of a first point and the y -coordinate of a second point. Naturally, using the $\mid-$ notation, this could be written much more compactly.



```
\usetikzlibrary {calc}
\begin{tikzpicture}
  \draw [help lines] (0,0) grid (3,2);
  \draw (1,0) coordinate (first point)
    -- (3,2) coordinate (second point);

  \fill[red] let \p1 = (first point),
             \p2 = (second point) in
             (\x1,\y2) circle [radius=2pt];
\end{tikzpicture}
```

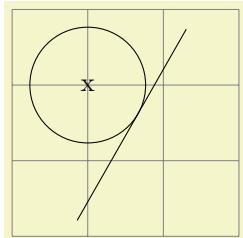
Note that the effect of a let operation is local to the body of the let operation. If you wish to access a computed coordinate outside the body, you must use a `coordinate` path operation:



```
\usetikzlibrary {calc}
\begin{tikzpicture}
  \draw [help lines] (0,0) grid (3,2);
  \path % let's define some points:
    let
      \p1      = (1,0),
      \p2      = (3,2),
      \p{center} = ($ (\p1) !.5! (\p2) $) % center
    in
      coordinate (p1) at (\p1)
      coordinate (p2) at (\p2)
      coordinate (center) at (\p{center});

  \draw (p1) -- (p2);
  \fill[red] (center) circle [radius=2pt];
\end{tikzpicture}
```

For a more useful application of the let operation, let us draw a circle that touches a given line:



```
\usetikzlibrary {calc}
\begin{tikzpicture}
  \draw [help lines] (0,0) grid (3,3);
  \coordinate (a) at (rnd,rnd);
  \coordinate (b) at (3-rnd,3-rnd);
  \draw (a) -- (b);

  \node (c) at (1,2) {x};

  \draw let \p1 = ($ (a)!(c)!(b) - (c) $),
        \n1 = {veclen(\x1,\y1)}
        in circle [at=(c), radius=\n1];
\end{tikzpicture}
```

14.16 The Scoping Operation

When Ti k Z encounters an opening or a closing brace ($\{$ or $\}$) at some point where a path operation should come, it will open or close a scope. All options that can be applied “locally” will be scoped inside the scope. For example, if you apply a transformation like `[xshift=1cm` inside the scoped area, the shifting only applies to the scope. On the other hand, an option like `color=red` does not have any effect inside a scope since it can only be applied to the path as a whole.

Concerning the effect of scopes on relative coordinates, please see Section 13.4.3.

14.17 The Node and Edge Operations

The `node` operation adds a so-called node to a path. This operation is special in the following sense: It does not change the current path in any way. In other words, this operation is not really a path operation,

but has an effect that is “external” to the path. The `edge` operation has similar effect in that it adds something *after* the main path has been drawn. However, it works like the `to` operation, that is, it adds a `to` path to the picture after the main path has been drawn.

Since these operations are quite complex, they are described in the separate Section 17.

14.18 The Graph Operation

The `graph` operation can be used to specify easily how a large number of nodes are connected. This operation is documented in a separate section, see Section 19.

14.19 The Pic Operation

The `pic` operation is used to insert a “short picture” (hence the “short” name) at the current position of the path. This operation is somewhat similar to the `node` operation and discussed in detail in Section 18.

14.20 The Attribute Animation Operation

```
\path ... :<animation attribute>={<options>} ...;
```

This path operation has the same effect as if you had said:

```
[animate = { myself:<animate attribute>={<options>} } ]
```

This causes an animation of `<animate attribute>` to be added to the current path, see Section 26 for details.



```
\usetikzlibrary {animations}
\begin{tikzpicture}
\draw [xshift = {0s = "0cm", 30s = "-3cm", repeats} (0,0) circle (5mm);
```

14.21 The PGF-Extra Operation

In some cases you may need to “do some calculations or some other stuff” while a path is constructed. For this, you would like to suspend the construction of the path and suspend TikZ’s parsing of the path, you would then like to have some TeX code executed, and would then like to resume the parsing of the path. This effect can be achieved using the following path operation `\pgfextra`. Note that this operation should only be used by real experts and should only be used deep inside clever macros, not on normal paths.

```
\pgfextra{<code>}
```

This command may only be used inside a TikZ path. There it is used like a normal path operation. The construction of the path is temporarily suspended and the `<code>` is executed. Then, the path construction is resumed.

```
\newdimen\mydim
\begin{tikzpicture}
\mydim=1cm
\draw (0pt,\mydim) \pgfextra{\mydim=2cm} -- (0pt,\mydim);
\end{tikzpicture}
```

```
\pgfextra<code> \endpgfextra
```

This is an alternative syntax for the `\pgfextra` command. If the code following `\pgfextra` does not start with a brace, the `<code>` is executed until `\endpgfextra` is encountered. What actually happens is that when `\pgfextra` is not followed by a brace, this completely shuts down the TikZ parser and `\endpgfextra` is a normal macro that restarts the parser.

```
\newdimen\mydim
\begin{tikzpicture}
\mydim=1cm
\draw (0pt,\mydim)
\pgfextra \mydim=2cm \endpgfextra -- (0pt,\mydim);
\end{tikzpicture}
```

14.22 Interacting with the Soft Path subsystem

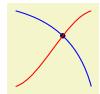
During construction TikZ stores the path internally as a *soft path*. Sometimes it is desirable to save a path during the stage of construction, restore it elsewhere and continue using it. There are two keys to facilitate this operation, which are explained below. To learn more about the soft path subsystem, refer to section ??.

/tikz/save path=*<macro>* (no default)

Save the current soft path into *<macro>*.

/tikz/use path=*<macro>* (no default)

Set the current path to the soft path stored in *<macro>*.



```
\usetikzlibrary {intersections}
\begin{tikzpicture}
  \path [save path=\pathA, name path=A] (0,1) to [bend left] (1,0);
  \path [save path=\pathB, name path=B]
    (0,0) .. controls (.33,.1) and (.66,.9) .. (1,1);

  \fill [name intersections={of=A and B}] (intersection-1) circle (1pt);

  \draw [blue] [use path=\pathA];
  \draw [red] [use path=\pathB];
\end{tikzpicture}
```

15 Actions on Paths

15.1 Overview

Once a path has been constructed, different things can be done with it. It can be drawn (or stroked) with a “pen”, it can be filled with a color or shading, it can be used for clipping subsequent drawing, it can be used to specify the extend of the picture – or any combination of these actions at the same time.

To decide what is to be done with a path, two methods can be used. First, you can use a special-purpose command like `\draw` to indicate that the path should be drawn. However, commands like `\draw` and `\fill` are just abbreviations for special cases of the more general method: Here, the `\path` command is used to specify the path. Then, options encountered on the path indicate what should be done with the path.

For example, `\path [draw] (0,0) circle (1cm);` means: “This is a path consisting of a circle around the origin. Do not do anything with it (throw it away).” However, if the option `draw` is encountered anywhere on the path, the circle will be drawn. “Anywhere” is any point on the path where an option can be given, which is everywhere where a path command like `circle (1cm)` or `rectangle (1,1)` or even just `(0,0)` would also be allowed. Thus, the following commands all draw the same circle:

```
\path [draw] (0,0) circle (1cm);
\path (0,0) [draw] circle (1cm);
\path (0,0) circle (1cm) [draw];
```

Finally, `\draw (0,0) circle (1cm);` also draws a path, because `\draw` is an abbreviation for `\path[draw]` and thus the command expands to the first line of the above example.

Similarly, `\fill` is an abbreviation for `\path[fill]` and `\filldraw` is an abbreviation for the command `\path[fill,draw]`. Since options accumulate, the following commands all have the same effect:

```
\path [draw,fill] (0,0) circle (1cm);
\path [draw] [fill] (0,0) circle (1cm);
\path [fill] (0,0) circle (1cm) [draw];
\draw [fill] (0,0) circle (1cm);
\fill (0,0) [draw] circle (1cm);
\filldraw (0,0) circle (1cm);
```

In the following subsection the different actions that can be performed on a path are explained. The following commands are abbreviations for certain sets of actions, but for many useful combinations there are no abbreviations:

\draw

Inside `{tikzpicture}` this is an abbreviation for `\path[draw]`.

\fill

Inside `{tikzpicture}` this is an abbreviation for `\path[fill]`.

\filldraw

Inside `{tikzpicture}` this is an abbreviation for `\path[fill,draw]`.

\pattern

Inside `{tikzpicture}` this is an abbreviation for `\path[pattern]`.

\shade

Inside `{tikzpicture}` this is an abbreviation for `\path[shade]`.

\shadedraw

Inside `{tikzpicture}` this is an abbreviation for `\path[shade,draw]`.

\clip

Inside `{tikzpicture}` this is an abbreviation for `\path[clip]`.

\useasboundingbox

Inside `{tikzpicture}` this is an abbreviation for `\path[use as bounding box]`.

15.2 Specifying a Color

The most unspecific option for setting colors is the following:

`/tikz/color=(color name)` (no default)

This option sets the color that is used for fill, drawing, and text inside the current scope. Any special settings for filling colors or drawing colors are immediately “overruled” by this option.

The `(color name)` is the name of a previously defined color. For L^AT_EX users, this is just a normal “L^AT_EX-color” and the `xcolor` extensions are allowed. Here is an example:

 `\tikz \fill[color=red!20] (0,0) circle (1ex);`

It is possible to “leave out” the `color=` part and you can also write:

 `\tikz \fill[red!20] (0,0) circle (1ex);`

What happens is that every option that TikZ does not know, like `red!20`, gets a “second chance” as a color name.

For plain T_EX users, it is not so easy to specify colors since plain T_EX has no “standardized” color naming mechanism. Because of this, PGF emulates the `xcolor` package, though the emulation is *extremely basic* (more precisely, what I could hack together in two hours or so). The emulation allows you to do the following:

- Specify a new color using `\definecolor`. Only the color models `gray`, `rgb`, and `RGB` are supported³.
Example: `\definecolor{orange}{rgb}{1,0.5,0}`
- Use `\colorlet` to define a new color based on an old one. Here, the `!` mechanism is supported, though only “once” (use multiple `\colorlet` for more fancy colors).
Example: `\colorlet{lightgray}{black!25}`
- Use `\color{<color name>}` to set the color in the current T_EX group. `\aftergroup`-hackery is used to restore the color after the group.

As pointed out above, the `color=` option applies to “everything” (except to shadings), which is not always what you want. Because of this, there are several more specialized color options. For example, the `draw=` option sets the color used for drawing, but does not modify the color used for filling. These color options are documented where the path action they influence is described.

15.3 Drawing a Path

You can draw a path using the following option:

`/tikz/draw=(color)` (default is scope’s color setting)

Causes the path to be drawn. “Drawing” (also known as “stroking”) can be thought of as picking up a pen and moving it along the path, thereby leaving “ink” on the canvas.

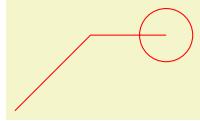
There are numerous parameters that influence how a line is drawn, like the thickness or the dash pattern. These options are explained below.

If the optional `(color)` argument is given, drawing is done using the given `(color)`. This color can be different from the current filling color, which allows you to draw and fill a path with different colors. If no `(color)` argument is given, the last usage of the `color=` option is used.

If the special color name `none` is given, this option causes drawing to be “switched off”. This is useful if a style has previously switched on drawing and you locally wish to undo this effect.

Although this option is normally used on paths to indicate that the path should be drawn, it also makes sense to use the option with a `{scope}` or `{tikzpicture}` environment. However, this will *not* cause all paths to be drawn. Instead, this just sets the `(color)` to be used for drawing paths inside the environment.

³ConT_EXt users should be aware that `\definecolor` has a different meaning in ConT_EXt. There is a low-level equivalent named `\pgfutil@definecolor` which can be used instead.



```
\begin{tikzpicture}
  \path[draw=red] (0,0) -- (1,1) -- (2,1) circle (10pt);
\end{tikzpicture}
```

The following subsections list the different options that influence how a path is drawn. All of these options only have an effect if the `draw` option is given (directly or indirectly).

15.3.1 Graphic Parameters: Line Width, Line Cap, and Line Join

`/tikz/line width=<dimension>`

(no default, initially `0.4pt`)

Specifies the line width. Note the space.



```
\tikz \draw[line width=5pt] (0,0) -- (1cm,1.5ex);
```

There are a number of predefined styles that provide more “natural” ways of setting the line width. You can also redefine these styles.

`/tikz/ultra thin`

(style, no value)

Sets the line width to 0.1pt.



```
\tikz \draw[ultra thin] (0,0) -- (1cm,1.5ex);
```

`/tikz/very thin`

(style, no value)

Sets the line width to 0.2pt.



```
\tikz \draw[very thin] (0,0) -- (1cm,1.5ex);
```

`/tikz/thin`

(style, no value)

Sets the line width to 0.4pt.



```
\tikz \draw[thin] (0,0) -- (1cm,1.5ex);
```

`/tikz/semithick`

(style, no value)

Sets the line width to 0.6pt.



```
\tikz \draw[semithick] (0,0) -- (1cm,1.5ex);
```

`/tikz/thick`

(style, no value)

Sets the line width to 0.8pt.



```
\tikz \draw[thick] (0,0) -- (1cm,1.5ex);
```

`/tikz/very thick`

(style, no value)

Sets the line width to 1.2pt.



```
\tikz \draw[very thick] (0,0) -- (1cm,1.5ex);
```

`/tikz/ultra thick`

(style, no value)

Sets the line width to 1.6pt.

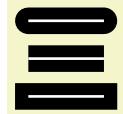


```
\tikz \draw[ultra thick] (0,0) -- (1cm,1.5ex);
```

`/tikz/line cap=<type>`

(no default, initially `butt`)

Specifies how lines “end”. Permissible `<type>` are `round`, `rect`, and `butt`. They have the following effects:



```
\begin{tikzpicture}
\begin{scope}[line width=10pt]
\draw[line cap=round] (0,1) -- +(1,0);
\draw[line cap=butt] (0,.5) -- +(1,0);
\draw[line cap=rect] (0,0) -- +(1,0);
\end{scope}
\draw[white, line width=1pt]
(0,0) -- +(1,0) (0,.5) -- +(1,0) (0,1) -- +(1,0);
\end{tikzpicture}
```

`/tikz/line join=<type>`

(no default, initially `miter`)

Specifies how lines “join”. Permissible `<type>` are `round`, `bevel`, and `miter`. They have the following effects:



```
\begin{tikzpicture}[line width=10pt]
\draw[line join=round] (0,0) -- ++(.5,1) -- ++(.5,-1);
\draw[line join=bevel] (1.25,0) -- ++(.5,1) -- ++(.5,-1);
\draw[line join=miter] (2.5,0) -- ++(.5,1) -- ++(.5,-1);
\useasboundingbox (0,1.5); % enlarge bounding box
\end{tikzpicture}
```

`/tikz/miter limit=<factor>`

(no default, initially 10)

When you use the miter join and there is a very sharp corner (a small angle), the miter join may protrude very far over the actual joining point. In this case, if it were to protrude by more than `<factor>` times the line width, the miter join is replaced by a bevel join.



```
\begin{tikzpicture}[line width=5pt]
\draw (0,0) -- ++(5,.5) -- ++(-5,.5);
\draw[miter limit=25] (6,0) -- ++(5,.5) -- ++(-5,.5);
\useasboundingbox (14,0); % make bounding box bigger
\end{tikzpicture}
```

15.3.2 Graphic Parameters: Dash Pattern

`/tikz/dash pattern=<dash pattern>`

(no default)

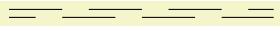
Sets the dashing pattern. The syntax is the same as in METAFONT. For example following pattern `on 2pt off 3pt on 4pt off 4pt` means “draw 2pt, then leave out 3pt, then draw 4pt once more, then leave out 4pt again, repeat”.



```
\begin{tikzpicture}[dash pattern=on 2pt off 3pt on 4pt off 4pt]
\draw (0pt,0pt) -- (3.5cm,0pt);
\end{tikzpicture}
```

`/tikz/dash phase=<dash phase>` (no default, initially 0pt)

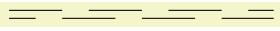
Shifts the start of the dash pattern by `<phase>`.



```
\begin{tikzpicture} [dash pattern=on 20pt off 10pt]
  \draw [dash phase=0pt] (0pt,3pt) -- (3.5cm,3pt);
  \draw [dash phase=10pt] (0pt,0pt) -- (3.5cm,0pt);
\end{tikzpicture}
```

`/tikz/dash=<dash pattern>phase<dash phase>` (no default)

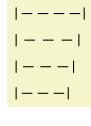
Sets the dashing pattern and phase at the same time.



```
\begin{tikzpicture}
  \draw [dash=on 20pt off 10pt phase 0pt] (0pt,3pt) -- (3.5cm,3pt);
  \draw [dash=on 20pt off 10pt phase 10pt] (0pt,0pt) -- (3.5cm,0pt);
\end{tikzpicture}
```

`/tikz/dash expand off` (no value)

Makes the `off` part of a dash pattern expandable such that it can stretch. This only works when there is a single `on` and a single `off` field and requires the `decorations` library. Right now this option has to be specified on the path where it is supposed to take effect after the `dash pattern` option because the dash pattern has to be known at the point where it is applied.



```
\usetikzlibrary {decorations}
\begin{tikzpicture} [|-|, dash pattern=on 4pt off 2pt]
  \draw [dash expand off] (0pt,30pt) -- (26pt,30pt);
  \draw [dash expand off] (0pt,20pt) -- (24pt,20pt);
  \draw [dash expand off] (0pt,10pt) -- (22pt,10pt);
  \draw [dash expand off] (0pt, 0pt) -- (20pt, 0pt);
\end{tikzpicture}
```

As for the line thickness, some predefined styles allow you to set the dashing conveniently.

`/tikz/solid` (style, no value)

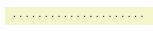
Shorthand for setting a solid line as “dash pattern”. This is the default.



```
\tikz \draw[solid] (0pt,0pt) -- (50pt,0pt);
```

`/tikz/dotted` (style, no value)

Shorthand for setting a dotted dash pattern.



```
\tikz \draw[dotted] (0pt,0pt) -- (50pt,0pt);
```

`/tikz/densely dotted` (style, no value)

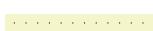
Shorthand for setting a densely dotted dash pattern.



```
\tikz \draw[densely dotted] (0pt,0pt) -- (50pt,0pt);
```

`/tikz/loosely dotted` (style, no value)

Shorthand for setting a loosely dotted dash pattern.



```
\tikz \draw[loosely dotted] (0pt,0pt) -- (50pt,0pt);
```

`/tikz/dashed` (style, no value)

Shorthand for setting a dashed dash pattern.



```
\tikz \draw[dashed] (0pt,0pt) -- (50pt,0pt);
```

`/tikz/densely dashed` (style, no value)

Shorthand for setting a densely dashed dash pattern.

 `\tikz \draw[densely dashed] (0pt,0pt) -- (50pt,0pt);`

`/tikz/loosely dashed` (style, no value)

Shorthand for setting a loosely dashed dash pattern.

 `\tikz \draw[loosely dashed] (0pt,0pt) -- (50pt,0pt);`

`/tikz/dash dot` (style, no value)

Shorthand for setting a dashed and dotted dash pattern.

 `\tikz \draw[dash dot] (0pt,0pt) -- (50pt,0pt);`

`/tikz/densely dash dot` (style, no value)

Shorthand for setting a densely dashed and dotted dash pattern.

 `\tikz \draw[densely dash dot] (0pt,0pt) -- (50pt,0pt);`

`/tikz/loosely dash dot` (style, no value)

Shorthand for setting a loosely dashed and dotted dash pattern.

 `\tikz \draw[loosely dash dot] (0pt,0pt) -- (50pt,0pt);`

`/tikz/dash dot dot` (style, no value)

Shorthand for setting a dashed and dotted dash pattern with more dots.

 `\tikz \draw[dash dot dot] (0pt,0pt) -- (50pt,0pt);`

`/tikz/densely dash dot dot` (style, no value)

Shorthand for setting a densely dashed and dotted dash pattern with more dots.

 `\tikz \draw[densely dash dot dot] (0pt,0pt) -- (50pt,0pt);`

`/tikz/loosely dash dot dot` (style, no value)

Shorthand for setting a loosely dashed and dotted dash pattern with more dots.

 `\tikz \draw[loosely dash dot dot] (0pt,0pt) -- (50pt,0pt);`

15.3.3 Graphic Parameters: Draw Opacity

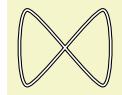
When a line is drawn, it will normally “obscure” everything behind it as if you had used perfectly opaque ink. It is also possible to ask TikZ to use an ink that is a little bit (or a big bit) transparent using the `draw opacity` option. This is explained in Section 23 on transparency in more detail.

15.3.4 Graphic Parameters: Double Lines and Bordered Lines

`/tikz/double=<core color>`

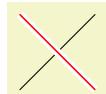
(default white)

This option causes “two” lines to be drawn instead of a single one. However, this is not what really happens. In reality, the path is drawn twice. First, with the normal drawing color, secondly with the `<core color>`, which is normally `white`. Upon the second drawing, the line width is reduced. The net effect is that it appears as if two lines had been drawn and this works well even with complicated, curved paths:



```
\tikz \draw[double]
  plot[smooth cycle] coordinates{(0,0) (1,1) (1,0) (0,1)};
```

You can also use the doubling option to create an effect in which a line seems to have a certain “border”:

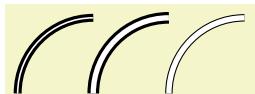


```
\begin{tikzpicture}
  \draw (0,0) -- (1,1);
  \draw[draw=white,double=red,very thick] (0,1) -- (1,0);
\end{tikzpicture}
```

`/tikz/double distance=<dimension>`

(no default, initially 0.6pt)

Sets the distance the “two” lines are spaced apart. In reality, this is the thickness of the line that is used to draw the path for the second time. The thickness of the *first* time the path is drawn is twice the normal line width plus the given `<dimension>`. As a side-effect, this option “selects” the `double` option.

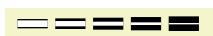


```
\begin{tikzpicture}
  \draw[very thick,double] (0,0) arc (180:90:1cm);
  \draw[very thick,double distance=2pt] (1,0) arc (180:90:1cm);
  \draw[thin,double distance=2pt] (2,0) arc (180:90:1cm);
\end{tikzpicture}
```

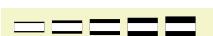
`/tikz/double distance between line centers=<dimension>`

(no default)

This option works like `double distance`, only the distance is not the distance between (inner) borders of the two main lines, but between their centers. Thus, the thickness the *first* time the path is drawn is the normal line width plus the given `<dimension>`, while the line width of the *second* line that is drawn is `<dimension>` minus the normal line width. As a side-effect, this option “selects” the `double` option.



```
\begin{tikzpicture}[double distance between line centers=3pt]
  \foreach \lw in {0.5,1,1.5,2,2.5}
    \draw[line width=\lw pt,double] (\lw,0) -- ++(4mm,0);
\end{tikzpicture}
```

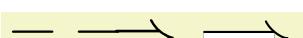


```
\begin{tikzpicture}[double distance=3pt]
  \foreach \lw in {0.5,1,1.5,2,2.5}
    \draw[line width=\lw pt,double] (\lw,0) -- ++(4mm,0);
\end{tikzpicture}
```

`/tikz/double equal sign distance`

(style, no value)

This style selects a double line distance such that it corresponds to the distance of the two lines in an equal sign.



```
\usepackage{amsmath} \usetikzlibrary{arrows.meta}
\Huge $=\!\implies\$ \tikz [baseline, double equal sign distance]
  \draw[double, thick, -{Implies[]}] (0,0.55ex) --+(3ex,0);
```



```
\usepackage{amsmath} \usetikzlibrary{arrows.meta}
\normalsize $=\!\implies\$ \tikz [baseline, double equal sign distance]
  \draw[double, -{Implies[]}] (0,0.6ex) --+(3ex,0);
```

```
= ==> 
\usepackage{amsmath} \usetikzlibrary{arrows.meta}
\tiny $=\!\implies\$ \tikz[baseline,double equal sign distance]
\draw[double,very thin,-{Implies[]}] (0,0.5ex) -- ++(3ex,0);
```

15.4 Adding Arrow Tips to a Path

In different situations, TikZ will add arrow tips to the end of a path. For this to happen, a number of different things need to be specified:

1. You must have used the `arrows` key, explained in detail in Section 16, to setup which kinds of arrow tips you would like.
2. The path may not be closed (like a circle or a rectangle) and, if it consists of several subpath, further restrictions apply as explained in Section 16.
3. The `tips` key must be set to an appropriate value, see Section 16 once more.

For the current section on paths, it is only important that when you add the `tips` option to a path that is not drawn, arrow tips will still be added at the beginning and at the end of the current path. This is true even when “only” arrow tips get drawn for a path without drawing the path itself. Here is an example:



```
\usepackage{amsmath} \usetikzlibrary{arrows.meta,bending}
\tikz \path[tips, -{Latex[open,length=10pt,bend]}] (0,0) to[bend left] (1,0);
```



```
\usepackage{amsmath} \usetikzlibrary{arrows.meta,bending}
\tikz \draw[tips, -{Latex[open,length=10pt,bend]}] (0,0) to[bend left] (1,0);
```

15.5 Filling a Path

To fill a path, use the following option:

`/tikz/fill=<color>` (default is scope's color setting)

This option causes the path to be filled. All unclosed parts of the path are first closed, if necessary. Then, the area enclosed by the path is filled with the current filling color, which is either the last color set using the general `color=` option or the optional `color <color>`. For self-intersection paths and for paths consisting of several closed areas, the “enclosed area” is somewhat complicated to define and two different definitions exist, namely the nonzero winding number rule and the even odd rule, see the explanation of these options, below.

Just as for the `draw` option, setting `<color>` to `none` disables filling locally.



```
\begin{tikzpicture}
\fill (0,0) -- (1,1) -- (2,1);
\fill (4,0) circle (.5cm) (4.5,0) circle (.5cm);
\fill[even odd rule] (6,0) circle (.5cm) (6.5,0) circle (.5cm);
\fill (8,0) -- (9,1) -- (10,0) circle (.5cm);
\end{tikzpicture}
```

If the `fill` option is used together with the `draw` option (either because both are given as options or because a `\filldraw` command is used), the path is filled *first*, then the path is drawn *second*. This is especially useful if different colors are selected for drawing and for filling. Even if the same color is used, there is a difference between this command and a plain `fill`: A “filldrawn” area will be slightly larger than a filled area because of the thickness of the “pen”.



```
\begin{tikzpicture}[fill=yellow!80!black, line width=5pt]
\filldraw (0,0) -- (1,1) -- (2,1);
\filldraw (4,0) circle (.5cm) (4.5,0) circle (.5cm);
\filldraw[even odd rule] (6,0) circle (.5cm) (6.5,0) circle (.5cm);
\filldraw (8,0) -- (9,1) -- (10,0) circle (.5cm);
\end{tikzpicture}
```

15.5.1 Graphic Parameters: Fill Pattern

Instead of filling a path with a single solid color, it is also possible to fill it with a *tiling pattern*. Imagine a small tile that contains a simple picture like a star. Then these tiles are (conceptually) repeated infinitely in all directions, but clipped against the path.

Tiling patterns come in two variants: *inherently colored patterns* and *form-only patterns*. An inherently colored pattern is, say, a red star with a black border and will always look like this. A form-only pattern may have a different color each time it is used, only the form of the pattern will stay the same. As such, form-only patterns do not have any colors of their own, but when it is used the current *pattern color* is used as its color.

Patterns are not overly flexible. In particular, it is not possible to change the size or orientation of a pattern without declaring a new pattern. For complicated cases, it may be easier to use two nested `\foreach` statements to simulate a pattern, but patterns are rendered *much* more quickly than simulated ones.

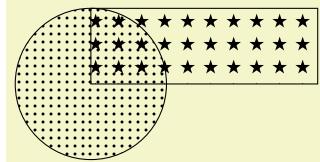
`/tikz/pattern=<name>`

(default is scope's pattern)

This option causes the path to be filled with a pattern. If the `<name>` is given, this pattern is used, otherwise the pattern set in the enclosing scope is used. As for the `draw` and `fill` options, setting `<name>` to `none` disables filling locally.

The pattern works like a fill color. In particular, setting a new fill color will fill the path with a solid color once more.

Strangely, no `<name>`s are permissible by default. You need to load for instance the `patterns` library, see Section 63, to install predefined patterns.

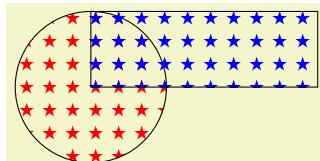


```
\usetikzlibrary {patterns}
\begin{tikzpicture}
\draw[pattern=dots] (0,0) circle (1cm);
\draw[pattern=fivepointed stars] (0,0) rectangle (3,1);
\end{tikzpicture}
```

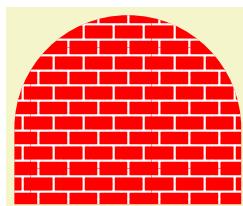
`/tikz/pattern color=<color>`

(no default)

This option is used to set the color to be used for form-only patterns. This option has no effect on inherently colored patterns.



```
\usetikzlibrary {patterns}
\begin{tikzpicture}
\draw[pattern color=red,pattern=fivepointed stars] (0,0) circle (1cm);
\draw[pattern color=blue,pattern=fivepointed stars] (0,0) rectangle (3,1);
\end{tikzpicture}
```



```
\usetikzlibrary {patterns}
\begin{tikzpicture}
\def\mypath{(0,0) -- +(0,1) arc (180:0:1.5cm) -- +(0,-1)}
\fill [red] \mypath;
\pattern[pattern color=white,pattern=bricks] \mypath;
\end{tikzpicture}
```

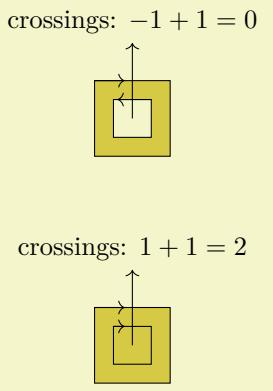
15.5.2 Graphic Parameters: Interior Rules

The following two options can be used to decide how interior points should be determined:

`/tikz/nonzero rule`

(no value)

If this rule is used (which is the default), the following method is used to determine whether a given point is “inside” the path: From the point, shoot a ray in some direction towards infinity (the direction is chosen such that no strange borderline cases occur). Then the ray may hit the path. Whenever it hits the path, we increase or decrease a counter, which is initially zero. If the ray hits the path as the path goes “from left to right” (relative to the ray), the counter is increased, otherwise it is decreased. Then, at the end, we check whether the counter is nonzero (hence the name). If so, the point is deemed to lie “inside”, otherwise it is “outside”. Sounds complicated? It is.



```
\begin{tikzpicture}
\filldraw[fill=yellow!80!black]
% Clockwise rectangle
(0,0) -- (0,1) -- (1,1) -- (1,0) -- cycle
% Counter-clockwise rectangle
(0.25,0.25) -- (0.75,0.25) -- (0.75,0.75) -- (0.25,0.75) -- cycle;

\draw[->] (0,1) -- (.4,1);
\draw[->] (0.75,0.75) -- (0.3,.75);

\draw[->] (0.5,0.5) -- +(0,1) node[above] {crossings: $-1+1 = 0$};

\begin{scope}[yshift=-3cm]
\filldraw[fill=yellow!80!black]
% Clockwise rectangle
(0,0) -- (0,1) -- (1,1) -- (1,0) -- cycle
% Clockwise rectangle
(0.25,0.25) -- (0.25,0.75) -- (0.75,0.75) -- (0.75,0.25) -- cycle;

\draw[->] (0,1) -- (.4,1);
\draw[->] (0.25,0.75) -- (0.4,.75);

\draw[->] (0.5,0.5) -- +(0,1) node[above] {crossings: $1+1 = 2$};
\end{scope}
\end{tikzpicture}
```

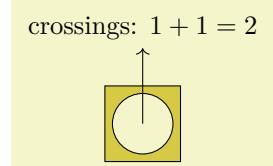
`/tikz/even odd rule`

(no value)

This option causes a different method to be used for determining the inside and outside of paths. While it is less flexible, it turns out to be more intuitive.

With this method, we also shoot rays from the point for which we wish to determine whether it is inside or outside the filling area. However, this time we only count how often we “hit” the path and declare the point to be “inside” if the number of hits is odd.

Using the even-odd rule, it is easy to “drill holes” into a path.



```
\begin{tikzpicture}
\filldraw[fill=yellow!80!black,even odd rule]
(0,0) rectangle (1,1) (0.5,0.5) circle (0.4cm);
\draw[->] (0.5,0.5) -- +(0,1) [above] node{crossings: $1+1 = 2$};
\end{tikzpicture}
```

15.5.3 Graphic Parameters: Fill Opacity

Analogously to the `draw opacity`, you can also set the fill opacity. Please see Section 23 for more details.

15.6 Generalized Filling: Using Arbitrary Pictures to Fill a Path

Sometimes you wish to “fill” a path with something even more complicated than a pattern, let alone a single color. For instance, you might wish to use an image to fill the path or some other, complicated

drawing. In principle, this effect can be achieved by first using the path for clipping and then, subsequently, drawing the desired image or picture. However, there is an option that makes this process much easier:

`/tikz/path picture=<code>` (no default)

When this option is given on a path and when the `<code>` is not empty, the following happens: After all other “filling” operations are done with the path, which are caused by the options `fill`, `pattern` and `shade`, a local scope is opened and the path is temporarily installed as a clipping path. Then, the `<code>` is executed, which can now draw something. Then, the local scope ends and, possibly, the path is stroked, provided the `draw` option has been given.

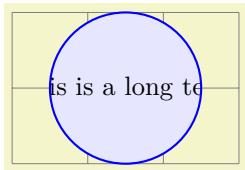
As with other keys like `fill` or `draw` this option needs to be given on a path, setting the `path picture` outside a path has no effect (the path picture is cleared at the beginning of each path).

The `<code>` can be any normal TikZ code like `\draw ...` or `\node ...`. As always, when you include an external graphic, you need to put it inside a `\node`.

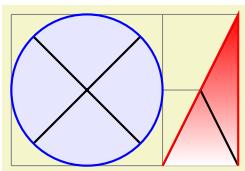
Note that no special actions are taken to transform the origin in any way. This means that the coordinate $(0,0)$ is still where it was when the path was being constructed and not – as one might expect – at the lower left corner of the path. However, you can use the following special node to access the size of the path:

Predefined node `path picture bounding box`

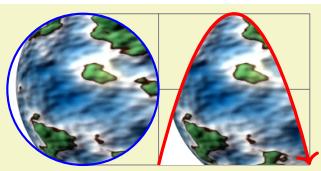
This node is of shape `rectangle`. Its size and position are those of `current path bounding box` just before the `<code>` of the path picture started to be executed. The `<code>` can construct its own paths, so accessing the `current path bounding box` inside the `<code>` yields the bounding box of any path that is currently being constructed inside the `<code>`.



```
\begin{tikzpicture}
\draw [help lines] (0,0) grid (3,2);
\filldraw [fill=blue!10,draw=blue,thick] (1.5,1) circle (1)
  [path picture={
    \node at (path picture bounding box.center) {
      This is a long text.
    };
  }];
\end{tikzpicture}
```



```
\begin{tikzpicture}[cross/.style={path picture={
  \draw[black]
    (path picture bounding box.south east) --
    (path picture bounding box.north west)
    (path picture bounding box.south west) --
    (path picture bounding box.north east);
}}]
\draw [help lines] (0,0) grid (3,2);
\filldraw [cross,fill=blue!10,draw=blue,thick] (1,1) circle (1);
\path [cross,top color=red,draw=red,thick] (2,0) -- (3,2) -- (3,0);
\end{tikzpicture}
```



```
\begin{tikzpicture}[path image/.style={%
    path picture={%
        \node at (path picture bounding box.center) {%
            \includegraphics[height=3cm]{#1}%
        };}%
    }%
]
\draw [help lines] (0,0) grid (3,2);
\draw [path image=./text-zh/images/brave-gnu-world-logo,draw=blue,thick]
(0,1) circle (1);
\draw [path image=./text-zh/images/brave-gnu-world-logo,draw=red,very thick,->]
(1,0) parabola[parabola height=2cm] (3,0);

\end{tikzpicture}
```

15.7 Shading a Path

You can shade a path using the `shade` option. A shading is like a filling, only the shading changes its color smoothly from one color to another.

`/tikz/shade` (no value)

Causes the path to be shaded using the currently selected shading (more on this later). If this option is used together with the `draw` option, then the path is first shaded, then drawn.

It is not an error to use this option together with the `fill` option, but it makes no sense.

 \tikz \shade (0,0) circle (1ex);
 \tikz \shadedraw (0,0) circle (1ex);

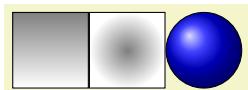
For some shadings it is not really clear how they can “fill” the path. For example, the `ball` shading normally looks like this: ●. How is this supposed to shade a rectangle? Or a triangle?

To solve this problem, the predefined shadings like `ball` or `axis` fill a large rectangle completely in a sensible way. Then, when the shading is used to “shade” a path, what actually happens is that the path is temporarily used for clipping and then the rectangular shading is drawn, scaled and shifted such that all parts of the path are filled.

The default shading is a smooth transition from gray to white and from top to bottom. However, other shadings are also possible, for example a shading that will sweep a color from the center to the corners outward. To choose the shading, you can use the `shading=` option, which will also automatically invoke the `shade` option. Note that this does *not* change the shading color, only the way the colors sweep. For changing the colors, other options are needed, which are explained below.

`/tikz/shading=<name>` (no default)

This selects a shading named `<name>`. The following shadings are predefined: `axis`, `radial`, and `ball`.



```
\tikz \shadedraw [shading=axis] (0,0) rectangle (1,1);
\tikz \shadedraw [shading=radial] (0,0) rectangle (1,1);
\tikz \shadedraw [shading=ball] (0,0) circle (.5cm);
```

The shadings as well as additional shadings are described in more detail in Section 70.

To change the color of a shading, special options are needed like `left color`, which sets the color of an axis shading from left to right. These options implicitly also select the correct shading type, see the following example



```
\tikz \shadedraw [left color=red,right color=blue]
(0,0) rectangle (1,1);
```

For a complete list of the possible options see Section 70 once more.

```
/tikz/shading angle=<degrees>
```

(no default, initially 0)

This option rotates the shading (not the path!) by the given angle. For example, we can turn a top-to-bottom axis shading into a left-to-right shading by rotating it by 90°.



```
\tikz \shadedraw [shading=axis, shading angle=90] (0,0) rectangle (1,1);
```

You can also define new shading types yourself. However, for this, you need to use the basic layer directly, which is, well, more basic and harder to use. Details on how to create a shading appropriate for filling paths are given in Section ??.

15.8 Establishing a Bounding Box

PGF is reasonably good at keeping track of the size of your picture and reserving just the right amount of space for it in the main document. However, in some cases you may want to say things like “do not count this for the picture size” or “the picture is actually a little large”. For this you can use the option `use as bounding box` or the command `\useasboundingbox`, which is just a shorthand for `\path[use as bounding box]`.

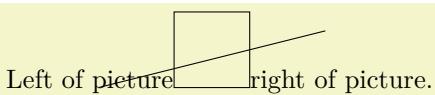
```
/tikz/use as bounding box
```

(no value)

Normally, when this option is given on a path, the bounding box of the present path is used to determine the size of the picture and the size of all *subsequent* paths are ignored. However, if there were previous path operations that have already established a larger bounding box, it will not be made smaller by this operation (consider the `\pgfresetboundingbox` command to reset the previous bounding box).

In a sense, `use as bounding box` has the same effect as clipping all subsequent drawing against the current path – without actually doing the clipping, only making PGF treat everything as if it were clipped.

The first application of this option is to have a `{tikzpicture}` overlap with the main text:



```
Left of picture\begin{tikzpicture}
  \draw[use as bounding box] (2,0) rectangle (3,1);
  \draw (1,0) -- (4,.75);
\end{tikzpicture}right of picture.
```

In a second application this option can be used to get better control over the white space around the picture:



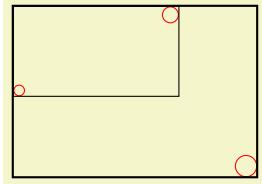
```
Left of picture
\begin{tikzpicture}
  \useasboundingbox (0,0) rectangle (3,1);
  \fill (.75,.25) circle (.5cm);
\end{tikzpicture}right of picture.
```

Note: If this option is used on a path inside a TeX group (scope), the effect “lasts” only until the end of the scope. Again, this behavior is the same as for clipping.

Consider using `\useasboundingbox` together with `\pgfresetboundingbox` in order to replace the bounding box with a new one.

There is a node that allows you to get the size of the current bounding box. The `current bounding box` node has the `rectangle` shape and its size is always the size of the current bounding box.

Similarly, the `current path bounding box` node has the `rectangle` shape and the size of the bounding box of the current path.



```
\begin{tikzpicture}
  \draw[red] (0,0) circle (2pt);
  \draw[red] (2,1) circle (3pt);

  \draw (current bounding box.south west) rectangle
        (current bounding box.north east);

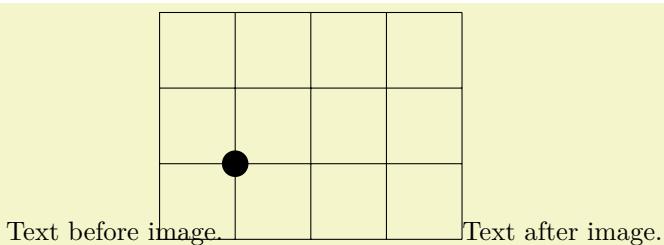
  \draw[red] (3,-1) circle (4pt);

  \draw[thick] (current bounding box.south west) rectangle
        (current bounding box.north east);
\end{tikzpicture}
```

Occasionally, you may want to align multiple `tikzpicture` environments horizontally and/or vertically at some prescribed position. The vertical alignment can be realized by means of the `baseline` option since TeX supports the concept of box depth natively. For horizontal alignment, things are slightly more involved. The following approach is realized by means of negative `\hspace{-s}` before and/or after the picture, thereby removing parts of the picture. However, the actual amount of negative horizontal space is provided by means of image coordinates using the `trim left` and `trim right` keys:

`/tikz/trim left=<dimension or coordinate or default>` (default 0pt)

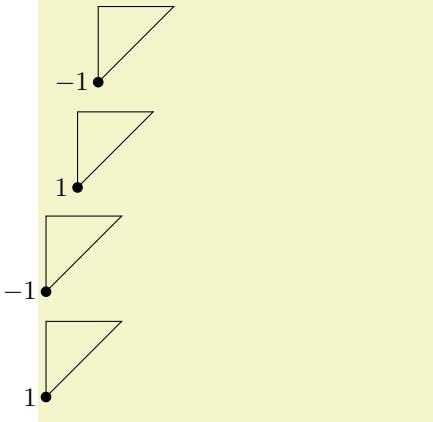
The `trim left` key tells PGF to discard everything which is left of the provided `<dimension or coordinate>`. Here, `<dimension>` is a single x coordinate of the picture and `<coordinate>` is a point with x and y coordinates (but only its x coordinate will be used). The effect is the same as if you issue `\hspace{-s}` where s is the difference of the picture's bounding box lower left x coordinate and the x coordinate specified as `<dimension or coordinate>`:



```
Text before image.%
\begin{tikzpicture}[trim left]
  \draw (-1,-1) grid (3,2);
  \fill (0,0) circle (5pt);
\end{tikzpicture}%
Text after image.
```

Since `trim left` uses the default `trim left=0pt`, everything left of $x = 0$ is removed from the bounding box.

The following example has once the relative long label `-1` and once the shorter label `1`. Horizontal alignment is established with `trim left`:



```
\begin{tikzpicture}
  \draw (0,1) -- (0,0) -- (1,1) -- cycle;
  \fill (0,0) circle (2pt);
  \node[left] at (0,0) {$-1$};
\end{tikzpicture}
\par
\begin{tikzpicture}
  \draw (0,1) -- (0,0) -- (1,1) -- cycle;
  \fill (0,0) circle (2pt);
  \node[left] at (0,0) {$1$};
\end{tikzpicture}
\par
\begin{tikzpicture}[trim left]
  \draw (0,1) -- (0,0) -- (1,1) -- cycle;
  \fill (0,0) circle (2pt);
  \node[left] at (0,0) {$-1$};
\end{tikzpicture}
\par
\begin{tikzpicture}[trim left]
  \draw (0,1) -- (0,0) -- (1,1) -- cycle;
  \fill (0,0) circle (2pt);
  \node[left] at (0,0) {$1$};
\end{tikzpicture}
```

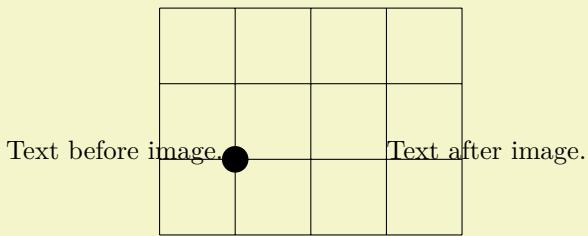
Use `trim left=default` to reset the value.

`/tikz/trim right=<dimension or coordinate or default>`

(no default)

This key is similar to `trim left`: it discards everything which is right of the provided `<dimension or coordinate>`. As for `trim left`, `<dimension>` denotes a single x coordinate of the picture and `<coordinate>` a coordinate with x and y value (although only its x component will be used).

We use the same example from above and add `trim right`:



```
Text before image.%
\begin{tikzpicture}[trim left, trim right=2cm, baseline]
  \draw (-1,-1) grid (3,2);
  \fill (0,0) circle (5pt);
\end{tikzpicture}%
Text after image.
```

In addition to `trim left=0pt`, we also discard everything which is right of $x=2cm$. Furthermore, the `baseline` key supports vertical alignment as well (using the $y=0cm$ baseline).

Use `trim right=default` to reset the value.

Note that `baseline`, `trim left` and `trim right` are currently the *only* supported way of truncated bounding boxes which are compatible with image externalization (see the `external` library for details).

`/pgf/trim lowlevel=true|false`

(no default, initially `false`)

This affects only the basic level image externalization: the initial configuration `trim lowlevel=false` stores the normal image, without trimming, and the trimming into a separate file. This allows reduced bounding boxes without clipping the rest away. The `trim lowlevel=true` information causes the image externalization to store the trimmed image, possibly resulting in clipping.

15.9 Clipping and Fading (Soft Clipping)

Clipping path means that all painting on the page is restricted to a certain area. This area need not be rectangular, rather an arbitrary path can be used to specify this area. The `clip` option, explained below, is used to specify the region that is to be used for clipping.

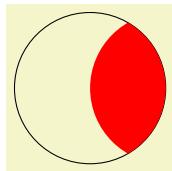
A *fading* (a term that I propose, fadings are commonly known as soft masks, transparency masks, opacity masks or soft clips) is similar to clipping, but a fading allows parts of the picture to be only “half clipped”. This means that a fading can specify that newly painted pixels should be partly transparent. The specification and handling of fadings is a bit complex and it is detailed in Section 23, which is devoted to transparency in general.

/tikz/clip

(no value)

This option causes all subsequent drawings to be clipped against the current path and the size of subsequent paths will not be important for the picture size. If you clip against a self-intersecting path, the even-odd rule or the nonzero winding number rule is used to determine whether a point is inside or outside the clipping region.

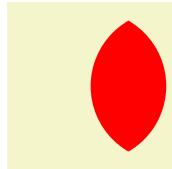
The clipping path is a graphic state parameter, so it will be reset at the end of the current scope. Multiple clippings accumulate, that is, clipping is always done against the intersection of all clipping areas that have been specified inside the current scopes. The only way of enlarging the clipping area is to end a `{scope}`.



```
\begin{tikzpicture}
  \draw[clip] (0,0) circle (1cm);
  \fill[red] (1,0) circle (1cm);
\end{tikzpicture}
```

It is usually a *very* good idea to apply the `clip` option only to the first path command in a scope.

If you “only wish to clip” and do not wish to draw anything, you can use the `\clip` command, which is a shorthand for `\path[clip]`.



```
\begin{tikzpicture}
  \clip (0,0) circle (1cm);
  \fill[red] (1,0) circle (1cm);
\end{tikzpicture}
```

To keep clipping local, use `{scope}` environments as in the following example:



```
\begin{tikzpicture}
  \draw (0,0) -- (0:1cm);
  \draw (0,0) -- (10:1cm);
  \draw (0,0) -- (20:1cm);
  \draw (0,0) -- (30:1cm);
  \begin{scope}[fill=red]
    \fill[clip] (0.2,0.2) rectangle (0.5,0.5);

    \draw (0,0) -- (40:1cm);
    \draw (0,0) -- (50:1cm);
    \draw (0,0) -- (60:1cm);
  \end{scope}
  \draw (0,0) -- (70:1cm);
  \draw (0,0) -- (80:1cm);
  \draw (0,0) -- (90:1cm);
\end{tikzpicture}
```

There is a slightly annoying catch: You cannot specify certain graphic options for the command used for clipping. For example, in the above code we could not have moved the `fill=red` to the `\fill` command. The reasons for this have to do with the internals of the PDF specification. You do not want to know the details. It is best simply not to specify any options for these commands.

15.10 Doing Multiple Actions on a Path

If more than one of the basic actions like drawing, clipping and filling are requested, they are automatically applied in a sensible order: First, a path is filled, then drawn, and then clipped (although it took

Apple two major revisions of their operating system to get this right...). Sometimes, however, you need finer control over what is done with a path. For instance, you might wish to first fill a path with a color, then repaint the path with a pattern and then repaint it with yet another pattern. In such cases you can use the following two options:

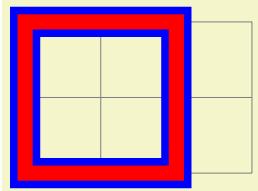
/tikz/preaction=<options>

(no default)

This option can be given to a `\path` command (or to derived commands like `\draw` which internally call `\path`). Similarly to options like `draw`, this option only has an effect when given to a `\path` or as part of the options of a `node`; as an option to a `{scope}` it has no effect.

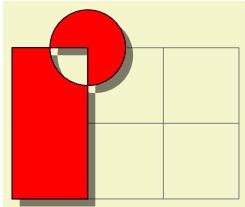
When this option is used on a `\path`, the effect is the following: When the path has been completely constructed and is about to be used, a scope is created. Inside this scope, the path is used but not with the original path options, but with `<options>` instead. Then, the path is used in the usual manner. In other words, the path is used twice: Once with `<options>` in force and then again with the normal path options in force.

Here is an example in which the path consists of a rectangle. The main action is to draw this path in red (which is why we see a red rectangle). However, the preaction is to draw the path in blue, which is why we see a blue rectangle behind the red rectangle.



```
\begin{tikzpicture}
\draw[help lines] (0,0) grid (3,2);
\draw
[preaction={draw, line width=4mm, blue}]
[line width=2mm,red] (0,0) rectangle (2,2);
\end{tikzpicture}
```

Note that when the preactions are performed, then the path is already “finished”. In particular, applying a coordinate transformation to the path has no effect. By comparison, applying a canvas transformation does have an effect. Let us use this to add a “shadow” to a path. For this, we use the preaction to fill the path in gray, shifted a bit to the right and down:

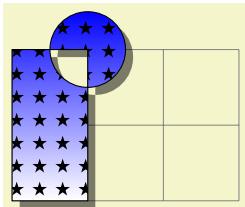


```
\begin{tikzpicture}
\draw[help lines] (0,0) grid (3,2);
\draw
[preaction={fill=black, opacity=.5,
            transform canvas={xshift=1mm, yshift=-1mm}}
 [fill=red] (0,0) rectangle (1,2)
 (1,2) circle (5mm);
\end{tikzpicture}
```

Naturally, you would normally create a style `shadow` that contains the above code. The `shadows` library, see Section 71, contains predefined shadows of this kind.

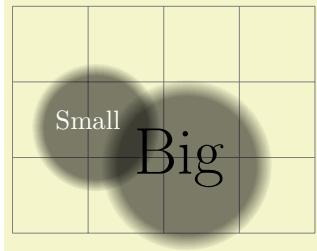
It is possible to use the `preaction` option multiple times. In this case, for each use of the `preaction` option, the path is used again (thus, the `<options>` do not accumulate in a single usage of the path). The path is used in the order of `preaction` options given.

In the following example, we use one `preaction` to add a shadow and another to provide a shading, while the main action is to use a pattern.



```
\usetikzlibrary {patterns}
\begin{tikzpicture}
\draw[help lines] (0,0) grid (3,2);
\draw [pattern=fivepointed stars]
[preaction={fill=black, opacity=.5,
            transform canvas={xshift=1mm, yshift=-1mm}}]
[preaction={top color=blue, bottom color=white}]
(0,0) rectangle (1,2)
(1,2) circle (5mm);
\end{tikzpicture}
```

A complicated application is shown in the following example, where the path is used several times with different fadings and shadings to create a special visual effect:



```
\usetikzlibrary {fadings,patterns}
\begin{tikzpicture}
[
    % Define an interesting style
    button/.style={
        % First preaction: Fuzzy shadow
        preaction={fill=black,path fading=circle with fuzzy edge 20 percent,
            opacity=.5,transform canvas={xshift=1mm,yshift=-1mm}},
        % Second preaction: Background pattern
        preaction={pattern=#1,
            path fading=circle with fuzzy edge 15 percent},
        % Third preaction: Make background shiny
        preaction={top color=white,
            bottom color=black!50,
            shading angle=45,
            path fading=circle with fuzzy edge 15 percent,
            opacity=0.2},
        % Fourth preaction: Make edge especially shiny
        preaction={path fading=fuzzy ring 15 percent,
            top color=black!5,
            bottom color=black!80,
            shading angle=45},
            inner sep=2ex
        },
        button/.default=horizontal lines light blue,
        circle
    ]
\draw [help lines] (0,0) grid (4,3);

\node [button] at (2.2,1) {\Huge Big};
\node [button=crosshatch dots light steel blue,
    text=white] at (1,1.5) {Small};
\end{tikzpicture}
```

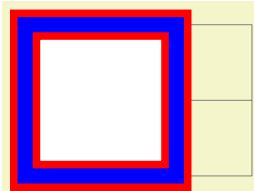
/tikz/postaction=*options*

(no default)

The postactions work in the same way as the preactions, only they are applied *after* the main action has been taken. Like preactions, multiple `postaction` options may be given to a `\path` command, in which case the path is reused several times, each time with a different set of options in force.

If both pre- and postactions are specified, then the preactions are taken first, then the main action, and then the post actions.

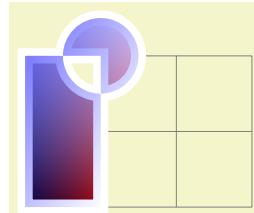
In the first example, we use a postaction to draw the path, after it has already been drawn:



```
\begin{tikzpicture}
\draw[help lines] (0,0) grid (3,2);

\draw
    [postaction={draw,line width=2mm,blue}]
    [line width=4mm,red,fill=white] (0,0) rectangle (2,2);
\end{tikzpicture}
```

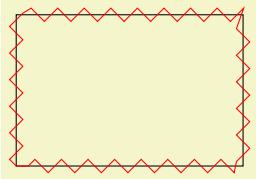
In another example, we use a postaction to “colorize” a path:



```
\usetikzlibrary {fadings}
\begin{tikzpicture}
\draw[help lines] (0,0) grid (3,2);
\draw
    [postaction={path fading=south,fill=white}]
    [postaction={path fading=south,fading angle=45,fill=blue,opacity=.5}]
    [left color=black,right color=red,draw=white,line width=2mm]
    (0,0) rectangle (1,2)
    (1,2) circle (5mm);
\end{tikzpicture}
```

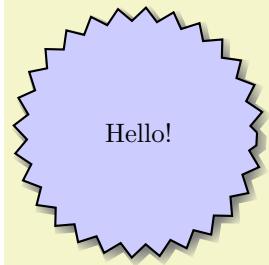
15.11 Decorating and Morphing a Path

Before a path is used, it is possible to first “decorate” and/or “morph” it. Morphing means that the path is replaced by another path that is slightly varied. Such morphings are a special case of the more general “decorations” described in detail in Section 24. For instance, in the following example the path is drawn twice: Once normally and then in a morphed (=decorated) manner.



```
\usetikzlibrary {decorations.pathmorphing}
\begin{tikzpicture}
  \draw (0,0) rectangle (3,2);
  \draw [red, decorate, decoration=zigzag]
    (0,0) rectangle (3,2);
\end{tikzpicture}
```

Naturally, we could have combined this into a single command using pre- or postaction. It is also possible to deform shapes:

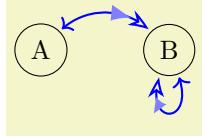


```
\usetikzlibrary {decorations.pathmorphing,shadows}
\begin{tikzpicture}
  \node [circular drop shadow={shadow scale=1.05},minimum size=3.13cm,
         decorate, decoration=zigzag,
         fill=blue!20,draw,thick,circle] {Hello!};
\end{tikzpicture}
```

16 Arrows

16.1 Overview

TikZ allows you to add (multiple) arrow tips to the end of lines as in \rightarrow or in \Rightarrow . It is possible to change which arrow tips are used “on-the-fly”, you can have several arrow tips in a row, and you can change the appearance of each of them individually using a special syntax. The following example is a perhaps slightly “excessive” demonstration of what you can do (you need to load the `arrows.meta` library for it to work):



```
\usetikzlibrary {arrows.meta,bending,positioning}
\begin{tikzpicture}
  \node [circle,draw] (A) at (0,0) {A};
  \node [circle,draw] (B) [right=of A] {B};

  \draw [draw = blue, thick,
         arrows={%
           Computer Modern Rightarrow [sep]
           - Latex[blue!50,length=8pt,bend,line width=0pt]
           Stealth[length=8pt,open,bend,sep]}]
    (A) edge [bend left=45] (B)
    (B) edge [in=-110, out=-70,looseness=8] (B);
\end{tikzpicture}
```

There are a number of predefined generic arrow tip kinds whose appearance you can modify in many ways using various options. It is also possible to define completely new arrow tip kinds, see Section ??, but doing this is somewhat harder than configuring an existing kind (it is like the difference between using a font at different sizes or faces like *italics*, compared to designing a new font yourself).

In the present section, we go over the various ways in which you can configure which particular arrow tips are *used*. The glorious details of how new arrow tips can be defined are explained in Section ??.

At the end of the present section, Section 16.5, you will find a description of the different predefined arrow tips from the `arrows.meta` library.

Remark: Almost all of the features described in the following were introduced in version 3.0 of TikZ. For compatibility reasons, the old arrow tips are still available. To differentiate between the old and new arrow tips, the following rule is used: The new, more powerful arrow tips start with an uppercase letter as in `Latex`, compared to the old arrow tip `latex`.

Remark: The libraries `arrows` and `arrows.spaced` are deprecated. Use `arrows.meta` instead/additionally, which allows you to do all that the old libraries offered, plus much more. However, the old libraries still work and you can even mix old and new arrow tips (only, the old arrow tips cannot be configured in the ways described in the rest of this section; saying `scale=2` for a `latex` arrow has no effect for instance, while for `Latex` arrows it doubles their size as one would expect.)

16.2 Where and When Arrow Tips Are Placed

In order to add arrow tips to the lines you draw, the following conditions must be met:

1. You have specified that arrow tips should be added to lines, using the `arrows` key or its short form.
2. You set the `tips` key to some value that causes tips to be drawn (to be explained later).
3. You do not use the `clip` key (directly or indirectly) with the current path.
4. The path actually has two “end points” (it is not “closed”).

Let us start with an introduction to the basics of the `arrows` key:

`/tikz/arrows=<start arrow specification>-<end arrow specification>` (no default)

This option sets the arrow tip(s) to be used at the start and end of lines. An empty value as in `->` for the start indicates that no arrow tip should be drawn at the start.

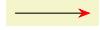
Note: Since the arrow option is so often used, you can leave out the text `arrows=`. What happens is that every (otherwise unknown) option that contains a `-` is interpreted as an arrow specification.



```
\usetikzlibrary {arrows.meta}
\begin{tikzpicture}
  \draw[->] (0,0) -- (1,0);
  \draw[>-Stealth] (0,0.3) -- (1,0.3);
\end{tikzpicture}
```

In the above example, the first start specification is empty and the second is `>`. The end specifications are `>` for the first line and `Stealth` for the second line. Note that it makes a difference whether `>` is used in a start specification or in an end specification: In an end specification it creates, as one would expect, a pointed tip at the end of the line. In the start specification, however, it creates a “reversed” version if this arrow – which happens to be what one would expect here.

The above specifications are very simple and only select a single arrow tip without any special configuration options, resulting in the “natural” versions of these arrow tips. It is also possible to “configure” arrow tips in many different ways, as explained in detail in Section 16.3 below by adding options in square brackets following the arrow tip kind:



```
\usetikzlibrary {arrows.meta}
\begin{tikzpicture}
\draw[-{Stealth[red]}] (0,0) -- (1,0);
\end{tikzpicture}
```

Note that in the example I had to surround the end specification by braces. This is necessary so that TikZ does not mistake the closing square bracket of the `Stealth` arrow tip’s options for the end of the options of the `\draw` command. In general, you often need to add braces when specifying arrow tips except for simple case like `->` or `<->`, which are pretty frequent, though. When in doubt, say `arrows={⟨start spec⟩-⟨end spec⟩}`, that will always work.

It is also possible to specify multiple (different) arrow tips in a row inside a specification, see Section 16.4 below for details.

As was pointed out earlier, to add arrow tips to a path, the path must have “end points” and not be “closed” – otherwise adding arrow tips makes little sense, after all. However, a path can actually consist of several subpath, which may be open or not and may even consist of only a single point (a single move-to). In this case, it is not immediately obvious, where arrow heads should be placed. The actual rules that TikZ uses are governed by the setting of the key `tips`:

`/pgf/tips=⟨value⟩` (default `true`, initially on `draw`)
alias `/tikz/tips`

This key governs in what situations arrow tips are added to a path. The following `⟨values⟩` are permissible:

- `true` (the value used when no `⟨value⟩` is specified)
- `proper`
- `on draw` (the initial value, if the key has not yet been used at all)
- `on proper draw`
- `never` or `false` (same effect)

Firstly, there are a whole bunch of situations where the setting of these (or other) options causes no arrow tips to be shown:

- If no arrow tips have been specified (for instance, by having said `arrows=-`), no arrow tips are drawn.
- If the `clip` option is set, no arrow tips are drawn.
- If `tips` has been set to `never` or `false`, no arrow tips are drawn.
- If `tips` has been set to `on draw` or `on proper draw`, but the `draw` option is not set, no arrow tips are drawn.
- If the path is empty (as in `\path ;`), no arrow tips are drawn.
- If at least one of the subpaths of a path is closed (`cycle` is used somewhere or something like `circle` or `rectangle`), arrow tips are never drawn anywhere – even if there are open subpaths.

Now, if we pass all of the above tests, we must have a closer look at the path. All its subpaths must now be open and there must be at least one subpath. We consider the last one. Arrow tips will only be added to this last subpath.

1. If this last subpath not degenerate (all coordinates on the subpath are the same as in a single “move-to” `\path (0,0);` or in a “move-to” followed by a “line-to” to the same position as in `\path (1,2) --(1,2)`), arrow tips are added to this last subpath now.

2. If the last subpath is degenerate, we add arrow tips pointing upward at the single coordinate mentioned in the path, but only for `tips` begin set to `true` or to `on draw` – and not for `proper` nor for `on proper draw`. In other words, “proper” suppresses arrow tips on degenerate paths.

	<pre>% No path, no arrow tips: \tikz [<->] \draw;</pre>
	<pre>% Degenerate path, draw arrow tips (but no path, it is degenerate...) \tikz [<->] \draw (0,0);</pre>
	<pre>% Degenerate path, tips=proper suppresses arrows \tikz [<->] \draw [tips=proper] (0,0);</pre>
	<pre>% Normal case: \tikz [<->] \draw (0,0) -- (1,0);</pre>
	<pre>% Two subpaths, only second gets tips \tikz [<->] \draw (0,0) -- (1,0) (2,0) -- (3,0);</pre>
	<pre>% Two subpaths, second degenerate, but still gets tips \tikz [<->] \draw (0,0) -- (1,0) (2,0);</pre>
	<pre>% Two subpaths, second degenerate, proper suppresses them \tikz [<->] \draw [tips=on proper draw] (0,0) -- (1,0) (2,0);</pre>
	<pre>% Two subpaths, but one is closed: No tips, even though last subpath is open \tikz [<->] \draw (0,0) circle[radius=2pt] (2,0) -- (3,0);</pre>

One common pitfall when arrow tips are added to a path should be addressed right here at the beginning: When TikZ positions an arrow tip at the start, for all its computations it only takes into account the first segment of the subpath to which the arrow tip is added. This “first segment” is the first line-to or curve-to operation (or arc or parabola or a similar operation) of the path; but note that decorations like `snake` will add many small line segments to paths. The important point is that if this first segment is very small, namely smaller than the arrow tip itself, strange things may result. As will be explained in Section 16.3.8, TikZ will modify the path by shortening the first segment and shortening a segment below its length may result in strange effects. Similarly, for tips at the end of a subpath, only the last segment is considered.

The bottom line is that wherever an arrow tip is added to a path, the line segment where it is added should be “long enough”.

16.3 Arrow Keys: Configuring the Appearance of a Single Arrow Tip

For standard arrow tip kinds, like `Stealth` or `Latex` or `Bar`, you can easily change their size, aspect ratio, color, and other parameters. This is similar to selecting a font face from a font family: “*This text*” is not just typeset in the font “Computer Modern”, but rather in “Computer Modern, italic face, 11pt size, medium weight, black color, no underline, ...”. Similarly, an arrow tip is not just a “`Stealth`” arrow tip, but rather a “`Stealth` arrow tip at its natural size, flexing, but not bending along the path, miter line caps, draw and fill colors identical to the path draw color, ...”

Just as most programs make it easy to “configure” which font should be used at a certain point in a text, TikZ tries to make it easy to specify which configuration of an arrow tip should be used. You use *arrow keys*, where a certain parameter like the `length` of an arrow is set to a given value using the standard key–value syntax. You can provide several arrow keys following an arrow tip kind in an arrow tip specification as in `Stealth[length=4pt, width=2pt]`.

While selecting a font may be easy, *designing* a new font is a highly creative and difficult process and more often than not, not all faces of a font are available on any given system. The difficulties involved in designing a new arrow tip are somewhat similar to designing a new letter for a font and, thus, it may also happen that not all configuration options are actually implemented for a given arrow tip. Naturally, for the

standard arrow tips, all configuration options are available – but for special-purpose arrow tips it may well happen that an arrow tip kind simply “ignores” some of the configurations given by you.

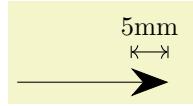
Some of the keys explained in the following are defined in the library `arrows.meta`, others are always available. This has to do with the question of whether the arrow key needs to be supported directly in the PGF core or not. In general, the following explanations assume that `arrows.meta` has been loaded.

16.3.1 Size

The most important configuration parameter of an arrow tip is undoubtedly its size. The following two keys are the main keys that are important in this context:

`/pgf/arrow keys/length=<dimension><line width factor><outer factor>` (no default)

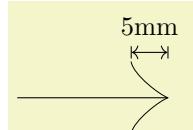
This parameter is usually the most important parameter that governs the size of an arrow tip: The `<dimension>` that you provide dictates the distance from the “very tip” of the arrow to its “back end” along the line:



```
\usetikzlibrary {arrows.meta}
\begin{tikz}
\draw [-{Stealth[length=5mm]}] (0,0) -- (2,0);
\draw [|<-|] (1.5,.4) -- node[above=1mm] {5mm} (2,.4);
\end{tikz}
```



```
\usetikzlibrary {arrows.meta}
\begin{tikz}
\draw [-{Latex[length=5mm]}] (0,0) -- (2,0);
\draw [|<-|] (1.5,.4) -- node[above=1mm] {5mm} (2,.4);
\end{tikz}
```



```
\usetikzlibrary {arrows.meta}
\begin{tikz}
\draw [-{Classical TikZ Rightarrow[length=5mm]}] (0,0) -- (2,0);
\draw [|<-|] (1.5,.6) -- node[above=1mm] {5mm} (2,.6);
\end{tikz}
```

The Line Width Factors. Following the `<dimension>`, you may put a space followed by a `<line width factor>`, which must be a plain number (no `pt` or `cm` following). When you provide such a number, the size of the arrow tip is not just `<dimension>`, but rather $\langle \text{dimension} \rangle + \langle \text{line width factor} \rangle \cdot w$ where w is the width of the to-be-drawn path. This makes it easy to vary the size of an arrow tip in accordance with the line width – usually a very good idea since thicker lines will need thicker arrow tips.

As an example, when you write `length=0pt 5`, the length of the arrow will be exactly five times the current line width. As another example, the default length of a `Latex` arrow is `length=3pt 4.5 0.8`. Let us ignore the `0.8` for a moment; the `4pt 4.5` then means that for the standard line width of `0.4pt`, the length of a `Latex` arrow will be exactly `4.8pt` (`3pt` plus `4.5` times `0.4pt`).

Following the line width factor, you can additionally provide an `<outer factor>`, again preceded by a space (the `0.8` in the above example). This factor is taken into consideration only when the `double` option is used, that is, when a so-called “inner line width”. For a double line, we can identify three different “line widths”, namely the inner line width w_i , the line width w_o of the two outer lines, and the “total line width” $w_t = w_i + 2w_o$. In the below examples, we have $w_i = 3pt$, $w_o = 1pt$, and $w_t = 5pt$. It is not immediately clear which of these line widths should be considered as w in the above formula $\langle \text{dimension} \rangle + \langle \text{line width factor} \rangle \cdot w$ for the computation of the length. One can argue both for w_t and also for w_o . Because of this, you use the `<outer factor>` to decide on one of them or even mix them: TikZ sets $w = \langle \text{outer factor} \rangle w_o + (1 - \langle \text{outer factor} \rangle) w_t$. Thus, when the outer factor is `0`, as in the first of the following examples and as is the default when it is not specified, the computed w will be the total line width $w_t = 5pt$. Since $w = 5pt$, we get a total length of `15pt` in the first example (because of the factor `3`). In contrast, in the last example, the outer factor is `1` and, thus, $w = w_o = 1pt$ and the resulting length is `3pt`. Finally, for the middle case, the “middle” between `5pt` and `1pt` is `3pt`, so the length is `9pt`.



```
\usetikzlibrary {arrows.meta}
\begin{tikz}
\draw [line width=1pt, double distance=3pt,
arrows = {-Latex[length=0pt 3 0]}] (0,0) -- (1,0);
\end{tikz}
```



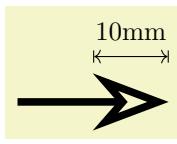
```
\usetikzlibrary {arrows.meta}
\begin{tikzpicture}
\draw [line width=1pt, double distance=3pt,
arrows = {-Latex[length=0pt 3 .5]}] (0,0) -- (1,0);
\end{tikzpicture}
```



```
\usetikzlibrary {arrows.meta}
\begin{tikzpicture}
\draw [line width=1pt, double distance=3pt,
arrows = {-Latex[length=0pt 3 1]}] (0,0) -- (1,0);
\end{tikzpicture}
```

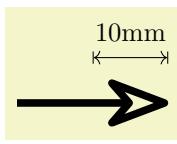
The Exact Length. For an arrow tip kind that is just an outline that is filled with a color, the specified length should *exactly* equal the distance from the tip to the back end. However, when the arrow tip is drawn by stroking a line, it is no longer obvious whether the `length` should refer to the extend of the stroked lines' path or of the resulting pixels (which will be wider because of the thickness of the stroking pen). The rules are as follows:

1. If the arrow tip consists of a closed path (like `Stealth` or `Latex`), imagine the arrow tip drawn from left to right using a miter line cap. Then the `length` should be the horizontal distance from the first drawn “pixel” to the last drawn “pixel”. Thus, the thickness of the stroked line and also the miter ends should be taken into account:



```
\usetikzlibrary {arrows.meta}
\begin{tikzpicture}
\draw [line width=1mm, -{Stealth[length=10mm, open]}]
(0,0) -- (2,0);
\draw [|<-|] (2,.6) -- node[above=1mm] {10mm} ++(-10mm,0);
\end{tikzpicture}
```

2. If, in the above case, the arrow is drawn using a round line join (see Section 16.3.7 for details on how to select this), the size of the arrow should still be the same as in the first case (that is, as if a miter join were used). This creates some “visual consistency” if the two modes are mixed or if you later one change the mode.



```
\usetikzlibrary {arrows.meta}
\begin{tikzpicture}
\draw [line width=1mm, -{Stealth[length=10mm, open, round]}]
(0,0) -- (2,0);
\draw [|<-|] (2,.6) -- node[above=1mm] {10mm} ++(-10mm,0);
\end{tikzpicture}
```

As the above example shows, however, a rounded arrow will still exactly “tip” the point where the line should end (the point $(2,0)$ in the above case). It is only the scaling of the arrow that is not affected.

/pgf/arrow keys/width=*dimension**line width factor**outer factor* (no default)

This key works like the `length` key, only it specifies the “width” of the arrow tip; so if width and length are identical, the arrow will just touch the borders of a square. (An exception to this rule are “halved” arrow tips, see Section 16.3.5.) The meaning of the two optional factor numbers following the *dimension* is the same as for the `length` key.



```
\usetikzlibrary {arrows.meta}
\begin{tikzpicture}
\draw [arrows = {-Latex[width=10pt, length=10pt]}] (0,0) -- (1,0);
\end{tikzpicture}
```



```
\usetikzlibrary {arrows.meta}
\begin{tikzpicture}
\draw [arrows = {-Latex[width=0pt 10, length=10pt]}] (0,0) -- (1,0);
\end{tikzpicture}
```

/pgf/arrow keys/width'=*dimension**length factor**line width factor* (no default)

The key (note the prime) has a similar effect as the `width` key. The difference is that the second, still optional parameter *length factor* specifies the width of the key not as a multiple of the line width, but as a multiple of the arrow length.

The idea is that if you write, say, `width'=0pt 0.5`, the width of the arrow will be half its length. Indeed, for standard arrow tips like `Stealth` the default width is specified in this way so that if you change the length of an arrow tip, you also change the width in such a way that the aspect ratio of the arrow tip is

kept. The other way round, if you modify the factor in `width'` without changing the length, you change the aspect ratio of the arrow tip.

Note that later changes of the length are taken into account for the computation. For instance, if you write

```
length = 10pt, width'=5pt 2, length=7pt
```

the resulting width will be $19pt = 5pt + 2 \cdot 7pt$.



```
\usetikzlibrary {arrows.meta}
\begin{tikzpicture}
\draw [arrows = {-Latex[width'=0pt .5, length=10pt]}] (0,0) -- (1,0);
\end{tikzpicture}
```



```
\usetikzlibrary {arrows.meta}
\begin{tikzpicture}
\draw [arrows = {-Latex[width'=0pt .5, length=15pt]}] (0,0) -- (1,0);
\end{tikzpicture}
```

The third, also optional, parameter allows you to add a multiple of the line width to the value computed in terms of the length.

/pgf/arrow keys/inset=*(dimension)**(line width factor)**(outer factor)* (no default)

The key is relevant only for some arrow tips such as the `Stealth` arrow tip. It specifies a distance by which something inside the arrow tip is set inwards; for the `Stealth` arrow tip it is the distance by which the back angle is moved inwards.

The computation of the distance works in the same way as for `length` and `width`: To the *(dimension)* we add *(line width factor)* times that line width, where the line width is computed based on the *(outer factor)* as described for the `length` key.



```
\usetikzlibrary {arrows.meta}
\begin{tikzpicture}
\draw [arrows = {-Stealth[length=10pt, inset=5pt]}] (0,0) -- (1,0);
\end{tikzpicture}
```



```
\usetikzlibrary {arrows.meta}
\begin{tikzpicture}
\draw [arrows = {-Stealth[length=10pt, inset=2pt]}] (0,0) -- (1,0);
\end{tikzpicture}
```

For most arrows for which there is no “natural inset” like, say, `Latex`, this key has no effect.

/pgf/arrow keys/inset'=*(dimension)**(length factor)**(line width factor)* (no default)

This key works like `inset`, only like `width'` the second parameter is a factor of the arrow length rather than of the line width. For instance, the `Stealth` arrow sets `inset'` to `0pt 0.325` to ensure that the inset is always at $13/40$ th of the arrow length if nothing else is specified.

/pgf/arrow keys/angle=*(angle)*:*(dimension)**(line width factor)**(outer factor)* (no default)

This key sets the `length` and the `width` of an arrow tip at the same time. The length will be the cosine of *(angle)*, while the width will be twice the sine of half the *(angle)* (this slightly awkward rule ensures that a `Stealth` arrow will have an opening angle of *(angle)* at its tip if this option is used). As for the `length` key, if the optional factors are given, they add a certain multiple of the line width to the *(dimension)* before the sine and cosines are computed.



```
\usetikzlibrary {arrows.meta}
\begin{tikzpicture}
\draw [arrows = {-Stealth[inset=0pt, angle=90:10pt]}] (0,0) -- (1,0);
\end{tikzpicture}
```



```
\usetikzlibrary {arrows.meta}
\begin{tikzpicture}
\draw [arrows = {-Stealth[inset=0pt, angle=30:10pt]}] (0,0) -- (1,0);
\end{tikzpicture}
```

/pgf/arrow keys/angle'=*(angle)* (no default)

Sets the width of the arrow to twice the tangent of *(angle)*/ 2 times the arrow length. This results in an arrow tip with an opening angle of *(angle)* at its tip and with the specified `length` unchanged.



```
\usetikzlibrary {arrows.meta}
\begin{tikzpicture}
\draw [arrows = {-Stealth[inset=0pt, length=10pt, angle'=90]}] (0,0) -- (1,0);
\end{tikzpicture}
```



```
\usetikzlibrary {arrows.meta}
\begin{tikzpicture}
\draw [arrows = {-Stealth[inset=0pt, length=10pt, angle'=30]}]
(0,0) -- (1,0);
\end{tikzpicture}
```

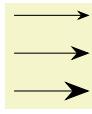
16.3.2 Scaling

In the previous section we saw that there are many options for getting “fine control” over the length and width of arrow tips. However, in some cases, you do not really care whether the arrow tip is 4pt long or 4.2pt long, you “just want it to be a little bit larger than usual”. In such cases, the following keys are useful:

`/pgf/arrows keys/scale=<factor>`

(no default, initially 1)

After all the other options listed in the previous (and also the following sections) have been processed, TikZ applies a *scaling* to the computed length, inset, and width of the arrow tip (and, possibly, to other size parameters defined by special-purpose arrow tip kinds). Everything is simply scaled by the given `<factor>`.



```
\usetikzlibrary {arrows.meta}
\begin{tikzpicture}
\draw [arrows = {-Stealth[]}] (0,1) -- (1,1);
\draw [arrows = {-Stealth[scale=1.5]}] (0,0.5) -- (1,0.5);
\draw [arrows = {-Stealth[scale=2]}] (0,0) -- (1,0);
\end{tikzpicture}
```

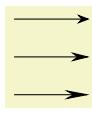
Note that scaling has *no* effect on the line width (as usual) and also not on the arrow padding (the `sep`).

You can get even more fine-grained control over scaling using the following keys (the `scale` key is just a shorthand for setting both of the following keys simultaneously):

`/pgf/arrows keys/scale length=<factor>`

(no default, initially 1)

This factor works like `scale`, only it is applied only to dimensions “along the axis of the arrow”, that is, to the length and to the inset, but not to the width.

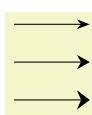


```
\usetikzlibrary {arrows.meta}
\begin{tikzpicture}
\draw [arrows = {-Stealth[]}] (0,1) -- (1,1);
\draw [arrows = {-Stealth[scale length=1.5]}] (0,0.5) -- (1,0.5);
\draw [arrows = {-Stealth[scale length=2]}] (0,0) -- (1,0);
\end{tikzpicture}
```

`/pgf/arrows keys/scale width=<factor>`

(no default, initially 1)

Like `scale length`, but for dimensions related to the width.



```
\usetikzlibrary {arrows.meta}
\begin{tikzpicture}
\draw [arrows = {-Stealth[]}] (0,1) -- (1,1);
\draw [arrows = {-Stealth[scale width=1.5]}] (0,0.5) -- (1,0.5);
\draw [arrows = {-Stealth[scale width=2]}] (0,0) -- (1,0);
\end{tikzpicture}
```

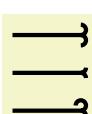
16.3.3 Arc Angles

A few arrow tips consist mainly of arcs, whose length can be specified. For these arrow tips, you use the following key:

`/pgf/arrows keys/arc=<degrees>`

(no default, initially 180)

Sets the angle of arcs in arrows to `<degrees>`. Note that this key is quite different from the `angle` key, which is “just a fancy way of setting the length and width”. In contrast, the `arc` key is used to set the degrees of arcs that are part of an arrow tip:



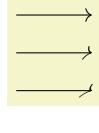
```
\usetikzlibrary {arrows.meta}
\begin{tikzpicture}
\draw [ultra thick] [
arrows = {-Hooks[]}] (0,1) -- (1,1);
\draw [ultra thick] [
arrows = {-Hooks[arc=90]}] (0,0.5) -- (1,0.5);
\draw [ultra thick] [
arrows = {-Hooks[arc=270]}] (0,0) -- (1,0);
\end{tikzpicture}
```

16.3.4 Slanting

You can “slant” arrow tips using the following key:

`/pgf/arrow keys/slant=<factor>` (no default, initially 0)

Slanting is used to create an “italics” effect for arrow tips: All arrow tips get “slanted” a little bit relative to the axis of the arrow:



```
\usetikzlibrary {arrows.meta}
\begin{tikzpicture}
\draw [arrows = {->}] (0,1) -- (1,1);
\draw [arrows = {->[slant=.5]}] (0,0.5) -- (1,0.5);
\draw [arrows = {->[slant=1]}] (0,0) -- (1,0);
\end{tikzpicture}
```

There is one thing to note about slanting: Slanting is done using a so-called “canvas transformation” and has no effect on positioning of the arrow tip. In particular, if an arrow tip gets slanted so strongly that it starts to protrude over the arrow tip end, this does not change the positioning of the arrow tip. Here is another example where slanting is used to match italic text:

$A \twoheadrightarrow B \twoheadleftarrow C$

```
\usetikzlibrary {arrows.meta,graphs}
\begin{tikzpicture}[>={slant=.3 To[] To[]}]
\graph [math nodes] { A -> B <-> C };
\end{tikzpicture}
```

16.3.5 Reversing, Halving, Swapping

`/pgf/arrow keys/reversed` (no value)

Adding this key to an arrow tip will “reverse its direction” so that is points in the opposite direction (but is still at that end of the line where the non-reversed arrow tip would have been drawn; so only the tip is reversed). For most arrow tips, this just results in an internal flip of a coordinate system, but some arrow tips actually use a slightly different version of the tip for reversed arrow tips (namely when the joining of the tip with the line would look strange). All of this happens automatically, so you do not need to worry about this.

If you apply this key twice, the effect cancels, which is useful for the definition of shorthands (which will be discussed later).



```
\usetikzlibrary {arrows.meta}
\begin{tikzpicture}[ultra thick]
\draw [arrows = {-Stealth[reversed]}] (0,0) -- (1,0);
```



```
\usetikzlibrary {arrows.meta}
\begin{tikzpicture}[ultra thick]
\draw [arrows = {-Stealth[reversed, reversed]}] (0,0) -- (1,0);
```

`/pgf/arrow keys/harpoon`

(no value)

The key requests that only the “left half” of the arrow tip should drawn:



```
\usetikzlibrary {arrows.meta}
\begin{tikzpicture}[ultra thick]
\draw [arrows = {-Stealth[harpoon]}] (0,0) -- (1,0);
```



```
\usetikzlibrary {arrows.meta}
\begin{tikzpicture}[ultra thick]
\draw [arrows = {->[harpoon]}] (0,0) -- (1,0);
```

Unlike the `reversed` key, which all arrows tip kinds support at least in a basic way, designers of arrow tips really need to take this key into account in their arrow tip code and often a lot of special attention needs to be paid to this key in the implementation. For this reason, only some arrow tips will support it.

`/pgf/arrow keys/swap`

(no value)

This key flips that arrow tip along the axis of the line. It makes sense only for asymmetric arrow tips like the harpoons created using the `harpoon` option.



```
\usetikzlibrary {arrows.meta}
\tikz [ultra thick] \draw [arrows = {-Stealth[harpoon]}] (0,0) -- (1,0);
```



```
\usetikzlibrary {arrows.meta}
\tikz [ultra thick] \draw [arrows = {-Stealth[harpoon,swap]}] (0,0) -- (1,0);
```

Swapping is always possible, no special code is needed on behalf of an arrow tip implementer.

`/pgf/arrow keys/left`

(no value)

A shorthand for `harpoon`.

`/pgf/arrow keys/right`

(no value)

A shorthand for `harpoon, swap`.



```
\usetikzlibrary {arrows.meta}
\tikz [ultra thick] \draw [arrows = {-Stealth[left]}] (0,0) -- (1,0);
```



```
\usetikzlibrary {arrows.meta}
\tikz [ultra thick] \draw [arrows = {-Stealth[right]}] (0,0) -- (1,0);
```

16.3.6 Coloring

Arrow tips are drawn using the same basic mechanisms as normal paths, so arrow tips can be stroked (drawn) and/or filled. However, we usually want the color of arrow tips to be identical to the color used to draw the path, even if a different color is used for filling the path. On the other hand, we may also sometimes wish to use a special color for the arrow tips that is different from both the line and fill colors of the main path.

The following options allow you to configure how arrow tips are colored:

`/pgf/arrow keys/color=<color or empty>`

(no default, initially empty)

Normally, an arrow tip gets the same color as the path to which it is attached. More precisely, it will get the current “draw color”, also known as “stroke color”, which you can set using `draw=<some color>`. By adding the option `color=` to an arrow tip (note that an “empty” color is specified in this way), you ask that the arrow tip gets this default draw color of the path. Since this is the default behaviour, you usually do not need to specify anything:



```
\usetikzlibrary {arrows.meta}
\tikz [ultra thick] \draw [red, arrows = {-Stealth}] (0,0) -- (1,0);
```



```
\usetikzlibrary {arrows.meta}
\tikz [ultra thick] \draw [blue, arrows = {-Stealth}] (0,0) -- (1,0);
```

Now, when you provide a `<color>` with this option, you request that the arrow tip should get this color instead of the color of the main path:



```
\usetikzlibrary {arrows.meta}
\tikz [ultra thick] \draw [red, arrows = {-Stealth[color=blue]}] (0,0) -- (1,0);
```



```
\usetikzlibrary {arrows.meta}
\tikz [ultra thick] \draw [red, arrows = {-Stealth[color=black]}] (0,0) -- (1,0);
```

Similar to the `color` option used in normal TikZ options, you may omit the `color=` part of the option. Whenever an `<arrow key>` is encountered that TikZ does not recognize, it will test whether the key is the name of a color and, if so, execute `color=<arrow key>`. So, the first of the above examples can be rewritten as follows:



```
\usetikzlibrary {arrows.meta}
\tikz [ultra thick] \draw [red, arrows = {-Stealth[blue]}] (0,0) -- (1,0);
```

The `<color>` will apply both to any drawing and filling operations used to construct the path. For instance, even though the `Stealth` arrow tips looks like a filled quadrilateral, it is actually constructed by drawing a quadrilateral and then filling it in the same color as the drawing (see the `fill` option below to see the difference).

When `color` is set to an empty text, the drawing color is always used to fill the arrow tips, even if a different color is specified for filling the path:



```
\usetikzlibrary {arrows.meta}
\tikz [ultra thick] \draw [draw=red, fill=red!50, arrows = {-Stealth[length=10pt]}]
(0,0) -- (1,1) -- (2,0);
```

As you can see in the above example, the filled area is not quite what you might have expected. The reason is that the path was actually internally shortened a bit so that the end of the “fat line” as inside the arrow tip and we get a “clear” arrow tip.

In general, it is a good idea not to add arrow tips to paths that are filled.

/pgf/arrow keys/fill=<color or none> (no default)

Use this key to explicitly set the color used for filling the arrow tips. This color can be different from the color used to draw (stroke) the arrow tip:



```
\usetikzlibrary {arrows.meta}
\tikz {
\draw [help lines] (0,-.5) grid [step=1mm] (1,.5);
\draw [thick, red, arrows = {-Stealth[fill=white,length=15pt]}] (0,0) -- (1,0);
}
```

You can also specify the special “color” `none`. In this case, the arrow tip is not filled at all (not even with white):



```
\usetikzlibrary {arrows.meta}
\tikz {
\draw [help lines] (0,-.5) grid [step=1mm] (1,.5);
\draw [thick, red, arrows = {-Stealth[fill=none,length=15pt]}] (0,0) -- (1,0);
}
```

Note that such “open” arrow tips are a bit difficult to draw in some case: The problem is that the line must be shortened by just the right amount so that it ends exactly on the back end of the arrow tip. In some cases, especially when double lines are used, this will not be possible.

/pgf/arrow keys/open (no value)

A shorthand for `fill=none`.

When you use both the `color` and `fill` option, the `color` option must come first since it will reset the filling to the color specified for drawing.



```
\usetikzlibrary {arrows.meta}
\tikz {
\draw [help lines] (0,-.5) grid [step=1mm] (1,.5);
\draw [thick, red, arrows = {-Stealth[color=blue, fill=white, length=15pt]}]
(0,0) -- (1,0);
}
```

Note that by setting `fill` to the special color `pgffillcolor`, you can cause the arrow tips to be filled using the color used to fill the main path. (This special color is always available and always set to the current filling color of the graphic state.):



```
\usetikzlibrary {arrows.meta}
\tikz [ultra thick] \draw [draw=red, fill=red!50,
arrows = {-Stealth[length=15pt, fill=pgffillcolor]}]
(0,0) -- (1,1) -- (2,0);
```

16.3.7 Line Styling

Arrow tips are created by drawing and possibly filling a path that makes up the arrow tip. When TikZ draws a path, there are different ways in which such a path can be drawn (such as dashing). Three particularly important parameters are the line join, the line cap, see Section 15.3.1 for an introduction, and the line width (thickness).

TikZ resets the line cap and line join each time it draws an arrow tip since you usually do not want their settings to “spill over” to the way the arrow tips are drawn. You can, however, change these values explicitly for an arrow tip:

`/pgf/arrow keys/line cap=<round or butt>` (no default)

Sets the line cap of all lines that are drawn in the arrow to a round cap or a butt cap. (Unlike for normal lines, the rect cap is not allowed.) Naturally, this key has no effect for arrows whose paths are closed.

Each arrow tip has a default value for the line cap, which can be overruled using this option.

Changing the cap should have no effect on the size of the arrow. However, it will have an effect on where the exact “tip” of the arrow is since this will always be exactly at the end of the arrow:



```
\usetikzlibrary {arrows.meta}
\begin{tikzpicture} [line width=2mm]
\draw [arrows = {-Computer Modern Rightarrow[line cap=butt]}]
(0,0) -- (1,0);

```



```
\usetikzlibrary {arrows.meta}
\begin{tikzpicture} [line width=2mm]
\draw [arrows = {-Computer Modern Rightarrow[line cap=round]}]
(0,0) -- (1,0);

```



```
\usetikzlibrary {arrows.meta}
\begin{tikzpicture} [line width=2mm]
\draw [arrows = {-Bracket[reversed,line cap=butt]}]
(0,0) -- (1,0);

```



```
\usetikzlibrary {arrows.meta}
\begin{tikzpicture} [line width=2mm]
\draw [arrows = {-Bracket[reversed,line cap=round]}]
(0,0) -- (1,0);

```

`/pgf/arrow keys/line join=<round or miter>` (no default)

Sets the line join to round or miter (bevel is not allowed). This time, the key only has an effect on paths that have “corners” in them. The same rules as for line cap apply: the size is not affected, but the tip end is:



```
\usetikzlibrary {arrows.meta}
\begin{tikzpicture} [line width=2mm]
\draw [arrows = {-Computer Modern Rightarrow[line join=miter]}]
(0,0) -- (1,0);

```



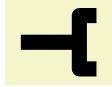
```
\usetikzlibrary {arrows.meta}
\begin{tikzpicture} [line width=2mm]
\draw [arrows = {-Computer Modern Rightarrow[line join=round]}]
(0,0) -- (1,0);

```



```
\usetikzlibrary {arrows.meta}
\begin{tikzpicture} [line width=2mm]
\draw [arrows = {-Bracket[reversed,line join=miter]}]
(0,0) -- (1,0);

```



```
\usetikzlibrary {arrows.meta}
\begin{tikz} [line width=2mm]
\draw [arrows = {-Bracket[reversed, line join=round]}] (0,0) -- (1,0);
```

The following keys set both of the above:

/pgf/arrow keys/round

(no value)

A shorthand for `line cap=round`, `line join=round`, resulting in “rounded” arrow heads.



```
\usetikzlibrary {arrows.meta}
\begin{tikz} [line width=2mm]
\draw [arrows = {-Computer Modern Rightarrow[round]}] (0,0) -- (1,0);
```



```
\usetikzlibrary {arrows.meta}
\begin{tikz} [line width=2mm]
\draw [arrows = {-Bracket[reversed, round]}] (0,0) -- (1,0);
```

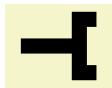
/pgf/arrow keys/sharp

(no value)

A shorthand for `line cap=butt`, `line join=miter`, resulting in “sharp” or “pointed” arrow heads.



```
\usetikzlibrary {arrows.meta}
\begin{tikz} [line width=2mm]
\draw [arrows = {-Computer Modern Rightarrow[sharp]}] (0,0) -- (1,0);
```



```
\usetikzlibrary {arrows.meta}
\begin{tikz} [line width=2mm]
\draw [arrows = {-Bracket[reversed, sharp]}] (0,0) -- (1,0);
```

You can also set the width of lines used inside arrow tips:

/pgf/arrow keys/line width=<dimension><line width factor><outer factor>

(no default)

This key sets the line width inside an arrow tip for drawing (out)lines of the arrow tip. When you set this width to `0pt`, which makes sense only for closed tips, the arrow tip is only filled. This can result in better rendering of some small arrow tips and in case of bend arrow tips (because the line joins will also be bend and not “mitered”).

The meaning of the factors is as usual the same as for `length` or `width`.



```
\usetikzlibrary {arrows.meta}
\begin{tikz} \draw [arrows = {-Latex[line width=0.1pt, fill=white, length=10pt]}] (0,0) -- (1,0);
```



```
\usetikzlibrary {arrows.meta}
\begin{tikz} \draw [arrows = {-Latex[line width=1pt, fill=white, length=10pt]}] (0,0) -- (1,0);
```

/pgf/arrow keys/line width'=<dimension><length factor>

(no default)

Works like `line width` only the factor is with respect to the `length`.

16.3.8 Bending and Flexing

Up to now, we have only added arrow tip to the end of straight lines, which is in some sense “easy”. Things get far more difficult, if the line to which we wish to end an arrow tip is curved. In the following, we have a look at the different actions that can be taken and how they can be configured.

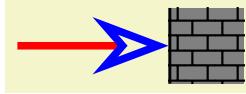
To get a feeling for the difficulties involved, consider the following situation: We have a “gray wall” at the x -coordinate of and a red line that ends in its middle.



```
\usetikzlibrary {patterns}
\def\wall{ \fill      [fill=black!50]  (1,-.5) rectangle (2,.5);
            \pattern [pattern=bricks] (1,-.5) rectangle (2,.5);
            \draw      [line width=1pt]  (1cm+.5pt,-.5) -- ++(0,1); }

\begin{tikzpicture}
    \wall
    % The "line"
    \draw [red,line width=1mm] (-1,0) -- (1,0);
\end{tikzpicture}
```

Now we wish to add a blue open arrow tip the red line like, say, `Stealth[length=1cm,open,blue]`:



```
\usetikzlibrary {arrows.meta}
\begin{tikzpicture}
    \wall
    \draw [red,line width=1mm,-{Stealth[length=1cm,open,blue]}]
        (-1,0) -- (1,0);
\end{tikzpicture}
```

There are several noteworthy things about the blue arrow tip:

1. Notice that the red line no longer goes all the way to the wall. Indeed, the red line ends more or less exactly where it meets the blue line, leaving the arrow tip empty. Now, recall that the red line was supposed to be the path `(-2,0)--(1,0)`; however, this path has obviously become much shorter (by 6.25mm to be precise). This effect is called *path shortening* in TikZ.
2. The very tip of the arrow just “touches” the wall, even we zoom out a lot. This point, where the original path ended and where the arrow tip should now lie, is called the *tip end* in TikZ.
3. Finally, the point where the red line touches the blue line is the point where the original path “visually ends”. Notice that this is not the same as the point that lies at a distance of the arrow’s `length` from the wall – rather it lies at a distance of `length` minus the `inset`. Let us call this point the *visual end* of the arrow.

As pointed out earlier, for straight lines, shortening the path and rotating and shifting the arrow tip so that it ends precisely at the tip end and the visual end lies on a line from the tip end to the start of the line is relatively easy.

For curved lines, things are much more difficult and TikZ copes with the difficulties in different ways, depending on which options you add to arrows. Here is now a curved red line to which we wish to add our arrow tip (the original straight red line is shown in light red):



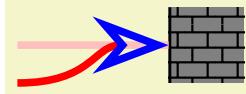
```
\begin{tikzpicture}
    \wall
    \draw [red!25,line width=1mm] (-1,0) -- (1,0);
    \draw [red,line width=1mm] (-1,-.5) .. controls (0,-.5) and (0,0) .. (1,0);
\end{tikzpicture}
```

The first way of dealing with curved lines is dubbed the “quick and dirty” way (although the option for selecting this option is politely just called “`quick`” . . .):

`/pgf/arrow keys/quick` (no value)

Recall that curves in TikZ are actually Bézier curves, which means that they start and end at certain points and we specify two vectors, one for the start and one for the end, that provide tangents to the curve at these points. In particular, for the end of the curve, there is a point called the *second support point* of the curve such that a tangent to the curve at the end goes through this point. In our above example, the second support point is at the middle of the light red line and, indeed, a tangent to the red line at the point touching the wall is perfectly horizontal.

In order to add our arrow tip to the curved path, our first objective is to “shorten” the path by 6.25mm. Unfortunately, this is now much more difficult than for a straight path. When the `quick` option is added to an arrow tip (it is also the default if no special libraries are loaded), we cheat somewhat: Instead of really moving along 6.25mm along the path, we simply shift the end of the curve by 6.25mm *along the tangent* (which is easy to compute). We also have to shift the second support point by the same amount to ensure that the line still has the same tangent at the end. This will result in the following:



```
\usetikzlibrary {arrows.meta}
\begin{tikzpicture}
\wall
\draw [red!25,line width=1mm] (-1,0) -- (1,0);
\draw [red,line width=1mm,-{Stealth[length=1cm,open,blue,quick]}]
(-1,-.5) .. controls (0,-.5) and (0,0) .. (1,0);
\end{tikzpicture}
```

The main problem with the above picture is that the red line is no longer equal to the original red line (notice much sharper curvature near its end). In our example this is not such a bad thing, but it certainly “not a nice thing” that adding arrow tips to a curve changes the overall shape of the curves. This is especially bothersome if there are several similar curves that have different arrow heads. In this case, the similar curves now suddenly look different.

Another big problem with the above approach is that it works only well if there is only a single arrow tip. When there are multiple ones, simply shifting them along the tangent as the `quick` option does produces less-than-satisfactory results:



```
\begin{tikzpicture}
\wall
\draw [red!25,line width=1mm] (-1,0) -- (1,0);
\draw [red,line width=1mm,-{[quick,sep]>>}]
(-1,-.5) .. controls (0,-.5) and (0,0) .. (1,0);
\end{tikzpicture}
```

Note that the third arrow tip does not really lie on the curve any more.

Because of the shortcomings of the `quick` key, more powerful mechanisms for shortening lines and rotating arrows tips have been implemented. To use them, you need to load the following library:

TikZ Library `bending`

```
\usetikzlibrary{bending} % LATEX and plain TEX
\usetikzlibrary[bending] % ConTEX
```

Load this library to use the `flex`, `flex'`, or `bending` arrow keys. When this library is loaded, `flex` becomes the default mode that is used with all paths, unless `quick` is explicitly selected for the arrow tip.

`/pgf/arrow keys/flex=<factor>` (default 1)

When the `bending` library is loaded, this key is applied to all arrow tips by default. It has the following effect:

1. Instead of simply shifting the visual end of the arrow along the tangent of the curve’s end, we really move it along the curve by the necessary distance. This operation is more expensive than the `quick` operation – but not *that* expensive, only expensive enough so that it is not selected by default for all arrow tips. Indeed, some compromises are made in the implementation where accuracy was traded for speed, so the distance by which the line end is shifted is not necessarily *exactly* 6.25mm; only something reasonably close.
2. The supports of the line are updated accordingly so that the shortened line will still follow *exactly* the original line. This means that the curve deformation effect caused by the `quick` command does not happen here.
3. Next, the arrow tip is rotated and shifted as follows: First, we shift it so that its tip is exactly at the tip end, where the original line ended. Then, the arrow is rotated so the *the visual end lies on the line*:

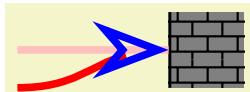


```
\usetikzlibrary {arrows.meta,bending}
\begin{tikzpicture}
\wall
\draw [red!25,line width=1mm] (-1,0) -- (1,0);
\draw [red,line width=1mm,-{Stealth[length=1cm,open,blue,flex]}]
(-1,-.5) .. controls (0,-.5) and (0,0) .. (1,0);
\end{tikzpicture}
```

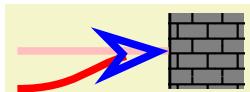
As can be seen in the example, the `flex` option gives a result that is visually pleasing and does not deform the path.

There is, however, one possible problem with the `flex` option: The arrow tip no longer points along the tangent of the end of the path. This may or may not be a problem, put especially for larger arrow tips readers will use the orientation of the arrow head to gauge the direction of the tangent of the line. If this tangent is important (for example, if it should be horizontal), then it may be necessary to enforce that the arrow tip “really points in the direction of the tangent”.

To achieve this, the `flex` option takes an optional `<factor>` parameter, which defaults to 1. This factor specifies how much the arrow tip should be rotated: If set to 0, the arrow points exactly along a tangent to curve at its tip. If set to 1, the arrow point exactly along a line from the visual end point on the curve to the tip. For values in the middle, we interpolate the rotation between these two extremes; so `flex=.5` will rotate the arrow’s visual end “halfway away from the tangent towards the actual position on the line”.



```
\usetikzlibrary {arrows.meta,bending}
\begin{tikzpicture}
  \wall
  \draw [red!25,line width=1mm] (-1,0) -- (1,0);
  \draw [red,line width=1mm,-{Stealth[length=1cm,open,blue,flex=0]}]
    (-1,-.5) .. controls (0,-.5) and (0,0) .. (1,0);
\end{tikzpicture}
```



```
\usetikzlibrary {arrows.meta,bending}
\begin{tikzpicture}
  \wall
  \draw [red!25,line width=1mm] (-1,0) -- (1,0);
  \draw [red,line width=1mm,-{Stealth[length=1cm,open,blue,flex=.5]}]
    (-1,-.5) .. controls (0,-.5) and (0,0) .. (1,0);
\end{tikzpicture}
```

Note how in the above examples the red line is visible inside the open arrow tip. Open arrow tips do not go well with a flex value other than 1. Here is a more realistic use of the `flex=0` key:



```
\usetikzlibrary {arrows.meta,bending}
\begin{tikzpicture}
  \wall
  \draw [red!25,line width=1mm] (-1,0) -- (1,0);
  \draw [red,line width=1mm,-{Stealth[length=1cm,flex=0]}]
    (-1,-.5) .. controls (0,-.5) and (0,0) .. (1,0);
\end{tikzpicture}
```

If there are several arrow tips on a path, the `flex` option positions them independently, so that each of them lies optimally on the path:



```
\usetikzlibrary {bending}
\begin{tikzpicture}
  \wall
  \draw [red!25,line width=1mm] (-1,0) -- (1,0);
  \draw [red,line width=1mm,-{[flex,sep]>>}]
    (-1,-.5) .. controls (0,-.5) and (0,0) .. (1,0);
\end{tikzpicture}
```

`/pgf/arrow keys/flex'=<factor>` (default 1)

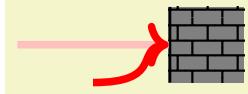
The `flex'` key is almost identical to the `flex` key. The only difference is that a factor of 1 corresponds to rotating the arrow tip so that the instead of the visual end, the “ultimate back end” of the arrow tip lies on the red path. In the example instead of having the arrow tip at a distance of 6.25mm from the tip lie on the path, we have the point at a distance of 1cm from the tip lie on the path:



```
\usetikzlibrary {arrows.meta,bending}
\begin{tikzpicture}
  \wall
  \draw [red!25,line width=1mm] (-1,0) -- (1,0);
  \draw [red,line width=1mm,-{Stealth[length=1cm,open,blue,flex']}]
    (-1,-.5) .. controls (0,-.5) and (0,0) .. (1,0);
\end{tikzpicture}
```

Otherwise, the factor works as for `flex` and, indeed `flex=0` and `flex'=0` have the same effect.

The main use of this option is not so much with an arrow tip like `Stealth` but rather with tips like the standard `>` in the context of a strongly curved line:



```
\usetikzlibrary {arrows.meta,bending}
\begin{tikzpicture}
  \wall
  \draw [red!25,line width=1mm] (-1,0) -- (1,0);
  \draw [red,line width=1mm,-{Computer Modern Rightarrow[flex]}]
    (0,-.5) .. controls (1,-.5) and (0.5,0) .. (1,0);
\end{tikzpicture}
```

In the example, the `flex` option does not really flex the arrow since for a tip like the Computer Modern arrow, the visual end is the same as the arrow tip – after all, the red line does, indeed, end almost exactly where it used to end.

Nevertheless, you may feel that the arrow tip looks “wrong” in the sense that it should be rotated. This is exactly what the `flex'` option does since it allows us to align the “back end” of the tip with the red line:



```
\usetikzlibrary {arrows.meta,bending}
\begin{tikzpicture}
  \wall
  \draw [red!25,line width=1mm] (-1,0) -- (1,0);
  \draw [red,line width=1mm,-{Computer Modern Rightarrow[flex'=.75]}]
    (0,-.5) .. controls (1,-.5) and (0.5,0) .. (1,0);
\end{tikzpicture}
```

In the example, I used `flex'=.75` so as not to overpronounce the effect. Usually, you will have to fiddle with it sometime to get the “perfectly aligned arrow tip”, but a value of `.75` is usually a good start.

/pgf/arrow keys/bend

(no value)

Bending an arrow tip is a radical solution to the problem of positioning arrow tips on a curved line: The arrow tip is no longer “rigid” but the drawing itself will now bend along the curve. This has the advantage that all the problems of flexing with wrong tangents and overflexing disappear. The downsides are longer computation times (bending an arrow is *much* more expensive than flexing it, let alone than quick mode) and also the fact that excessive bending can lead to ugly arrow tips. On the other hand, for most arrow tips their bend version are visually quite pleasing and create a sophisticated look:



```
\usetikzlibrary {arrows.meta,bending}
\begin{tikzpicture}
  \wall
  \draw [red!25,line width=1mm] (-1,0) -- (1,0);
  \draw [red,line width=1mm,-{Stealth[length=20pt,bend]}]
    (-1,-.5) .. controls (0,-.5) and (0,0) .. (1,0);
\end{tikzpicture}
```



```
\usetikzlibrary {bending}
\begin{tikzpicture}
  \wall
  \draw [red!25,line width=1mm] (-1,0) -- (1,0);
  \draw [red,line width=1mm,-{[bend,sep]>>}]
    (-1,-.5) .. controls (0,-.5) and (0,0) .. (1,0);
\end{tikzpicture}
```



```
\usetikzlibrary {arrows.meta,bending}
\begin{tikzpicture}
  \wall
  \draw [red!25,line width=1mm] (-1,0) -- (1,0);
  \draw [red,line width=1mm,-{Stealth[bend,round,length=20pt]}]
    (0,-.5) .. controls (1,-.5) and (0.25,0) .. (1,0);
\end{tikzpicture}
```

16.4 Arrow Tip Specifications

16.4.1 Syntax

When you select the arrow tips for the start and the end of a path, you can specify a whole sequence of arrow tips, each having its own local options. At the beginning of this section, it was pointed out that the syntax for selecting the start and end arrow tips is the following:

$\langle start\ specification \rangle - \langle end\ specification \rangle$

We now have a closer look at what these specifications may look like. The general syntax of the $\langle start\ specification \rangle$ is as follows:

$[\langle options\ for\ all\ tips \rangle] \langle first\ arrow\ tip\ spec \rangle \langle second\ arrow\ tip\ spec \rangle \langle third\ arrow\ tip\ spec \rangle \dots$

As can be seen, an arrow tip specification may start with some options in brackets. If this is the case, the $\langle options\ for\ all\ tips \rangle$ will, indeed, be applied to all arrow tips that follow. (We will see, in a moment, that there are even more places where options may be specified and a list of the ordering in which the options are applied will be given later.)

The main part of a specification is taken up by a sequence of individual arrow tip specifications. Such a specification can be of three kinds:

1. It can be of the form $\langle arrow\ tip\ kind\ name \rangle [\langle options \rangle]$.
2. It can be of the form $\langle shorthand \rangle [\langle options \rangle]$.
3. It can be of the form $\langle single\ char\ shorthand \rangle [\langle options \rangle]$. Note that only for this form the brackets are optional.

The easiest kind is the first one: This adds an arrow tip of the kind $\langle arrow\ tip\ kind\ name \rangle$ to the sequence of arrow tips with the $\langle options \rangle$ applied to it at the start (for the $\langle start\ specification \rangle$) or at the end (for the $\langle end\ specification \rangle$). Note that for the $\langle start\ specification \rangle$ the first arrow tip specified in this way will be at the very start of the curve, while for the $\langle end\ specification \rangle$ the ordering is reversed: The last arrow tip specified will be at the very end of the curve. This implies that a specification like

`Stealth[] Latex[] - Latex[] Stealth[]`

will give perfectly symmetric arrow tips on a line (as one would expect).

It is important that even if there are no $\langle options \rangle$ for an arrow tip, the square brackets still need to be written to indicate the end of the arrow tip's name. Indeed, the opening brackets are used to divide the arrow tip specification into names.

Instead of a $\langle arrow\ tip\ kind\ name \rangle$, you may also provide the name of a so-called *shorthand*. Shorthands look like normal arrow tip kind names and, indeed, you will often be using shorthands without noticing that you do. The idea is that instead of, say, `Computer Modern Rightarrow` you might wish to just write `Rightarrow` or perhaps just `To` or even just `>`. For this, you can create a shorthand that tells TikZ that whenever this shorthand is used, another arrow tip kind is meant. (Actually, shorthands are somewhat more powerful, we have a detailed look at them in Section 16.4.4.) For shorthands, the same rules apply as for normal arrow tip kinds: You *need* to provide brackets so that TikZ can find the end of the name inside a longer specification.

The third kind of arrow tip specifications consist of just a single letter like `>` or `)` or `*` or even `o` or `x` (but you may not use `[`, `]`, or `-` since they will confuse the parser). These single letter arrow specifications will invariably be shorthands that select some “real” arrow tip instead. An important feature of single letter arrow tips is that they do *not* need to be followed by options (but they may).

Now, since we can use any letter for single letter shorthands, how can TikZ tell whether by `foo[]` we mean an arrow tip kind `foo` without any options or whether we mean an arrow tip called `f`, followed by two arrow tips called `oo`? Or perhaps an arrow tip called `f` followed by an arrow tip called `oo`? To solve this problem, the following rule is used to determine which of the three possible specifications listed above applies: First, we check whether everything from the current position up to the next opening bracket (or up to the end) is the name of an arrow tip or of a shorthand. In our case, `foo` would first be tested under this rule. Only if `foo` is neither the name of an arrow tip kind nor of a shorthand does TikZ consider the first letter of the specification, `f` in our case. If this is not the name of a shorthand, an error is raised. Otherwise the arrow tip corresponding to `f` is added to the list of arrow tips and the process restarts with the rest.

Thus, we would next text whether `oo` is the name of an arrow tip or shorthand and, if not, whether `o` is such a name.

All of the above rules mean that you can rather easily specify arrow tip sequences if they either mostly consist of single letter names or of longer names. Here are some examples:

- `-»»` is interpreted as three times the `>` shorthand since `»»` is not the name of any arrow tip kind (and neither is `»`).
- `->[]»` has the same effect as the above.
- `-[]»»` also has the same effect.
- `->[]>[]>[]` so does this.
- `->Stealth` yields an arrow tip `>` followed by a `Stealth` arrow at the end.
- `-Stealth>` is illegal since there is no arrow tip `Stealth>` and since `S` is also not the name of any arrow tip.
- `-Stealth[] >` is legal and does what was presumably meant in the previous item.
- `< Stealth-` is legal and is the counterpart to `-Stealth[] >`.
- `-Stealth[length=5pt] Stealth[length=6pt]` `Stealth[length=6pt]` selects two stealth arrow tips, but at slightly different sizes for the end of lines.

An interesting question concerns how flexing and bending interact with multiple arrow tips: After all, flexing and quick mode use different ways of shortening the path so we cannot really mix them. The following rule is used: We check, independently for the start and the end specifications, whether at least one arrow tip in them uses one of the options `flex`, `flex'`, or `bend`. If so, all `quick` settings in the other arrow tips are ignored and treated as if `flex` had been selected for them, too.

16.4.2 Specifying Paddings

When you provide several arrow tips in a row, all of them are added to the start or end of the line:

```
««————→»» \tikz \draw [««->»»] (0,0) -- (2,0);
```

The question now is what will be the distance between them? For this, the following arrow key is important:

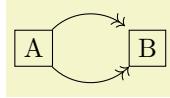
`/pgf/arrow keys/sep=<dimension><line width factor><outer factor>` (default `0.88pt .3 1`)

When a sequence of arrow tips is specified in an arrow tip specification for the end of the line, the arrow tips are normally arranged in such a way that the tip of each arrow ends exactly at the “back end” of the next arrow tip (for start specifications, the ordering is inverted, of course). Now, when the `sep` option is set, instead of exactly touching the back end of the next arrow, the specified `<dimension>` is added as additional space (the distance may also be negative, resulting in an overlap of the arrow tips). The optional factors have the same meaning as for the `length` key, see that key for details.

Let us now have a look at some examples. First, we use two arrow tips with different separations between them:

```
————→»» \usetikzlibrary {arrows.meta}
\tikz {
  \draw [-{>[sep=1pt]>[sep= 2pt]>}] (0,1.0) -- (1,1.0);
  \draw [-{>[sep=1pt]>[sep=-2pt]>}] (0,0.5) -- (1,0.5);
  \draw [-{>[sep] >[sep]}] (0,0.0) -- (1,0.0);
}
```

You can also specify a `sep` for the last arrow tip in the sequence (for end specifications, otherwise for the first arrow tip). In this case, this first arrow tip will not exactly “touch” the point where the path ends, but will rather leave the specified amount of space. This is usually quite desirable.



```
\usetikzlibrary {arrows.meta,positioning}
\begin{tikzpicture}
    \node [draw] (A) {A};
    \node [draw] (B) [right=of A] {B};

    \draw [-{>[sep=2pt]}] (A) to [bend left=45] (B);
    \draw [- >] (A) to [bend right=45] (B);
\end{tikzpicture}
```

Indeed, adding a `sep` to an arrow tip is *very* desirable, so you will usually write something like `>={To[sep]}` somewhere near the start of your files.

One arrow tip kind can be quite useful in this context: The arrow tip kind `_`. It draws nothing and has zero length, *but* it has `sep` set as a default option. Since it is a single letter shorthand, you can write short and clean “code” in this way:

`\tikz \draw [->_] (0,0) -- (1,0);`

`\tikz \draw [->__>] (0,0) -- (1,0);`

However, using the `sep` option will be faster than using the `_` arrow tip and it also allows you to specify the desired length directly.

16.4.3 Specifying the Line End

In the previous examples of sequences of arrow tips, the line of the path always ended at the last of the arrow tips (for end specifications) or at the first of the arrow tips (for start specifications). Often, this is what you may want, but not always. Fortunately, it is quite easy to specify the desired end of the line: The special single char shorthand `.` is reserved to indicate that last arrow that is still part of the line; in other words, the line will stop at the last arrow before `.` is encountered (for end specifications) or at the first arrow following `.` (for start specifications).



```
\tikz [very thick] \draw [<<->>] (0,0) -- (2,0);
```



```
\tikz [very thick] \draw [<.<-.>>] (0,0) -- (2,0);
```



```
\tikz [very thick] \draw [<.<->.] (0,0) -- (2,0);
```



```
\tikz [very thick] \draw [<.<->.] (0,0) to [bend left] (2,0);
```

It is permissible that there are several dots in a specification, in this case the first one “wins” (for end specifications, otherwise the last one).

Note that `.` is parsed as any other shorthand. In particular, if you wish to add a dot after a normal arrow tip kind name, you need to add brackets:



```
\tikz [very thick] \draw [<{Stealth[] . Stealth[] Stealth[]}+] (0,0) -- (2,0);
```

Adding options to `.` is permissible, but they have no effect. In particular, `sep` has no effect since a dot is not an arrow.

16.4.4 Defining Shorthands

It is often desirable to create “shorthands” for the names of arrow tips that you are going to use very often. Indeed, in most documents you will only need a single arrow tip kind and it would be useful that you could refer to it just as `>` in your arrow tip specifications. As another example, you might constantly wish

to switch between a filled and a non-filled circle as arrow tips and would like to use `*` and `o` as shorthands for these case. Finally, you might just like to shorten a long name like `Computer Modern Rightarrow` down to just, say `To` or something similar.

All of these case can be addressed by defining appropriate shorthands. This is done using the following handler:

Key handler `<key>/ .tip=<end specification>`

Defined the `<key>` as a name that can be used inside arrow tip specifications. If the `<key>` has a path before it, this path is ignored (so there is only one “namespace” for arrow tips). Whenever it is used, it will be replaced by the `<end specification>`. Note that you must *always* provide (only) an end specification; when the `<key>` is used inside a start specification, the ordering and the meaning of the keys inside the `<end specification>` are translated automatically.

```
→» \usetikzlibrary {arrows.meta}
\begin{tikzpicture}
\draw [foo /.tip = {Stealth[sep]}] (0,0) -- (2,0);
\end{tikzpicture}
```



```
→» \usetikzlibrary {arrows.meta}
\begin{tikzpicture}
\draw [foo /.tip = {Stealth[sep] Latex[sep]}, bar /.tip = {Stealth[length=10pt,open]}] (0,0) -- (2,0);
\end{tikzpicture}
```

In the last of the examples, we used `foo[red]` to make the arrows red. Any options given to a shorthand upon use will be passed on to the actual arrows tip for which the shorthand stands. Thus, we could also have written `Stealth[sep,red] Latex[sep,red]` instead of `foo[red]`. In other words, the “replacement” of a shorthand by its “meaning” is a semantic replacement rather than a syntactic replacement. In particular, the `<end specification>` will be parsed immediately when the shorthand is being defined. However, this applies only to the options inside the specification, whose values are evaluated immediately. In contrast, which actual arrow tip kind is meant by a given shorthand used inside the `<end specification>` is resolved only up each use of the shorthand. This means that when you write

```
dup /.tip = »
```

and then later write

```
> /.tip = whatever
```

then `dup` will have the effect as if you had written `whatever[] whatever[]`. You will find that this behaviour is what one would expect.

There is one problem we have not yet addressed: The asymmetry of single letter arrow tips like `>` or `.`. When someone writes

```
←→ \tikz \draw [<->] (0,0) -- (1,0);
```

we rightfully expect one arrow tip pointing left at the left end and an arrow tip pointing right at the right end. However, compare

```
→→ \tikz \draw [>->] (0,0) -- (1,0);
```

```
↔ \usetikzlibrary {arrows.meta}
\begin{tikzpicture}
\draw [Stealth-Stealth] (0,0) -- (1,0);
\end{tikzpicture}
```

In both cases, we have *identical* text in the start and end specifications, but in the first case we rightfully expect the left arrow to be flipped.

The solution to this problem is that it is possible to define two names for the same arrow tip, namely one that is used inside start specifications and one for end specifications. Now, we can decree that the “name of `>`” inside start specifications is simply `<` and the above problems disappear.

To specify different names for a shorthand in start and end specifications, use the following syntax: Instead of `<key>`, you use `<name in start specifications>-<name in end specifications>`. Thus, to set the `>` key correctly, you actually need to write



```
\usetikzlibrary {arrows.meta}
\begin{tikzpicture}[\tip = Stealth]
\draw [<->] (0,0) -- (1,0);
```



```
\usetikzlibrary {arrows.meta}
\begin{tikzpicture}[\tip = Latex]
\draw [->->] (0,0) -- (1,0);
```

Note that the above also works even though we have not set `<` as an arrow tip name for end specifications! The reason this works is that the TikZ (more precisely, PGF) actually uses the following definition internally:

```
>-< /.tip = >[reversed]
```

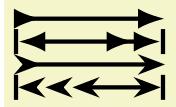
Translation: “When `<` is used in an end specification, please replace it by `>`, but reversed. Also, when `>` is used in a start specification, we also mean this inverted `>`.”

By default, `>` is a shorthand for `To` and `To` is a shorthand for `to` (an arrow from the old libraries) when `arrows.meta` is not loaded library. When `arrows.meta` is loaded, `To` is redefined to mean the same as `Computer Modern Rightarrow`.

`/tikz/>=(end arrow specification)`

(no default)

This is a short way of saying `<->/.tip=(end arrow specification)`.



```
\usetikzlibrary {arrows.meta}
\begin{tikzpicture}[scale=2, ultra thick]
\begin{scope} [>=Latex]
\draw [>->] (0pt,3ex) -- (1cm,3ex);
\draw [|<->|] (0pt,2ex) -- (1cm,2ex);
\end{scope}
\begin{scope} [>=Stealth]
\draw [>->] (0pt,1ex) -- (1cm,1ex);
\draw [|<.=>|] (0pt,0ex) -- (1cm,0ex);
\end{scope}
\end{tikzpicture}
```

`/tikz/shorten <=(length)`

(no default)

Shorten the path by `(length)` in the direction of the starting point.

`/tikz/shorten >=(length)`

(no default)

Shorten the path by `(length)` in the direction of the end point.

16.4.5 Scoping of Arrow Keys

There are numerous places where you can specify keys for an arrow tip. There is, however, one final place that we have not yet mentioned:

`/tikz/arrows=[(arrow keys)]`

(no default)

The `arrows` key, which is normally used to set the arrow tips for the current scope, can also be used to set some arrow keys for the current scope. When the argument to `arrows` starts with an opening bracket and only otherwise contains one further closing bracket at the very end, this semantic of the `arrow` key is assumed.

The `(arrow keys)` will be set for the rest of current scope. This is useful for generally setting some design parameters or for generally switching on, say, bending as in:

```
\begin{tikzpicture} [\arrows={[[bend]]} ... % Bend all arrows
```

We can now summarize which arrow keys are applied in what order when an arrow tip is used:

1. First, the so-called *defaults* are applied, which are values for the different parameters of a key. They are fixed in the definition of the key and cannot be changed. Since they are executed first, they are only the ultimate fallback.
2. The `(keys)` from the use of `arrows=[(keys)]` in all enclosing scopes.

3. Recursively, the `keys` provided with the arrow tip inside shorthands.
4. The keys provided at the beginning of an arrow tip specification in brackets.
5. The keys provided directly next to the arrow tip inside the specification.

16.5 Reference: Arrow Tips

TikZ Library `arrows.meta`

```
\usepgflibrary{arrows.meta} % LATEX and plain TEX and pure pgf
\usepgflibrary[arrows.meta] % ConTEXt and pure pgf
\usetikzlibrary{arrows.meta} % LATEX and plain TEX when using TikZ
\usetikzlibrary[arrows.meta] % ConTEXt when using TikZ
```

This library defines a large number of standard “meta” arrow tips. “Meta” means that you can configure these arrow tips in many different ways like changing their size or their line caps and joins and many other details.

The only reason this library is not loaded by default is for compatibility with older versions of TikZ. You can, however, safely load and use this library alongside the older libraries `arrows` and `arrows.spaced`.

The different arrow tip kinds defined in the `arrows.meta` library can be classified in different groups:

- *Barbed* arrow tips consist mainly of lines that “point backward” from the tip of the arrow and which are not filled. For them, filling has no effect. A typical example is →. Here is the list of defined arrow tips:

Appearance of the below at line width	0.4pt	0.8pt	1.6pt
<code>Arc Barb[]</code>	→ thin	→ thick	→
<code>Bar[]</code>	→ thin	→ thick	→
<code>Bracket[]</code>	→ thin	→ thick	→
<code>Hooks[]</code>	→ thin	→ thick	→
<code>Parenthesis[]</code>	→ thin	→ thick	→
<code>Straight Barb[]</code>	→ thin	→ thick	→
<code>Tee Barb[]</code>	→ thin	→ thick	→

All of these arrow tips can be configured and resized in many different ways as described in the following. Above, they are shown at their “natural” sizes, which are chosen in such a way that for a line width of 0.4pt their width matches the height of a letter “x” in Computer Modern at 11pt (with some “overshooting” to create visual consistency).

- *Mathematical* arrow tips are actually a subclass of the barbed arrow tips, but we list them separately. They contain arrow tips that look exactly like the tips of arrows used in mathematical fonts such as the \to-symbol → from standard T_EX.

Appearance of the below at line width	0.4pt	0.8pt	1.6pt
<code>Classical TikZ Rightarrow[]</code>	→ thin	→ thick	→
<code>Computer Modern Rightarrow[]</code>	→ thin	→ thick	→
<code>Implies[]</code> on double line	== thin	== thick	==
<code>To[]</code>	→ thin	→ thick	→

The To arrow tip is a shorthand for Computer Modern Rightarrow when `arrows.meta` is loaded.

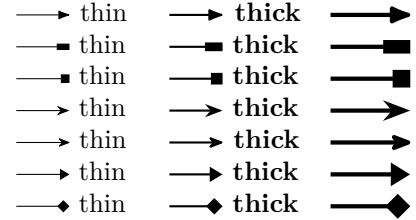
- *Geometric* arrow tips consist of a filled shape like a kite or a circle or a “stealth-fighter-like” shape. A typical example is →. These arrow tips can also be used in an “open” variant as in →.

Appearance of the below at line width	0.4pt	0.8pt	1.6pt
<code>Circle[]</code>	● thin	● thick	●
<code>Diamond[]</code>	◆ thin	◆ thick	◆
<code>Ellipse[]</code>	○ thin	○ thick	○
<code>Kite[]</code>	◇ thin	◇ thick	◇
<code>Latex[]</code>	→ thin	→ thick	→

```

Latex[round]
Rectangle[]
Square[]
Stealth[]
Stealth[round]
Triangle[]
Turned Square[]

```



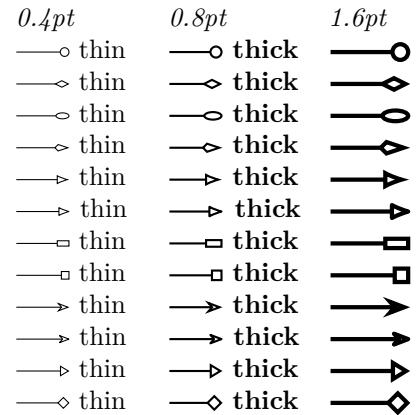
Here are the “open” variants:

Appearance of the below at line width

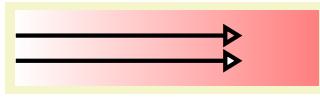
```

Circle/open]
Diamond/open]
Ellipse/open]
Kite/open]
Latex/open]
Latex[round,open]
Rectangle/open]
Square/open]
Stealth/open]
Stealth[round,open]
Triangle/open]
Turned Square/open]

```



Note that “open” arrow tips are not the same as “filled with white”, which is also available (just say `fill=white`). The difference is that the background will “shine through” an open arrow, while a filled arrow always obscures the background:



```

\usetikzlibrary {arrows.meta}
\begin{tikzpicture}
\shade [left color=white, right color=red!50] (0,0) rectangle (4,1);
\draw [ultra thick,-{Triangle[open]}] (0,2/3) -- ++ (3,0);
\draw [ultra thick,-{Triangle[fill=white]}] (0,1/3) -- ++ (3,0);
\end{tikzpicture}

```

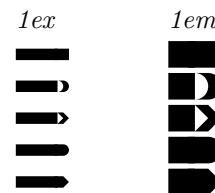
- *Cap* arrow tips are used to add a “cap” to the end of a line. The graphic languages underlying TikZ (PDF, POSTSCRIPT or SVG) all support three basic types of line caps on a very low level: round, rectangular, and “butt”. Using cap arrow tips, you can add new caps to lines and use different caps for the end and the start. An example is the line

Appearance of the below at line width

```

Butt Cap[]
Fast Round[]
Fast Triangle[]
Round Cap[]
Triangle Cap[]

```



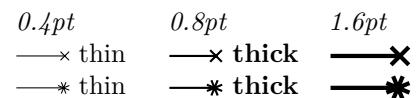
- *Special* arrow tips are used for some specific purpose and do not fit into the above categories.

Appearance of the below at line width

```

Rays[]
Rays[n=8]

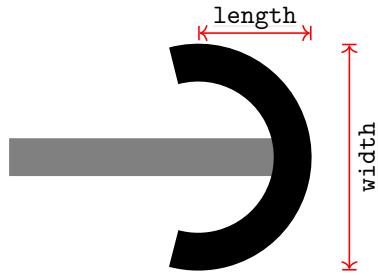
```



16.5.1 Barbed Arrow Tips

Arrow Tip Kind Arc Barb

This arrow tip attaches an arc to the end of the line whose angle is given by the `arc` option. The `length` and `width` parameters refer to the size of the arrow tip for `arc` set to 180 degrees, which is why in the example for `arc=210` the actual length is larger than the specified `length`. The line width is taken into account for the computation of the length and width. Use the `round` option to add round caps to the end of the arcs.



Appearance of the below at line width

```
Arc Barb[]
Arc Barb[sep] Arc Barb[]
Arc Barb[sep] . Arc Barb[]
Arc Barb[arc=120]
Arc Barb[arc=270]
Arc Barb[length=2pt]
Arc Barb[length=2pt, width=5pt]
Arc Barb[line width=2pt]
Arc Barb[reversed]
Arc Barb[round]
Arc Barb[slant=.3]
Arc Barb[left]
Arc Barb[right]
Arc Barb[harpーンon, reversed]
Arc Barb[red]
```

The following options have no effect: `open`, `fill`.

On double lines, the arrow tip will not look correct.

Arrow Tip Kind Bar

A simple bar. This is a simple instance of Tee Barb for length zero.

Arrow Tip Kind Bracket

This is an instance of the **TeeBarb** arrow tip that results in something resembling a bracket. Just like the **Parenthesis** arrow tip, a **Bracket** is not modelled from a text square bracket, but rather its size has been chosen so that it fits with the other arrow tips.



Appearance of the below at line width

```
Bracket []
Bracket [sep] Bracket []
Bracket [sep] . Bracket []
Bracket [reversed]
Bracket [round]
Bracket [slant=.3]
Bracket [left]
Bracket [right]
Bracket [harpoon, reversed]
Bracket [red]
```

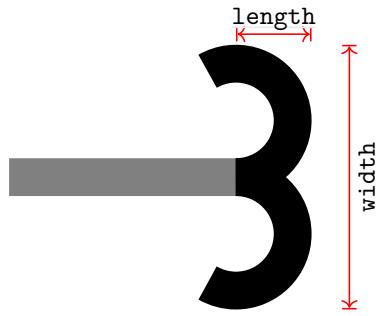
The image displays a grid of line samples for comparison. The columns represent different line thicknesses: 'thin' (left), 'thick' (middle), and 'very thick' (right). Each row contains three pairs of lines, each pair consisting of a solid line and a dashed line of the same thickness. This allows for a visual comparison of how line weight affects readability.

The following options have no effect: `open`, `fill`.

On double lines, the arrow tip will not look correct.

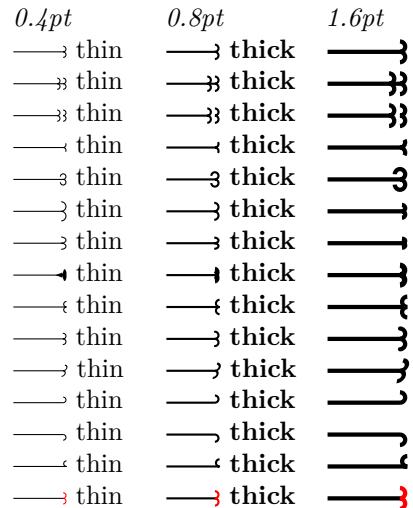
Arrow Tip Kind Hooks

This arrow tip attaches two “hooks” to the end of the line. The `length` and `width` parameters refer to the size of the arrow tip if both arcs are 180 degrees; in the example the arc is 210 degrees and, thus, the arrow is actually longer than the `length` dictates. The line width is taken into account for the computation of the length and width. The `arc` option is used to specify the angle of the arcs. Use the `round` option to add round caps to the end of the arcs.



Appearance of the below at line width

```
Hooks []
Hooks [sep] Hooks []
Hooks [sep] . Hooks []
Hooks [arc=120]
Hooks [arc=270]
Hooks [length=2pt]
Hooks [length=2pt, width=5pt]
Hooks [line width=2pt]
Hooks [reversed]
Hooks [round]
Hooks [slant=.3]
Hooks [left]
Hooks [right]
Hooks [harpoon, reversed]
Hooks [red]
```



The following options have no effect: `open`, `fill`.

On `double` lines, the arrow tip will not look correct.

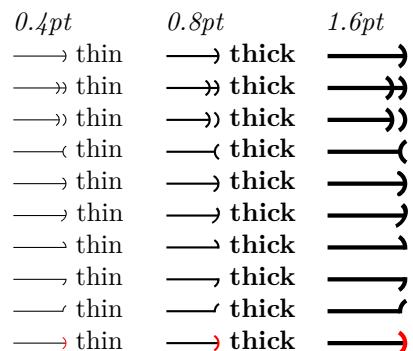
Arrow Tip Kind Parenthesis

This arrow tip is an instantiation of the `ArcBarb` so that it resembles a parenthesis. However, the idea is not to recreate a “real” parenthesis as it is used in text, but rather a “bow” at a size that harmonizes with the other arrow tips at their default sizes.



Appearance of the below at line width

```
Parenthesis []
Parenthesis [sep] Parenthesis []
Parenthesis [sep] . Parenthesis []
Parenthesis [reversed]
Parenthesis [round]
Parenthesis [slant=.3]
Parenthesis [left]
Parenthesis [right]
Parenthesis [harpoon, reversed]
Parenthesis [red]
```

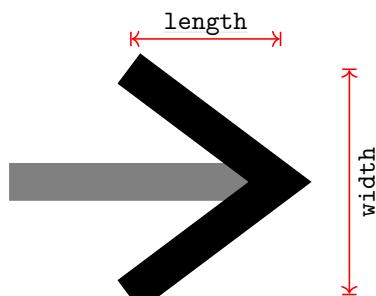


The following options have no effect: `open`, `fill`.

On `double` lines, the arrow tip will not look correct.

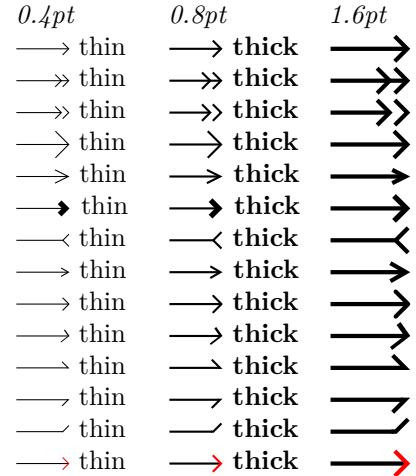
Arrow Tip Kind Straight Barb

This is the “archetypal” arrow head, consisting of just two straight lines. The `length` and `width` parameters refer to the horizontal and vertical distances between the points on the path making up the arrow tip. As can be seen, the line width of the arrow tip’s path is not taken into account. The `angle` option is particularly useful to set the opening angle at the tip of the arrow head. The `round` option gives a “softer” or “rounder” version of the arrow tip.



Appearance of the below at line width

```
Straight Barb[]
Straight Barb[] Straight Barb[]
Straight Barb[] . Straight Barb[]
Straight Barb[length=5pt]
Straight Barb[length=5pt,width=5pt]
Straight Barb[line width=2pt]
Straight Barb[reversed]
Straight Barb[angle=60:2pt 3]
Straight Barb[round]
Straight Barb[slant=.3]
Straight Barb[left]
Straight Barb[right]
Straight Barb[harpoon,reversed]
Straight Barb[red]
```

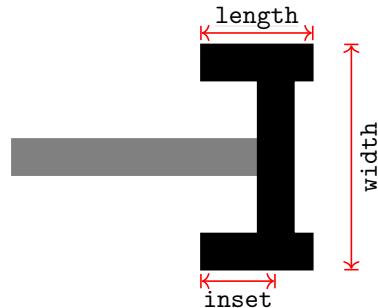


The following options have no effect: `open`, `fill`.

On `double` lines, the arrow tip will not look correct.

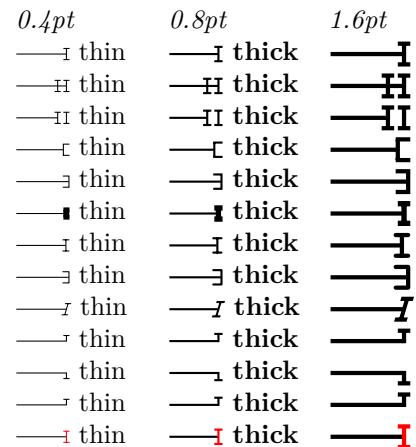
Arrow Tip Kind Tee Barb

This arrow tip attaches a little “T” on both sides of the tip. The arrow `inset` dictates the distance from the back end to the middle of the stem of the T. When the inset is equal to the length, the arrow tip is drawn as a single line, not as three lines (this is important for the “round” version since, then, the corners get rounded).



Appearance of the below at line width

```
Tee Barb[]
Tee Barb[sep] Tee Barb[]
Tee Barb[sep] . Tee Barb[]
Tee Barb[inset=0pt]
Tee Barb[inset'=0pt 1]
Tee Barb[line width=2pt]
Tee Barb[round]
Tee Barb[round,inset'=0pt 1]
Tee Barb[slant=.3]
Tee Barb[left]
Tee Barb[right]
Tee Barb[harpoon,reversed]
Tee Barb[red]
```



The following options have no effect: `open`, `fill`.

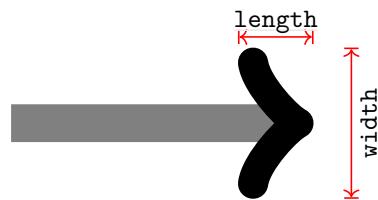
On `double` lines, the arrow tip will not look correct.

16.5.2 Mathematical Barbed Arrow Tips

Arrow Tip Kind Classical TikZ Rightarrow

This arrow tip is the “old” or “classical” arrow tip that used to be the standard in TikZ in earlier versions. It was modelled on an old version of the tip of `\rightarrow` of the Computer Modern fonts. However, this “old version” was really old, Donald Knuth (the designer of both TeX and of the Computer Modern fonts) replaced the arrow tip of the mathematical fonts in 1992.

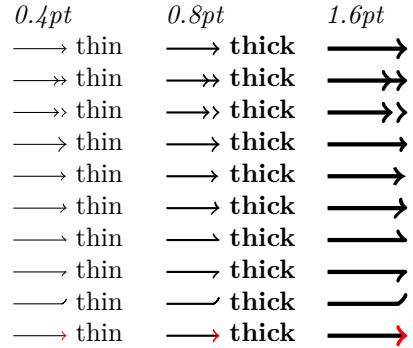
The main problem with this arrow tip is that it is “too small” at its natural size. I recommend using the new Computer Modern Rightarrow arrow tip instead, which matches the current `\rightarrow`. This new version



is also the default used as `>` and as `To`, now.

Appearance of the below at line width

<code>Classical TikZ Rightarrow[]</code>	<code>0.4pt</code>	<code>0.8pt</code>	<code>1.6pt</code>
<code>Classical TikZ Rightarrow[sep]</code>			
<code>Classical TikZ Rightarrow[sep] . Classical TikZ Rightarrow[]</code>			
<code>Classical TikZ Rightarrow[length=3pt]</code>			
<code>Classical TikZ Rightarrow[sharp]</code>			
<code>Classical TikZ Rightarrow[slant=.3]</code>			
<code>Classical TikZ Rightarrow[left]</code>			
<code>Classical TikZ Rightarrow[right]</code>			
<code>Classical TikZ Rightarrow[harpoon,reversed]</code>			
<code>Classical TikZ Rightarrow[red]</code>			

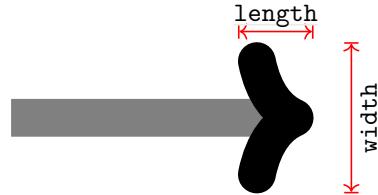


The following options have no effect: `open`, `fill`.

On `double` lines, the arrow tip will not look correct.

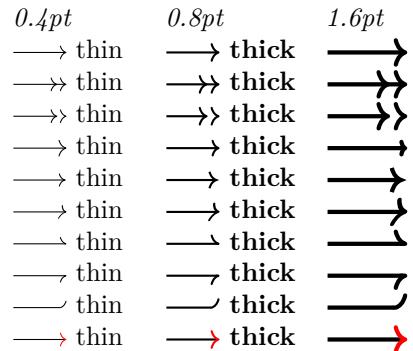
Arrow Tip Kind Computer Modern Rightarrow

For a line width of `0.4pt` (the default), this arrow tip looks very much like `\rightarrow` of the Computer Modern math fonts. However, it is not a “perfect” match: the line caps and joins of the “real” \rightarrow are rounded differently from this arrow tip; but it takes a keen eye to notice the difference. When the `arrows.meta` library is loaded, this arrow tip becomes the default of `To` and, thus, is used whenever `>` is used (unless, of course, you redefined `>`).



Appearance of the below at line width

<code>Computer Modern Rightarrow[]</code>	<code>0.4pt</code>	<code>0.8pt</code>	<code>1.6pt</code>
<code>Computer Modern Rightarrow[sep]</code>			
<code>Computer Modern Rightarrow[sep] . Computer Modern Rightarrow[]</code>			
<code>Computer Modern Rightarrow[length=3pt]</code>			
<code>Computer Modern Rightarrow[sharp]</code>			
<code>Computer Modern Rightarrow[slant=.3]</code>			
<code>Computer Modern Rightarrow[left]</code>			
<code>Computer Modern Rightarrow[right]</code>			
<code>Computer Modern Rightarrow[harpoon,reversed]</code>			
<code>Computer Modern Rightarrow[red]</code>			

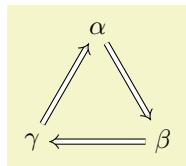


The following options have no effect: `open`, `fill`.

On `double` lines, the arrow tip will not look correct.

Arrow Tip Kind Implies

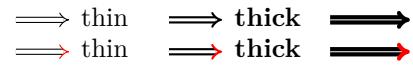
This arrow tip makes only sense in conjunction with the `double` option. The idea is that you attach it to a double line to get something that looks like TeX’s `\implies` arrow (\implies). A typical use of this arrow tip is



```
\usetikzlibrary {arrows.meta,graphs}
\begin{tikzpicture} \graph [clockwise=3, math nodes,
edges = {double equal sign distance, -Implies}] {
"\alpha", "\beta", "\gamma";
"\alpha" --> "\beta" --> "\gamma" --> "\alpha"
};
```

Appearance of the below at line width

<code>Implies[]</code> on double line	<code>0.4pt</code>	<code>0.8pt</code>	<code>1.6pt</code>
<code>Implies[red]</code> on double line			



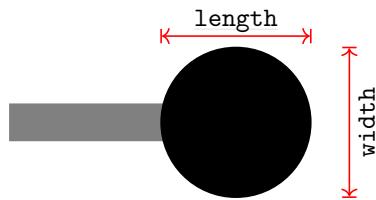
Arrow Tip Kind To

This is a shorthand for `Computer Modern Rightarrow` when the `arrows.meta` library is loaded. Otherwise, it is a shorthand for the classical TikZ rightarrow.

16.5.3 Geometric Arrow Tips

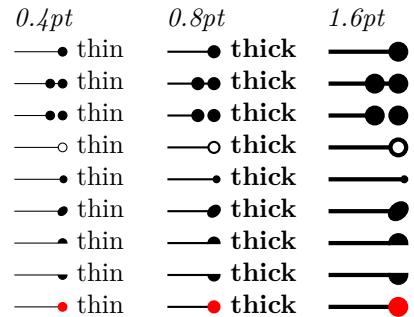
Arrow Tip Kind Circle

Although this tip is called “circle”, you can also use it to draw ellipses if you set the length and width to different values. Neither round nor reversed has any effect on this arrow tip.



Appearance of the below at line width

```
Circle[]
Circle[] Circle[]
Circle[] . Circle[]
Circle[open]
Circle[length=3pt]
Circle[slant=.3]
Circle[left]
Circle[right]
Circle[red]
```

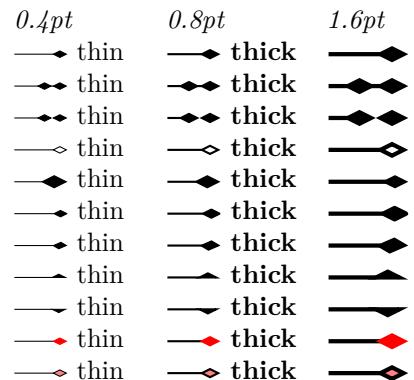


Arrow Tip Kind Diamond

This is an instance of Kite where the length is larger than the width.

Appearance of the below at line width

```
Diamond[]
Diamond[] Diamond[]
Diamond[] . Diamond[]
Diamond[open]
Diamond[length=10pt]
Diamond[round]
Diamond[slant=.3]
Diamond[left]
Diamond[right]
Diamond[red]
Diamond[fill=red!50]
```

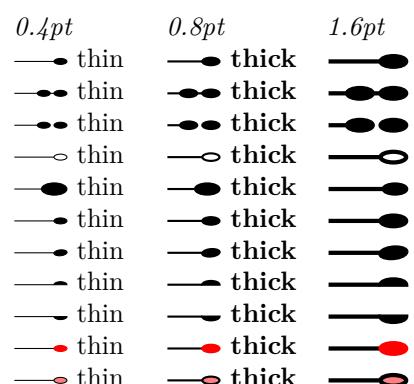


Arrow Tip Kind Ellipse

This is a shorthand for a “circle” that is twice as wide as high.

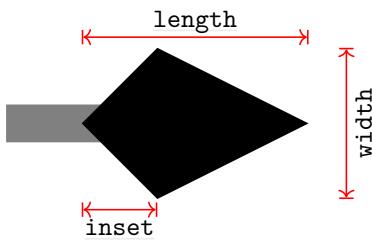
Appearance of the below at line width

```
Ellipse[]
Ellipse[] Ellipse[]
Ellipse[] . Ellipse[]
Ellipse[open]
Ellipse[length=10pt]
Ellipse[round]
Ellipse[slant=.3]
Ellipse[left]
Ellipse[right]
Ellipse[red]
Ellipse[fill=red!50]
```



Arrow Tip Kind Kite

This arrow tip consists of four lines that form a “kite”. The `inset` prescribed how far the width-axis of the kite is removed from the back end. Note that the `inset` cannot be negative, use a `Stealth` arrow tip for this.

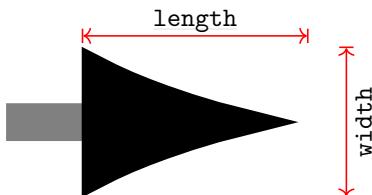


Appearance of the below at line width

```
Kite[]
Kite[sep] Kite[]
Kite[sep] . Kite[]
Kite[open]
Kite[length=6pt,width=4pt]
Kite[length=6pt,width=4pt,inset=1.5pt]
Kite[round]
Kite[slant=.3]
Kite[left]
Kite[right]
Kite[red]
```

Arrow Tip Kind Latex

This arrow tip is the same as the arrow tip used in L^AT_EX's standard pictures (via the `\vec` command), if you set the length to 4pt. The default size for this arrow tip was set slightly larger so that it fits better with the other geometric arrow tips.



Appearance of the below at line width

```
Latex[]  
Latex[sep] Latex[]  
Latex[sep] . Latex[]  
Latex[open]  
Latex[length=4pt]  
Latex[round]  
Latex[slant=.3]  
Latex[left]  
Latex[right]  
Latex[red]
```

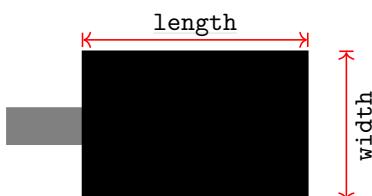
The diagram consists of three vertical columns of arrows pointing to the right. The first column contains 10 arrows, all labeled 'thin' above them. The second column contains 10 arrows, all labeled 'thick' above them. The third column contains 10 arrows, with the first six labeled 'thin' and the last four labeled 'thick'.

Arrow Tip Kind LaTeX

Another spelling for the **Latex** arrow tip.

Arrow Tip Kind Rectangle

A rectangular arrow tip. By default, it is twice as long as high.



Appearance of the below at line width

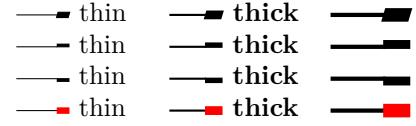
```
Rectangle[]  
Rectangle[sep] Rectangle[]  
Rectangle[sep] . Rectangle[]  
Rectangle[open]  
Rectangle[length=4pt]  
Rectangle[round]
```

The figure displays three groups of line styles, each group corresponding to a different line width: 0.4pt, 0.8pt, and 1.6pt. Within each group, there are three distinct line types: thin, thick, and dashed.

```

Rectangle[slant=.3]
Rectangle[left]
Rectangle[right]
Rectangle[red]

```



Arrow Tip Kind Square

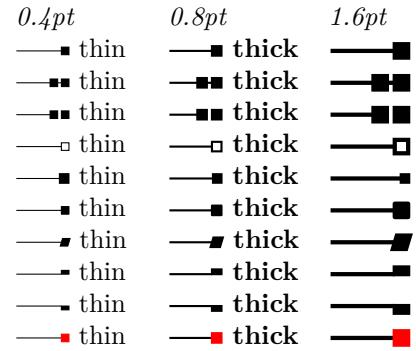
An instance of the `Rectangle` whose width is identical to the length.

Appearance of the below at line width

```

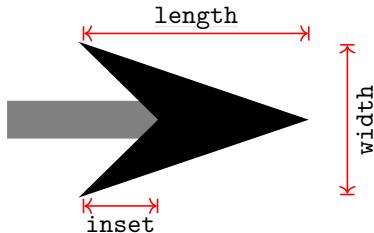
Square[]
Square[sep] Square[]
Square[sep] . Square[]
Square[open]
Square[length=4pt]
Square[round]
Square[slant=.3]
Square[left]
Square[right]
Square[red]

```



Arrow Tip Kind Stealth

This arrow tip is similar to a `Kite`, only the `inset` now counts “inwards”. Because of that sharp angles, for this arrow tip it makes quite a difference, visually, if use the `round` option. Also, using the `harpoon` option (or `left` or `right`) will *lengthen* the arrow tip because of the even sharper corner at the tip.

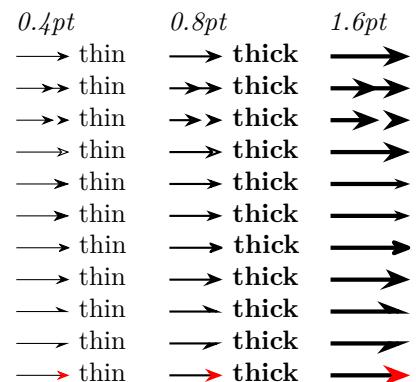


Appearance of the below at line width

```

Stealth[]
Stealth[sep] Stealth[]
Stealth[sep] . Stealth[]
Stealth[open]
Stealth[length=6pt,width=4pt]
Stealth[length=6pt,width=4pt,inset=1.5pt]
Stealth[round]
Stealth[slant=.3]
Stealth[left]
Stealth[right]
Stealth[red]

```



Arrow Tip Kind Triangle

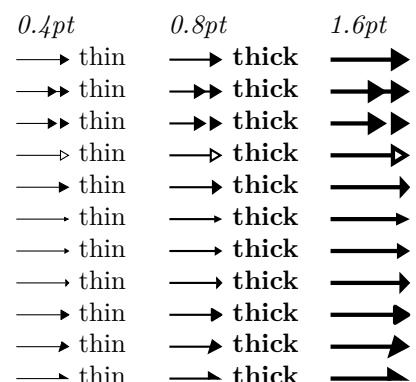
An instance of a `Kite` with zero inset.

Appearance of the below at line width

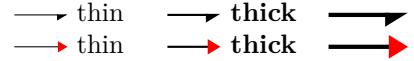
```

Triangle[]
Triangle[sep] Triangle[]
Triangle[sep] . Triangle[]
Triangle[open]
Triangle[length=4pt]
Triangle[angle=45:1pt 3]
Triangle[angle=60:1pt 3]
Triangle[angle=90:1pt 3]
Triangle[round]
Triangle[slant=.3]
Triangle[left]

```



```
Triangle[right]
Triangle[red]
```



Arrow Tip Kind Turned Square

An instance of a Kite with identical width and height and mid-inset.

```
Appearance of the below at line width
Turned Square[]

Turned Square[sep] Turned Square[]
Turned Square[sep] . Turned Square[]

Turned Square[open]

Turned Square[length=4pt]

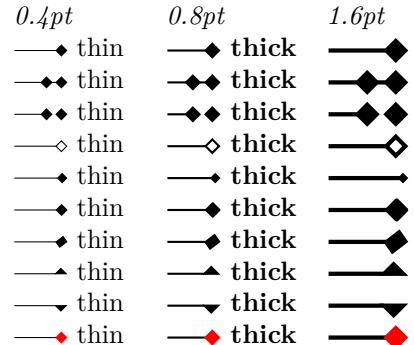
Turned Square[round]

Turned Square[slant=.3]

Turned Square[left]

Turned Square[right]

Turned Square[red]
```



16.5.4 Caps

Recall that a *cap* is a way of ending a line. The graphic languages underlying TikZ (PDF, POSTSCRIPT or SVG) all support three basic types of line caps on a very low level: round, rectangular, and “butt”. Using cap arrow tips, you can add new caps to lines and use different caps for the end and the start.

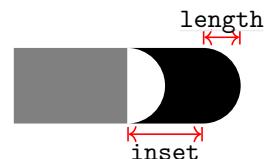
Arrow Tip Kind Butt Cap

This arrow tip ends the line “in the normal way” with a straight end. This arrow tip is only need to “cover up” the actual line cap, if this happens to differ from the normal cap. In the following example, the line cap is “round”, but, nevertheless, the right end is a “butt” cap:

```
\usepackage{arrows.meta}
\begin{tikzpicture}
\draw [line width=1ex, line cap=round, -Butt Cap] (0,0) -- (1,0);
\end{tikzpicture}
```

Arrow Tip Kind Fast Round

This arrow tip is not really a cap, you use it in conjunction with (typically) the RoundCap. The idea is that you end your line using the round cap and then add several Fast Rounds. As for RoundCap, the length parameter dictates the length of the “main part”, the inset sets the length of a line that comes before this tip.

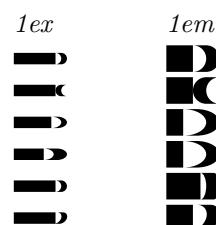


```
\usepackage{arrows.meta}
\begin{tikzpicture}
\draw [line width=1ex,
       -{Round Cap[] . Fast Round[] Fast Round[]}]
(0,0) -- (1,0);
\end{tikzpicture}
```

Note that in conjunction with the `bend` option, this works even quite well for curves:

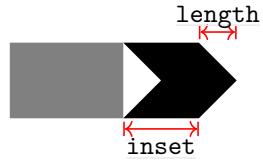
```
\usepackage{arrows.meta,bending}
\begin{tikzpicture}
[f/.tip = Fast Round] % shorthand
\draw [line width=1ex, -{[bend] Round Cap[] . f f f}]
(0,0) to [bend left] (1,0);
\end{tikzpicture}
```

```
Appearance of the below at line width
Fast Round[]
Fast Round[reversed]
Fast Round[cap angle=60]
Fast Round[cap angle=60,inset=5pt]
Fast Round[length=.5ex]
Fast Round[slant=.3]
```



Arrow Tip Kind Fast Triangle

This arrow tip works like `FastRound`, only for triangular caps.



```
\usetikzlibrary {arrows.meta}
\begin{tikzpicture}
\draw [line width=1ex,
-{Triangle Cap [] . Fast Triangle[] . Fast Triangle[]}]
(0,0) -- (1,0);
\end{tikzpicture}
```

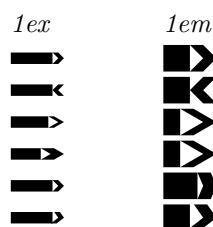
Again, this tip works well for curves:



```
\usetikzlibrary {arrows.meta,bending}
\begin{tikzpicture}
\f/.tip = Fast Triangle % shorthand
\draw [line width=1ex, -{[bend] Triangle Cap[] . f f f}]
(0,0) to [bend left] (1,0);
\end{tikzpicture}
```

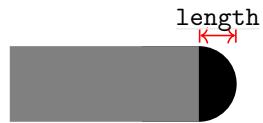
Appearance of the below at line width

- `Fast Triangle[]`
- `Fast Triangle[reversed]`
- `Fast Triangle[cap angle=60]`
- `Fast Triangle[cap angle=60,inset=5pt]`
- `Fast Triangle[length=.5ex]`
- `Fast Triangle[slant=.3]`



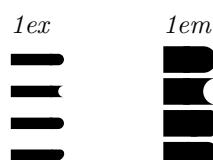
Arrow Tip Kind Round Cap

This arrow tip ends the line using a half circle or, if the length has been modified, a half-ellipse.



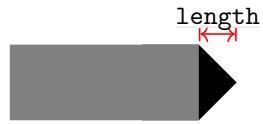
Appearance of the below at line width

- `Round Cap[]`
- `Round Cap[reversed]`
- `Round Cap[length=.5ex]`
- `Round Cap[slant=.3]`



Arrow Tip Kind Triangle Cap

This arrow tip ends the line using a triangle whose length is given by the `length` option.



You can get any angle you want at the tip by specifying a length that is an appropriate multiple of the line width. The following options does this computation for you:

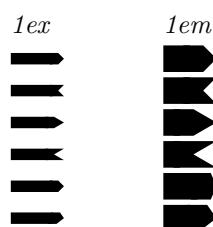
`/pgf/arrow keys/cap angle=<angle>`

(no default)

Sets `length` to an appropriate multiple of the line width so that the angle of a `Triangle Cap` is exactly `<angle>` at the tip.

Appearance of the below at line width

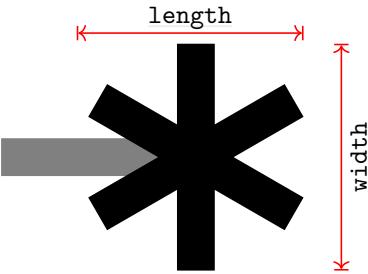
- `Triangle Cap[]`
- `Triangle Cap[reversed]`
- `Triangle Cap[cap angle=60]`
- `Triangle Cap[cap angle=60,reversed]`
- `Triangle Cap[length=.5ex]`
- `Triangle Cap[slant=.3]`



16.5.5 Special Arrow Tips

Arrow Tip Kind Rays

This arrow tip attaches a “bundle of rays” to the tip. The number of evenly spaced rays is given by the `n` arrow key (see below). When the number is even, the rays will lie to the left and to the right of the direction of the arrow; when the number is odd, the rays are rotated in such a way that one of them points perpendicular to the direction of the arrow (this is to ensure that no ray points in the direction of the line, which would look strange). The `length` and `width` describe the length and width of an ellipse into which the rays fit.



Appearance of the below at line width

```
Rays []
Rays [sep] Rays []
Rays [sep] . Rays []
Rays [width'=0pt 2]
Rays [round]
Rays [n=2]
Rays [n=3]
Rays [n=4]
Rays [n=5]
Rays [n=6]
Rays [n=7]
Rays [n=8]
Rays [n=9]
Rays [slant=.3]
Rays [left]
Rays [right]
Rays [left,n=5]
Rays [right,n=5]
Rays [red]
```

<i>0.4pt</i>	<i>0.8pt</i>	<i>1.6pt</i>
—× thin	—× thick	—×
—×× thin	—×× thick	—××
—×× thin	—×× thick	—××
—× thin	—× thick	—×
—× thin	—× thick	—×
—+ thin	—+ thick	—+
—γ thin	—γ thick	—γ
—× thin	—× thick	—×
—* thin	—* thick	—*
—* thin	—* thick	—*
—* thin	—* thick	—*
—* thin	—* thick	—*
—* thin	—* thick	—*
—× thin	—× thick	—×
—γ thin	—γ thick	—γ
—γ thin	—γ thick	—γ
—→ thin	—→ thick	—↑
—→ thin	—→ thick	—↑
—× thin	—× thick	—×

`/pgf/arrow keys/n=<number>`

(no default, initially 4)

Sets the number of rays in a `Rays` arrow tip.

17 Nodes and Edges

17.1 Overview

In the present section, the usage of `nodes` in TikZ is explained. A node is typically a rectangle or circle or another simple shape with some text on it.

Nodes are added to paths using the special path operation `node`. Nodes *are not part of the path itself*. Rather, they are added to the picture just before or after the path has been drawn.

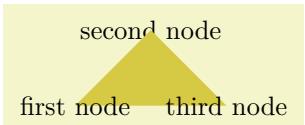
In Section 17.2 the basic syntax of the node operation is explained, followed in Section 17.3 by the syntax for multi-part nodes, which are nodes that contain several different text parts. After this, the different options for the text in nodes are explained. In Section 17.5 the concept of `anchors` is introduced along with their usage. In Section 17.7 the different ways transformations affect nodes are studied. Sections 17.8 and 17.9 are about placing nodes on or next to straight lines and curves. Section 17.11 explains how a node can be used as a “pseudo-coordinate”. Section 17.12 introduces the `edge` operation, which works similar to the `to` operation and also similar to the `node` operation.

17.2 Nodes and Their Shapes

In the simplest case, a node is just some text that is placed at some coordinate. However, a node can also have a border drawn around it or have a more complex background and foreground. Indeed, some nodes do not have a text at all, but consist solely of the background. You can name nodes so that you can reference their coordinates later in the same picture or, if certain precautions are taken as explained in Section 17.13, also in different pictures.

There are no special TeX commands for adding a node to a picture; rather, there is path operation called `node` for this. Nodes are created whenever TikZ encounters `node` or `coordinate` at a point on a path where it would expect a normal path operation (like `--(1,1)` or `rectangle (1,1)`). It is also possible to give node specifications *inside* certain path operations as explained later.

The node operation is typically followed by some options, which apply only to the node. Then, you can optionally *name* the node by providing a name in parentheses. Lastly, for the `node` operation you must provide some label text for the node in curly braces, while for the `coordinate` operation you may not. The node is placed at the current position of the path either *after the path has been drawn* or (more seldomly and only if you add the `behind path` option) *just before the path is drawn*. Thus, all nodes are drawn “on top” or “behind” the path and are retained until the path is complete. If there are several nodes on a path, perhaps some behind and some on top of the path, first come the nodes behind the path in the order they were encountered, then comes that path, and then come the remaining node, again in the order they are encountered.



```
\tikz \fill [fill=yellow!80!black]
  (0,0) node           {first node}
  -- (1,1) node[behind path] {second node}
  -- (2,0) node           {third node};
```

17.2.1 Syntax of the Node Command

The syntax for specifying nodes is the following:

```
\path ... node <foreach statements> [<options>] (<name>) at(<coordinate>) :<animation
attribute>=<options> {<node contents>} ... ;
```

Since this path operation is one of the most involved around, let us go over it step by step.

Order of the parts of the specification. Everything between “`node`” and the opening brace of a node is optional. If there are `<foreach statements>`, they must come first, directly following “`node`”. Other than that, the ordering of all the other elements of a node specification (the `<options>`, the `<name>`, `<coordinate>`, and `<animation attribute>`) is arbitrary, indeed, there can be multiple occurrences of any of these elements (although for the name and the coordinate this makes no sense).

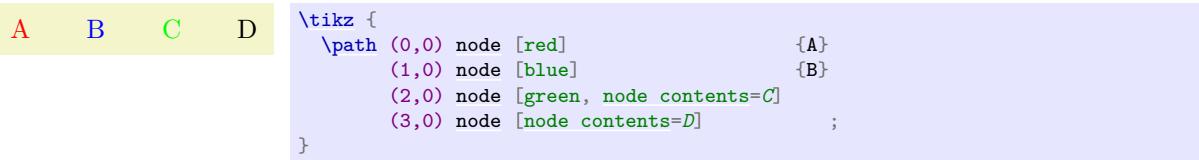
The text of a node. At the end of a node, you must (normally) provide some `<node contents>` in curly braces; indeed, the “end” of the node specification is detected by the opening curly brace. For normal nodes it is possible to use “fragile” stuff inside the `<node contents>` like the `\verb` command (for the

technically savvy: code inside the `<node contents>` is allowed to change catcodes; however, this rule does not apply to “nodes on a path” to be discussed later).

Instead of giving `<node contents>` at the end of the node in curly braces, you can also use the following key:

`/tikz/node contents=<node contents>` (no default)

This key sets the contents of the node to the given text as if you had given it at the end in curly braces. When the option is used inside the options of a node, the parsing of the node stops immediately after the end of the option block. In particular, the option block cannot be followed by further option blocks or curly braces (or, rather, these do not count as part of the node specification.) Also note that the `<node contents>` may not contain fragile stuff since the catcodes get fixed upon reading the options. Here is an example:



Specifying the location of the node. Nodes are placed at the last position mentioned on the path. The effect of adding “at” to a node specification is that the coordinate given after `at` is used instead. The `at` syntax is not available when a node is given inside a path operation (it would not make any sense there).

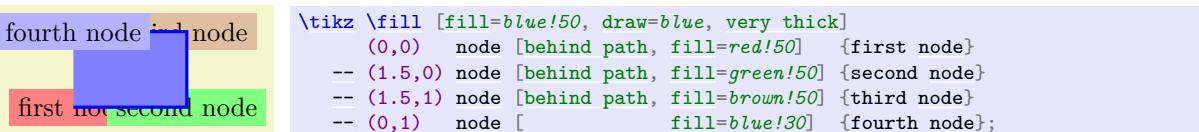
`/tikz/at=<coordinate>` (no default)

This is another way of specifying the `at` coordinate. Note that, typically, you will have to enclose the `<coordinate>` in curly braces so that a comma inside the `<coordinate>` does not confuse TeX.

Another aspect of the “location” of a node is whether it appears *in front of* or *behind* the current path. You can change which of these two possibilities happens on a node-by-node basis using the following keys:

`/tikz/behind path` (no value)

When this key is set, either as a local option for the node or some surrounding scope, the node will be drawn behind the current path. For this, TikZ collects all nodes defined on the current path with this option set and then inserts all of them, in the order they appear, just before it draws the path. Thus, several nodes with this option set may obscure one another, but never the path itself. “Just before it draws the path” actually means that the nodes are inserted into the page output just before any pre-actions are applied to the path (see below for what pre-actions are).



Note that `behind path` only applies to the current path; not to the current scope or picture. To put a node “behind everything” you need to use layers and options like `on background layer`, see the `backgrounds` library in Section 45.

`/tikz/in front of path` (no value)

This is the opposite of `behind path`: It causes nodes to be drawn on top of the path. Since this is the default behaviour, you usually do not need this option; it is only needed when an enclosing scope has used `behind path` and you now wish to “switch back” to the normal behaviour.

The name of a node. The `(<name>)` is a name for later reference and it is optional. You may also add the option `name=<name>` to the `<option>` list; it has the same effect.

`/tikz/name=<node name>` (no default)

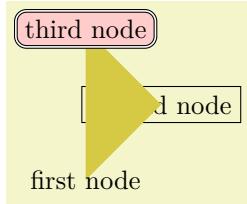
Assigns a name to the node for later reference. Since this is a “high-level” name (drivers never know of it), you can use spaces, number, letters, or whatever you like when naming a node. Thus, you can name a node just `1` or perhaps `start of chart` or even `y_1`. Your node name should *not* contain any punctuation like a dot, a comma, or a colon since these are used to detect what kind of coordinate you mean when you reference a node.

`/tikz/alias=<another node name>` (no default)

This option allows you to provide another name for the node. Giving this option multiple times will allow you to access the node via several aliases. Using the `node also` syntax, you can also assign an alias name to a node at a later point, see Section 17.14.

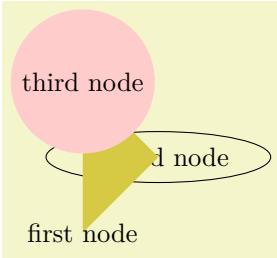
The options of a node. The `<options>` is an optional list of options that *apply only to the node* and have no effect outside. The other way round, most “outside” options also apply to the node, but not all. For example, the “outside” rotation does not apply to nodes (unless some special options are used, sigh). Also, the outside path action, like `draw` or `fill`, never applies to the node and must be given in the node (unless some special other options are used, deep sigh).

The shape of a node. As mentioned before, we can add a border and even a background to a node:



```
\tikz \fill[fill=yellow!80!black]
(0,0) node [first node]
-- (1,1) node[draw, behind path] {second node}
-- (0,2) node[fill=red!20,draw,double,rounded corners] {third node};
```

The “border” is actually just a special case of a much more general mechanism. Each node has a certain `shape` which, by default, is a rectangle. However, we can also ask TikZ to use a circle shape instead or an ellipse shape (you have to include one of the `shapes.geometric` library for the latter shape):



```
\usetikzlibrary {shapes.geometric}
\tikz \fill[fill=yellow!80!black]
(0,0) node [first node]
-- (1,1) node[ellipse,draw, behind path] {second node}
-- (0,2) node[circle,fill=red!20] {third node};
```

There are many more shapes available such as, say, a shape for a resistor or a large arrow, see the `shapes` library in Section 72 for details.

To select the shape of a node, the following option is used:

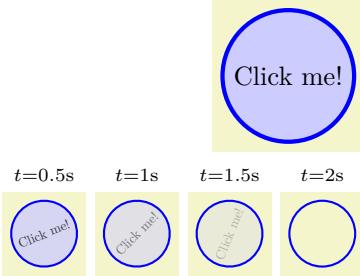
`/tikz/shape=<shape name>` (no default, initially `rectangle`)

Select the shape either of the current node or, when this option is not given inside a node but somewhere outside, the shape of all nodes in the current scope.

Since this option is used often, you can leave out the `shape=`. When TikZ encounters an option like `circle` that it does not know, it will, after everything else has failed, check whether this option is the name of some shape. If so, that shape is selected as if you had said `shape=<shape name>`.

By default, the following shapes are available: `rectangle`, `circle`, `coordinate`. Details of these shapes, like their anchors and size options, are discussed in Section 17.2.2.

Animating a node. When you say `:<animation attribute>=<options>`, an *animation* of the specified attribute is added to the node. Animations are discussed in detail in Section 26. Here is a typical example of how this syntax can be used:



```
\usetikzlibrary {animations}
\begin{tikzpicture}
  \node [fill opacity = { 0s="1", 2s="0", begin on=click },
         rotate = { 0s="0", 2s="90", begin on=click },
         fill = blue!20, draw = blue, ultra thick, circle]
    {Click me!};

```

The foreach statement for nodes. At the beginning of a node specification (and only there) you can provide multiple *<foreach statements>*, each of which has the form `foreach <var> in {<list>}` (note that there is no slash before `foreach`). When they are given, instead of a single node, multiple nodes will be created: The `<var>` will iterate over all values of `<list>` and for each of them, a new node is created. These nodes are all created using all the text following the *<foreach statements>*, but in each copy the `<var>` will have the current value of the current element in the `<list>`.

As an example, the following two codes have the same effect:

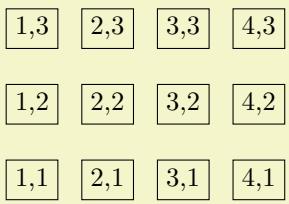
1 2 3

```
\tikz \draw (0,0) node foreach \x in {1,2,3} at (\x,0) {\x};
```

1 2 3

```
\tikz \draw (0,0) node at (1,0) {1} node at (2,0) {2} node at (3,0) {3};
```

When you provide several `foreach` statements, they work like “nested loops”:



```
\tikz \node foreach \x in {1,...,4} foreach \y in {1,2,3}
  [draw] at (\x,\y) {\x,\y};
```

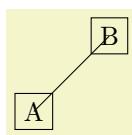
As the example shows, a `<list>` can contain ellipses (three dots) to indicated that a larger number of numbers is meant. Indeed, you can use the full power of the `\foreach` command here, including multiple parameters and options, see Section ??.

Styles for nodes. The following styles influence how nodes are rendered:

`/tikz/every node`

(style, initially empty)

This style is installed at the beginning of every node.

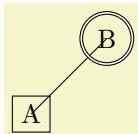


```
\begin{tikzpicture}[every node/.style={draw}]
  \draw (0,0) node {A} -- (1,1) node {B};
\end{tikzpicture}
```

`/tikz/every <shape> node`

(style, initially empty)

These styles are installed at the beginning of a node of a given `<shape>`. For example, `every rectangle node` is used for rectangle nodes, and so on.



```
\begin{tikzpicture}
  [every rectangle node/.style={draw},
   every circle node/.style={draw,double}]
  \draw (0,0) node[rectangle] {A} -- (1,1) node[circle] {B};
\end{tikzpicture}
```

`/tikz/execute at begin node=<code>` (no default)

This option causes `<code>` to be executed at the beginning of a node. Using this option multiple times will cause the code to accumulate.

`/tikz/execute at end node=<code>` (no default)

This option installs `<code>` that will be executed at the end of the node. Using this option multiple times will cause the code to accumulate.

ABCD	<pre>\begin{tikzpicture} [execute at begin node={A}, execute at end node={D}] \node[execute at begin node={B}] {C}; \end{tikzpicture}</pre>
------	--

Name scopes. It turns out that the name of a node can further be influenced using two keys:

`/tikz/name prefix=<text>` (no default, initially empty)

The value of this key is prefixed to every node inside the current scope. This includes both the naming of the node (via the `name` key or via the implicit `(<name>)` syntax) as well as any referencing of the node. Outside the scope, the nodes can (and need to) be referenced using “full name” consisting of the prefix and the node name.

The net effect of this is that you can set the name prefix at the beginning of a scope to some value and then use short and simple names for the nodes inside the scope. Later, outside the scope, you can reference the nodes via their full name:

	<pre>\tikz { \begin{scope}[name prefix = top-] \node (A) at (0,1) {A}; \node (B) at (1,1) {B}; \draw (A) -- (B); \end{scope} \begin{scope}[name prefix = bottom-] \node (A) at (0,0) {A}; \node (B) at (1,0) {B}; \draw (A) -- (B); \end{scope} \draw [red] (top-A) -- (bottom-B); }</pre>
--	---

As can be seen, name prefixing makes it easy to write reusable code.

`/tikz/name suffix=<text>` (no default, initially empty)

Works as `name prefix`, only the `<text>` is appended to every node name in the current scope.

There is a special syntax for specifying “light-weight” nodes:

`\path ... coordinate[<options>](<name>)at(<coordinate>) ... ;`

This has the same effect as

`\node[shape=coordinate[<options>]](<name>)at(<coordinate>){};`

where the `at` part may be omitted.

Since nodes are often the only path operation on paths, there are two special commands for creating paths containing only a node:

`\node`

Inside `{tikzpicture}` this is an abbreviation for `\path node`.

`\coordinate`

Inside `{tikzpicture}` this is an abbreviation for `\path coordinate`.

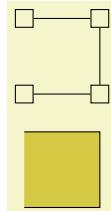
17.2.2 Predefined Shapes

PGF and TikZ define three shapes, by default:

- `rectangle`,
- `circle`, and
- `coordinate`.

By loading library packages, you can define more shapes like ellipses or diamonds; see Section 72 for the complete list of shapes.

The `coordinate` shape is handled in a special way by TikZ. When a node `x` whose shape is `coordinate` is used as a coordinate (`x`), this has the same effect as if you had said `(x.center)`. None of the special “line shortening rules” apply in this case. This can be useful since, normally, the line shortening causes paths to be segmented and they cannot be used for filling. Here is an example that demonstrates the difference:



```
\begin{tikzpicture}[every node/.style={draw}]
  \path[yshift=1.5cm,shape=rectangle]
    (0,0) node(a1){} (1,0) node(a2){}
    (1,1) node(a3){} (0,1) node(a4){};
  \filldraw[fill=yellow!80!black] (a1) -- (a2) -- (a3) -- (a4);

  \path[shape=coordinate]
    (0,0) coordinate(b1) (1,0) coordinate(b2)
    (1,1) coordinate(b3) (0,1) coordinate(b4);
  \filldraw[fill=yellow!80!black] (b1) -- (b2) -- (b3) -- (b4);
\end{tikzpicture}
```

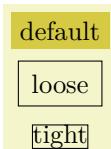
17.2.3 Common Options: Separations, Margins, Padding and Border Rotation

The exact behaviour of shapes differs, shapes defined for more special purposes (like a, say, transistor shape) will have even more custom behaviors. However, there are some options that apply to most shapes:

`/pgf/inner sep=<dimension>` (no default, initially `.3333em`)
alias `/tikz/inner sep`

An additional (invisible) separation space of `<dimension>` will be added inside the shape, between the text and the shape’s background path. The effect is as if you had added appropriate horizontal and vertical skips at the beginning and end of the text to make it a bit “larger”.

For those familiar with CSS, this is the same as `padding`.



```
\begin{tikzpicture}
  \draw (0,0) node[inner sep=0pt,draw] {tight}
        (0cm,2em) node[inner sep=5pt,draw] {loose}
        (0cm,4em) node[fill=yellow!80!black] {default};
\end{tikzpicture}
```

`/pgf/inner xsep=<dimension>` (no default, initially `.3333em`)
alias `/tikz/inner xsep`

Specifies the inner separation in the *x*-direction, only.

`/pgf/inner ysep=<dimension>` (no default, initially `.3333em`)
alias `/tikz/inner ysep`

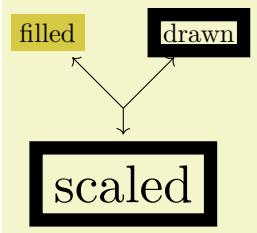
Specifies the inner separation in the *y*-direction, only.

`/pgf/outer sep=<dimension or “auto”>` (no default)
alias `/tikz/outer sep`

This option adds an additional (invisible) separation space of `<dimension>` outside the background path. The main effect of this option is that all anchors will move a little “to the outside”.

For those familiar with CSS, this is same as `margin`.

The default for this option is half the line width. When the default is used and when the background path is draw, the anchors will lie exactly on the “outside border” of the path (not on the path itself).



```
\begin{tikzpicture}
\draw [line width=5pt]
(0,0) node[fill=yellow!80!black] (f) {filled}
(2,0) node[draw] (d) {drawn}
(1,-2) node[draw,scale=2] (s) {scaled};

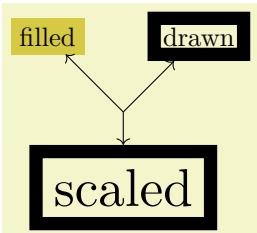
\draw[->] (1,-1) -- (f);
\draw[->] (1,-1) -- (d);
\draw[->] (1,-1) -- (s);

\end{tikzpicture}
```

As the above example demonstrates, the standard settings for the outer sep are not always “correct”. First, when a shape is filled, but not drawn, the outer sep should actually be 0. Second, when a node is scaled, for instance by a factor of 5, the outer separation also gets scaled by a factor of 5, while the line width stays at its original width; again causing problems.

In such cases, you can say `outer sep=auto` to make TikZ try to compensate for the effects described above. This is done by, firstly, setting the outer sep to 0 when no drawing is done and, secondly, setting the outer separations to half the line width (as before) times two adjustment factors, one for the horizontal separations and one for the vertical separations (see Section ?? for details on these factors). Note, however, that these factors can compensate only for transformations that are either scalings plus rotations or scalings with different magnitudes in the horizontal and the vertical direction. If you apply slanting, the factors will only approximate the correct values.

In general, it is a good idea to say `outer sep=auto` at some early stage. It is not the default mainly for compatibility with earlier versions.



```
\begin{tikzpicture}[outer sep=auto]
\draw [line width=5pt]
(0,0) node[fill=yellow!80!black] (f) {filled}
(2,0) node[draw] (d) {drawn}
(1,-2) node[draw,scale=2] (s) {scaled};

\draw[->] (1,-1) -- (f);
\draw[->] (1,-1) -- (d);
\draw[->] (1,-1) -- (s);

\end{tikzpicture}
```

`/pgf/outer xsep=<dimension>`
alias `/tikz/outer xsep`

(no default, initially $.5\pgflinewidth$)

Specifies the outer separation in the *x*-direction, only. This value will be overwritten when `outer sep` is set, either to the value given there or a computed value in case of `auto`.

`/pgf/outer ysep=<dimension>`
alias `/tikz/outer ysep`

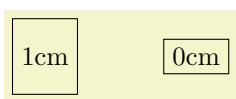
(no default, initially $.5\pgflinewidth$)

Specifies the outer separation in the *y*-direction, only.

`/pgf/minimum height=<dimension>`
alias `/tikz/minimum height`

(no default, initially 1pt)

This option ensures that the height of the shape (including the inner, but ignoring the outer separation) will be at least `<dimension>`. Thus, if the text plus the inner separation is not at least as large as `<dimension>`, the shape will be enlarged appropriately. However, if the text is already larger than `<dimension>`, the shape will not be shrunk.



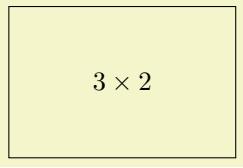
```
\begin{tikzpicture}
\draw (0,0) node[minimum height=1cm,draw] {1cm}
(2,0) node[minimum height=0cm,draw] {0cm};

\end{tikzpicture}
```

`/pgf/minimum width=<dimension>`
alias `/tikz/minimum width`

(no default, initially 1pt)

Same as `minimum height`, only for the width.



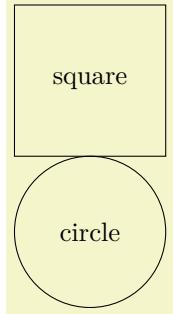
```
\begin{tikzpicture}
  \draw (0,0) node[minimum height=2cm,minimum width=3cm,draw] {$3 \times 2$};
\end{tikzpicture}
```

`/pgf/minimum size=<dimension>`

(no default)

alias `/tikz/minimum size`

Sets both the minimum height and width at the same time.



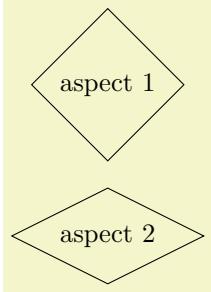
```
\begin{tikzpicture}
  \draw (0,0) node[minimum size=2cm,draw] {square};
  \draw (0,-2) node[minimum size=2cm,draw,circle] {circle};
\end{tikzpicture}
```

`/pgf/shape aspect=<aspect ratio>`

(no default)

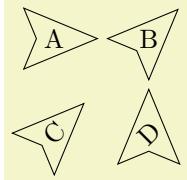
alias `/tikz/shape aspect`

Sets a desired aspect ratio for the shape. For the `diamond` shape, this option sets the ratio between width and height of the shape.



```
\usetikzlibrary {shapes.geometric}
\begin{tikzpicture}
  \draw (0,0) node[shape aspect=1,diamond,draw] {aspect 1};
  \draw (0,-2) node[shape aspect=2,diamond,draw] {aspect 2};
\end{tikzpicture}
```

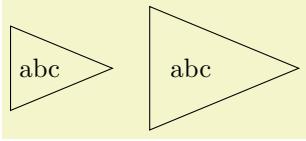
Some shapes (but not all), support a special kind of rotation. This rotation affects only the border of a shape and is independent of the node contents, but *in addition* to any other transformations.



```
\usetikzlibrary {shapes.geometric}
\tikzset{every node/.style={dart, shape border uses incircle,
  inner sep=1pt, draw}}
\tikz \node foreach \a/\b/\c in {A/0/0, B/45/0, C/0/45, D/45/45}
  [shape border rotate=\b, rotate=\c] at (\b/36,-\c/36) {\a};
```

There are two types of rotation: restricted and unrestricted. Which type of rotation is applied is determined by on how the shape border is constructed. If the shape border is constructed using an incircle, that is, a circle that tightly fits the node contents (including the `inner sep`), then the rotation can be unrestricted. If, however, the border is constructed using the natural dimensions of the node contents, the rotation is restricted to integer multiples of 90 degrees.

Why should there be two kinds of rotation and border construction? Borders constructed using the natural dimensions of the node contents provide a much tighter fit to the node contents, but to maintain this tight fit, the border rotation must be restricted to integer multiples of 90 degrees. By using an incircle, unrestricted rotation is possible, but the border will not make a very tight fit to the node contents.



```
\usetikzlibrary {shapes.geometric}
\ tikzset{every node/.style={isosceles triangle, draw}}
\begin{tikzpicture}
\node {abc};
\node [shape border uses incircle] at (2,0) {abc};
\end{tikzpicture}
```

There are PGF keys that determine how a shape border is constructed, and to specify its rotation. It should be noted that not all shapes support these keys, so reference should be made to the documentation for individual shapes.

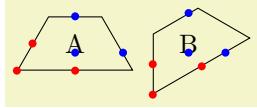
`/pgf/shape border uses incircle=<boolean>` (default `true`)
alias `/tikz/shape border uses incircle`

Determines if the border of a shape is constructed using the incircle. If no value is given `<boolean>` will take the default value `true`.

`/pgf/shape border rotate=<angle>` (no default, initially 0)
alias `/tikz/shape border rotate`

Rotates the border of a shape independently of the node contents, but in addition to any other transformations. If the shape border is not constructed using the incircle, the rotation will be rounded to the nearest integer multiple of 90 degrees when the shape is drawn.

Note that if the border of the shape is rotated, the compass point anchors, and ‘text box’ anchors (including `mid east`, `base west`, and so on), *do not rotate*, but the other anchors do:



```
\usetikzlibrary {shapes.geometric}
\ tikzset{every node/.style={shape=trapezium, draw, shape border uses incircle}}
\begin{tikzpicture}
\node at (0,0) (A) {A};
\node [shape border rotate=30] at (1.5,0) (B) {B};
\foreach \s/\t in
{left side/base east, bottom side/north, bottom left corner/base}{
\fill[red] (A.\s) circle(1.5pt) (B.\s) circle(1.5pt);
\fill[blue] (A.\t) circle(1.5pt) (B.\t) circle(1.5pt);
}
\end{tikzpicture}
```

Finally, a somewhat unfortunate side-effect of rotating shape borders is that the supporting shapes do not distinguish between `outer xsep` and `outer ysep`, and typically, the larger of the two values will be used.

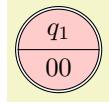
17.3 Multi-Part Nodes

Most nodes just have a single simple text label. However, nodes of a more complicated shape might be made up from several *node parts*. For example, in automata theory a so-called Moore state has a state name, drawn in the upper part of the state circle, and an output text, drawn in the lower part of the state circle. These two parts are quite independent. Similarly, a UML class shape would have a name part, a method part, and an attributes part. Different molecule shapes might use parts for the different atoms to be drawn at the different positions, and so on.

Both PGF and TikZ support such multipart nodes. On the lower level, PGF provides a system for specifying that a shape consists of several parts. On the TikZ level, you specify the different node parts by using the following command:

`\nodepart[<options>]{<part name>}`

This command can only be used inside the `<text>` argument of a `node` path operation. It works a little bit like a `\part` command in L^AT_EX. It will stop the typesetting of whatever node part was typeset until now and then start putting all following text into the node part named `<part name>` – until another `\partname` is encountered or until the node `<text>` ends. The `<options>` will be local to this part.



```
\usetikzlibrary {shapes.multipart}
\begin{tikzpicture}
  \node [circle split,draw,double,fill=red!20]
  {
    % No \nodepart has been used, yet. So, the following is put in the
    % ``text'' node part by default.
    $q_1$%
    \nodepart[lower] % Ok, end ``text'' part, start ``output'' part
    $00$%
  }; % output part ended.
\end{tikzpicture}
```

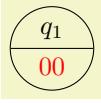
You will have to lookup which parts are defined by a shape.

The following styles influences node parts:

`/tikz/every <part name> node part`

(style, initially empty)

This style is installed at the beginning of every node part named *<part name>*.



```
\usetikzlibrary {shapes.multipart}
\tikz [every lower node part/.style={red}]
  \node [circle split,draw] {$q_1$ \nodepart[lower] $00$};
```

17.4 The Node Text

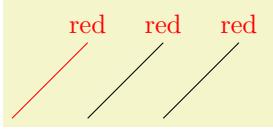
17.4.1 Text Parameters: Color and Opacity

The simplest option for the text in nodes is its color. Normally, this color is just the last color installed using `color=`, possibly inherited from another scope. However, it is possible to specifically set the color used for text using the following option:

`/tikz/text=<color>`

(no default)

Sets the color to be used for text labels. A `color=` option will immediately override this option.



```
\begin{tikzpicture}
  \draw[red] (0,0) -- +(1,1) node[above] {red};
  \draw[text=red] (1,0) -- +(1,1) node[above] {red};
  \draw (2,0) -- +(1,1) node[above,red] {red};
\end{tikzpicture}
```

Just like the color itself, you may also wish to set the opacity of the text only. For this, use the `text opacity` option, which is detailed in Section 23.

17.4.2 Text Parameters: Font

Next, you may wish to adjust the font used for the text. Naturally, you can just use a font command like `\small` or `\rm` at the beginning of a node. However, the following two options make it easier to set the font used in nodes on a general basis. Let us start with:

`/tikz/node font=`

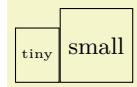
(no default)

This option sets the font used for all text used in a node.



```
\begin{tikzpicture}
  \draw[node font=\itshape] (1,0) -- +(1,1) node[above] {italic};
\end{tikzpicture}
```

Since the ** are executed at a very early stage in the construction of the node, the font selected using this command will also dictate the values of dimensions defined in terms of `em` or `ex`. For instance, when the `minimum height` of a node is `3em`, the actual height will be (at least) three times the line distance selected by the **:



```
\tikz \node [node font=\tiny, minimum height=3em, draw] {tiny};
\tikz \node [node font=\small, minimum height=3em, draw] {small};
```

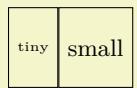
The other font command is:

`/tikz/font=` (no default)

Sets the font used for the text inside nodes. However, this font will *not* be installed when any of the dimensions of the node are being computed, so dimensions like `1em` will be with respect to the font used outside the node (usually the font that was in force when the picture started).

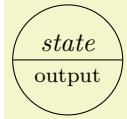
italic

```
\begin{tikzpicture}
  \node [font=\itshape] {italic};
\end{tikzpicture}
```



```
\tikz \node [font=\tiny, minimum height=3em, draw] {tiny};
\tikz \node [font=\small, minimum height=3em, draw] {small};
```

A useful example of how the `font` option can be used is the following:



```
\usetikzlibrary {shapes.multipart}
\tikz [every text node part/.style={font=\itshape},
       every lower node part/.style={font=\footnotesize}]
\node [circle split,draw] {state \nodepart[lower] output};
```

As can be seen, the font can be changed for each node part. This does *not* work with the `node font` command since, as the name suggests, this command can only be used to select the “overall” font for the node and this is done very early.

17.4.3 Text Parameters: Alignment and Width for Multi-Line Text

Normally, when a node is typeset, all the text you give in the braces is put in one long line (in an `\hbox`, to be precise) and the node will become as wide as necessary.

From time to time you may wish to create nodes that contain multiple lines of text. There are three different ways of achieving this:

1. Inside the node, you can put some standard environment that produces multi-line, aligned text. For instance, you can use a `{tabular}` inside a node:

upper left	upper right
lower left	lower right

```
\tikz \node [draw] {
  \begin{tabular}{cc}
    upper left & upper right \\
    lower left & lower right
  \end{tabular}
};
```

This approach offers the most flexibility in the sense that it allows you to use all of the alignment commands offered by your format of choice.

2. You use `\backslash\backslash` inside your node to mark the end of lines and then request TikZ to arrange these lines in some manner. This will only be done, however, if the `align` option has been given.

This is a demonstration.

```
\tikz[align=left] \node [draw] {This is a\backslash\backslash demonstration.};
```

```
This is a  
demonstration.
```

```
\tikz[align=center] \node[draw] {This is a\\demonstration.};
```

The `\\\` command takes an optional extra space as an argument in square brackets.

```
This is a  
demonstration text for  
alignments.
```

```
\tikz \node[fill=yellow!80!black,align=right]  
{This is a\\[-2pt] demonstration text for\\[tex] alignments.};
```

3. You can request that TikZ does an automatic line-breaking for you inside the node by specifying a fixed `text width` for the node. In this case, you can still use `\\\` to enforce a line-break. Note that when you specify a text width, the node will have this width, independently of whether the text actually “reaches the end” of the node.

Let us now first have a look at the `text width` command.

`/tikz/text width=<dimension>`

(no default)

This option will put the text of a node in a box of the given width (something akin to a `{minipage}` of this width, only portable across formats). If the node text is not as wide as `<dimension>`, it will nevertheless be put in a box of this width. If it is larger, line breaking will be done.

By default, when this option is given, a ragged right border will be used (`align=left`). This is sensible since, typically, these boxes are narrow and justifying the text looks ugly. You can, however, change the alignment using `align` or directly using commands like `\centering`.

```
This is a demon-  
stration text for  
showing how line  
breaking works.
```

```
\tikz \draw (0,0) node[fill=yellow!80!black,text width=3cm]  
{This is a demonstration text for showing how line breaking works.};
```

Setting `<dimension>` to an empty string causes the automatic line breaking to be disabled.

`/tikz/align=<alignment option>`

(no default)

This key is used to set up an alignment for multi-line text inside a node. If `text width` is set to some width (let us call this *alignment with line breaking*), the `align` key will setup the `\leftskip` and the `\rightskip` in such a way that the text is broken and aligned according to `<alignment option>`. If `text width` is not set (that is, set to the empty string; let us call this *alignment without line breaking*), then a different mechanism is used internally, namely the key `node halign header`, is set to an appropriate value. While this key, which is documented below, is not to be used by beginners, the net effect is simple: When `text width` is not set, you can use `\\\` to break lines and align them according to `<alignment option>` and the resulting node’s width will be minimal to encompass the resulting lines.

In detail, you can set `<alignment option>` to one of the following values:

align=left For alignment without line breaking, the different lines are simply aligned such that their left borders are below one another.

```
This is a  
demonstration text for  
alignments.
```

```
\tikz \node[fill=yellow!80!black,align=left]  
{This is a\\ demonstration text for\\ alignments.};
```

For alignment with line breaking, the same will happen; only the lines will now, additionally, be broken automatically:

This is a demonstration text for showing how line breaking works.

```
\tikz \node[fill=yellow!80!black,text width=3cm,align=left]
  {This is a demonstration text for showing how line breaking works.};
```

align=flush left For alignment without line breaking this option has exactly the same effect as `left`. However, for alignment with line breaking, there is a difference: While `left` uses the original plain TeX definition of a ragged right border, in which TeX will try to balance the right border as well as possible, `flush left` causes the right border to be ragged in the L^AT_EX-style, in which no balancing occurs. This looks ugly, but it may be useful for very narrow boxes and when you wish to avoid hyphenations.

This is a demonstration text for showing how line breaking works.

```
\tikz \node[fill=yellow!80!black,text width=3cm,align=flush left]
  {This is a demonstration text for showing how line breaking works.};
```

align=right Works like `left`, only for right alignment.

This is a demonstration text for alignments.

```
\tikz \node[fill=yellow!80!black,align=right]
  {This is a\\ demonstration text for\\ alignments.};
```

This is a demonstration text for showing how line breaking works.

```
\tikz \node[fill=yellow!80!black,text width=3cm,align=right]
  {This is a demonstration text for showing how line breaking works.};
```

align=flush right Works like `flush left`, only for right alignment.

This is a demonstration text for showing how line breaking works.

```
\tikz \node[fill=yellow!80!black,text width=3cm,align=flush right]
  {This is a demonstration text for showing how line breaking works.};
```

align=center Works like `left` or `right`, only for centered alignment.

This is a demonstration text for alignments.

```
\tikz \node[fill=yellow!80!black,align=center]
  {This is a\\ demonstration text for\\ alignments.};
```

This is a demonstration text for showing how line breaking works.

```
\tikz \node[fill=yellow!80!black,text width=3cm,align=center]
  {This is a demonstration text for showing how line breaking works.};
```

There is one annoying problem with the `center` alignment (but not with `flush center` and the other options): If you specify a large line width and the node text fits on a single line and is, in fact, much shorter than the specified `text width`, an underfull horizontal box will result. Unfortunately, this cannot be avoided, due to the way TeX works (more precisely, I have thought long and hard about this and have not been able to figure out a sensible way to avoid this). For this reason, TikZ switches off horizontal badness warnings inside boxes with `align=center`. Since this will also suppress some “wanted” warnings, there is also an option for switching the warnings on once more:

`/tikz/badness warnings for centered text=<true or false>` (no default, initially `false`)

If set to true, normal badness warnings will be issued for centered boxes. Note that you may get annoying warnings for perfectly normal boxes, namely whenever the box is very large and the contents is not long enough to fill the box sufficiently.

`align=flush center` Works like `flush left` or `flush right`, only for center alignment. Because of all the trouble that results from the `center` option in conjunction with narrow lines, I suggest picking this option rather than `center` unless you have longer text, in which case `center` will give the typographically better results.

This is a demonstration text for showing how line breaking works.

```
\tikz \node[fill=yellow!80!black, text width=3cm, align=flush center]
  {This is a demonstration text for showing how line breaking works.};
```

`align=justify` For alignment without line breaking, this has the same effect as `left`. For alignment with line breaking, this causes the text to be “justified”. Use this only with rather broad nodes.

This is a demonstration text for showing how line breaking works.

```
\tikz \node[fill=yellow!80!black, text width=3cm, align=justify]
  {This is a demonstration text for showing how line breaking works.};
```

In the above example, TeX complains (rightfully) about three very badly typeset lines. (For this manual I asked TeX to stop complaining by using `\hbadness=10000`, but this is a foul deed, indeed.)

`align=none` Disables all alignments and `\&` will not be redefined.

`/tikz/node halign header=<macro storing a header>` (no default, initially empty)

This is the key that is used by `align` internally for alignment without line breaking. Read the following only if you are familiar with the `\halign` command.

This key only has an effect if `text width` is empty, otherwise it is ignored. Furthermore, if `<macro storing a header>` is empty, then this key also has no effect. So, suppose `text width` is empty, but `<header>` is not. In this case the following happens:

When the node text is parsed, the command `\&` is redefined internally. This redefinition is done in such a way that the text from the start of the node to the first occurrence of `\&` is put in an `\hbox`. Then the text following `\&` up to the next `\&` is put in another `\hbox`. This goes on until the text between the last `\&` and the closing `}` is also put in an `\hbox`.

The `<macro storing a header>` should be a macro that contains some text suitable for use as a header for the `\halign` command. For instance, you might define

```
\def\myheader{\hfil\hfil##\hfil\cr}
\tikz [node halign header=\myheader] ...
```

You cannot just say `node halign header=\hfil\hfil#\hfil\cr` because this confuses TeX inside matrices, so this detour via a macro is needed.

Next, conceptually, all these boxes are recursively put inside an `\halign` command. Assuming that `<first>` is the first of the above boxes, the command `\halign{\<header> \box<first> \cr}` is used to create a new box, which we will call the `<previous box>`. Then, the following box is created, where `<second>` is the second input box: `\halign{\<header> \box<previous box> \cr \box<second>\cr}`. Let us call the resulting box the `<previous box>` once more. Then the next box that is created is `\halign{\<header> \box<previous box> \cr \box<third>\cr}`.

All of this means that if `<header>` is an `\halign` header like `\hfil#\hfil\cr`, then all boxes will be centered relative to one another. Similarly, a `<header>` of `\hfil#\cr` causes the text to be flushed right.

Note that this mechanism is not flexible enough to allow multiple columns inside `<header>`. You will have to use a `tabular` or a `matrix` in such cases.

One further note: Since the text of each line is placed in a box, settings will be local to each “line”. This is very similar to the way a cell in a `tabular` or a `matrix` behaves.

17.4.4 Text Parameters: Height and Depth of Text

In addition to changing the width of nodes, you can also change the height of nodes. This can be done in two ways: First, you can use the option `minimum height`, which ensures that the height of the whole node is at least the given height (this option is described in more detail later). Second, you can use the option `text height`, which sets the height of the text itself, more precisely, of the TeX text box of the text. Note that the `text height` typically is not the height of the shape’s box: In addition to the `text height`, an internal `inner sep` is added as extra space and the text depth is also taken into account.

I recommend using `minimum size` instead of `text height` except for special situations.

`/tikz/text height=<dimension>` (no default)

Sets the height of the text boxes in shapes. Thus, when you write something like `node {text}`, the `text` is first typeset, resulting in some box of a certain height. This height is then replaced by the height `text height`. The resulting box is then used to determine the size of the shape, which will typically be larger. When you write `text height=` without specifying anything, the “natural” size of the text box remains unchanged.



```
\tikz \node[draw] {y};  
\tikz \node[draw,text height=10pt] {y};
```

`/tikz/text depth=<dimension>` (no default)

This option works like `text height`, only for the depth of the text box. This option is mostly useful when you need to ensure a uniform depth of text boxes that need to be aligned.

17.5 Positioning Nodes

When you place a node at some coordinate, the node is centered on this coordinate by default. This is often undesirable and it would be better to have the node to the right or above the actual coordinate.

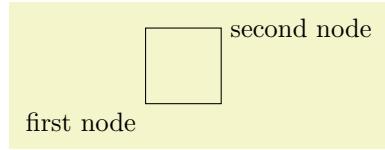
17.5.1 Positioning Nodes Using Anchors

PGF uses a so-called anchoring mechanism to give you a very fine control over the placement. The idea is simple: Imagine a node of rectangular shape of a certain size. PGF defines numerous anchor positions in the shape. For example the upper right corner is called, well, not “upper right anchor”, but the `north east` anchor of the shape. The center of the shape has an anchor called `center` on top of it, and so on. Here are some examples (a complete list is given in Section 17.2.2).



Now, when you place a node at a certain coordinate, you can ask TikZ to place the node shifted around in such a way that a certain anchor is at the coordinate. In the following example, we ask TikZ to shift the

first node such that its `north east` anchor is at coordinate $(0,0)$ and that the `west` anchor of the second node is at coordinate $(1,1)$.



```
\tikz \draw (0,0) node[anchor=north east] {first node}
rectangle (1,1) node[anchor=west] {second node};
```

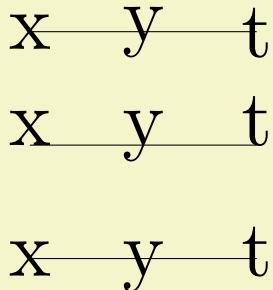
Since the default anchor is `center`, the default behaviour is to shift the node in such a way that it is centered on the current position.

`/tikz/anchor=<anchor name>` (no default)

Causes the node to be shifted such that its anchor `<anchor name>` lies on the current coordinate.

The only anchor that is present in all shapes is `center`. However, most shapes will at least define anchors in all “compass directions”. Furthermore, the standard shapes also define a `base` anchor, as well as `base west` and `base east`, for placing things on the baseline of the text.

The standard shapes also define a `mid` anchor (and `mid west` and `mid east`). This anchor is half the height of the character “x” above the base line. This anchor is useful for vertically centering multiple nodes that have different heights and depth. Here is an example:



```
\begin{tikzpicture}[scale=3,transform shape]
% First, center alignment -> wobbles
\draw[anchor=center] (0,1) node{x} -- (0.5,1) node{y} -- (1,1) node{t};
% Second, base alignment -> no wobble, but too high
\draw[anchor=base] (0,.5) node{x} -- (0.5,.5) node{y} -- (1,.5) node{t};
% Third, mid alignment
\draw[anchor=mid] (0,0) node{x} -- (0.5,0) node{y} -- (1,0) node{t};
\end{tikzpicture}
```

17.5.2 Basic Placement Options

Unfortunately, while perfectly logical, it is often rather counter-intuitive that in order to place a node *above* a given point, you need to specify the `south` anchor. For this reason, there are some useful options that allow you to select the standard anchors more intuitively:

`/tikz/above=<offset>` (default `0pt`)

Does the same as `anchor=south`. If the `<offset>` is specified, the node is additionally shifted upwards by the given `<offset>`.

above
● `\tikz \fill (0,0) circle (2pt) node[above] {above};`

above
● `\tikz \fill (0,0) circle (2pt) node[above=2pt] {above};`

`/tikz/below=⟨offset⟩` (default 0pt)

Similar to above.

`/tikz/left=⟨offset⟩` (default 0pt)

Similar to above.

`/tikz/right=⟨offset⟩` (default 0pt)

Similar to above.

`/tikz/above left` (no value)

Does the same as `anchor=south east`. Note that giving both `above` and `left` options does not have the same effect as `above left`, rather only the last `left` “wins”. Actually, this option also takes an `⟨offset⟩` parameter, but using this parameter without using the `positioning` library is deprecated. (The `positioning` library changes the meaning of this parameter to something more sensible.)

above left 

```
\tikz \fill (0,0) circle (2pt) node[above left] {above left};
```

`/tikz/above right` (no value)

Similar to `above left`.

above right 

```
\tikz \fill (0,0) circle (2pt) node[above right] {above right};
```

`/tikz/below left` (no value)

Similar to `above left`.

`/tikz/below right` (no value)

Similar to `above left`.

`/tikz/centered` (no value)

A shorthand for `anchor=center`.

17.5.3 Advanced Placement Options

While the standard placement options suffice for simple cases, the `positioning` library offers more convenient placement options.

TikZ Library `positioning`

```
\usetikzlibrary{positioning} % LATEX and plain TEX  
\usetikzlibrary[positioning] % ConTEX
```

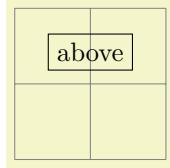
The library defines additional options for placing nodes conveniently. It also redefines the standard options like `above` so that they give you better control of node placement.

When this library is loaded, the options like `above` or `above left` behave differently.

`/tikz/above=⟨specification⟩` (default 0pt)

With the `positioning` library loaded, the `above` option does not take a simple `⟨dimension⟩` as its parameter. Rather, it can (also) take a more elaborate `⟨specification⟩` as parameter. This `⟨specification⟩` has the following general form: It starts with an optional `⟨shifting part⟩` and is followed by an optional `⟨of-part⟩`. Let us start with the `⟨shifting part⟩`, which can have three forms:

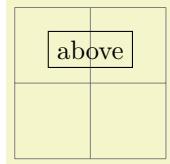
1. It can simply be a `⟨dimension⟩` (or a mathematical expression that evaluates to a dimension) like `2cm` or `3cm/2+4cm`. In this case, the following happens: the node’s anchor is set to `south` and the node is vertically shifted upwards by `⟨dimension⟩`.



```
\begin{tikzpicture}
  \draw[help lines] (0,0) grid (2,2);
  \node at (1,1) [above=2pt+3pt,draw] {above};
\end{tikzpicture}
```

This use of the `above` option is the same as if the `positioning` library were not loaded.

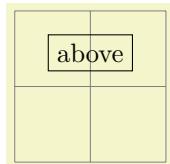
2. It can be a `(number)` (that is, any mathematical expression that does not include a unit like `pt` or `cm`). Examples are `2` or `3+sin(60)`. In this case, the anchor is also set to `south` and the node is vertically shifted by the vertical component of the coordinate `(0,<number>)`.



```
\begin{tikzpicture}
  \draw[help lines] (0,0) grid (2,2);
  \node at (1,1) [above=.2,draw] {above};
  % south border of the node is now 2mm above (1,1)
\end{tikzpicture}
```

3. It can be of the form `(number or dimension 1)` and `(number or dimension 2)`. This specification does not make particular sense for the `above` option, it is much more useful for options like `above left`. The reason it is allowed for the `above` option is that it is sometimes automatically used, as explained later.

The effect of this option is the following. First, the point `((number or dimension 2),(number or dimension 1))` is computed (note the inverted order), using the normal rules for evaluating such a coordinate, yielding some position. Then, the node is shifted by the vertical component of this point. The anchor is set to `south`.

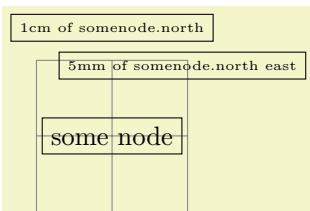


```
\usetikzlibrary {positioning}
\begin{tikzpicture}
  \draw[help lines] (0,0) grid (2,2);
  \node at (1,1) [above=.2 and 3mm,draw] {above};
  % south border of the node is also 2mm above (1,1)
\end{tikzpicture}
```

The `<shifting part>` can optionally be followed by a `<of-part>`, which has one of the following forms:

1. The `<of-part>` can be `of <coordinate>`, where `<coordinate>` is *not* in parentheses and it is *not* just a node name. An example would be `of somenode.north` or `of {2,3}`. In this case, the following happens: First, the node's `at` parameter is set to the `<coordinate>`. Second, the node is shifted according to the `<shift-part>`. Third, the anchor is set to `south`.

Here is a basic example:



```
\usetikzlibrary {positioning}
\begin{tikzpicture}[every node/.style=draw]
  \draw[help lines] (0,0) grid (2,2);
  \node (somenode) at (1,1) {some node};

  \node [above=1cm of somenode.north] {\tiny 1cm of somenode.north};
  \node [above=5mm of somenode.north east] {\tiny 5mm of somenode.north east};
\end{tikzpicture}
```

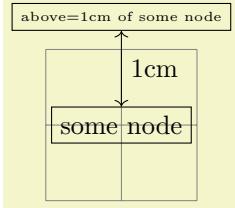
As can be seen the `above=5mm of somenode.north east` option does, indeed, place the node 5mm above the north east anchor of `somenode`. The same effect could have been achieved writing `above=5mm` followed by `at=(somenode.north east)`.

If the `<shifting-part>` is missing, the shift is not zero, but rather the value of the `node distance` key is used, see below.

2. The `<of-part>` can be `of <node name>`. An example would be `of somenode`. In this case, the following usually happens:

- The anchor is set to `south`.
- The node is shifted according to the `<shifting part>` or, if it is missing, according to the value of `node distance`.
- The node's `at` parameter is set to `<node name>.north`.

The net effect of all this is that the new node will be placed in such a way that the distance between its south border and `<node name>`'s north border is exactly the given distance.



```
\usetikzlibrary {positioning}
\begin{tikzpicture}[every node/.style=draw]
  \draw[help lines] (0,0) grid (2,2);
  \node (some node) at (1,1) {some node};

  \node (other node) [above=1cm of some node] {\tiny above=1cm of some node};

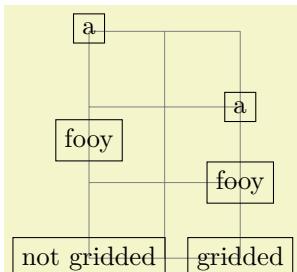
  \draw [<->] (some node.north) -- (other node.south)
    node [midway,right,draw=none] {1cm};
\end{tikzpicture}
```

It is possible to change the behaviour of this `<specification>` rather drastically, using the following key:

`/tikz/on grid=<boolean>` (no default, initially `false`)

When this key is set to `true`, an `<of-part>` of the current form behaves differently: The anchors set for the current node as well as the anchor used for the other `<node name>` are set to `center`.

This has the following effect: When you say `above=1cm of somenode` with `on grid` set to true, the new node will be placed in such a way that its center is 1cm above the center of `somenode`. Repeatedly placing nodes in this way will result in nodes that are centered on "grid coordinate", hence the name of the option.



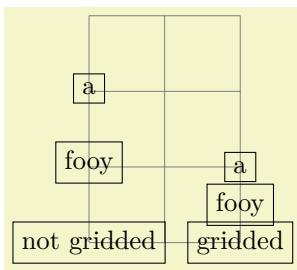
```
\usetikzlibrary {positioning}
\begin{tikzpicture}[every node/.style=draw]
  \draw[help lines] (0,0) grid (2,3);

  % Not gridded
  \node (a1) at (0,0) {not gridded};
  \node (b1) [above=1cm of a1] {fooy};
  \node (c1) [above=1cm of b1] {a};

  % gridded
  \node (a2) at (2,0) {gridded};
  \node (b2) [on grid,above=1cm of a2] {fooy};
  \node (c2) [on grid,above=1cm of b2] {a};
\end{tikzpicture}
```

`/tikz/node distance=<shifting part>` (no default, initially `1cm` and `1cm`)

The value of this key is used as `<shifting part>` is used if and only if a `<of-part>` is present, but no `<shifting part>`.



```
\usetikzlibrary {positioning}
\begin{tikzpicture}[every node/.style=draw,node distance=5mm]
  \draw[help lines] (0,0) grid (2,3);

  % Not gridded
  \node (a1) at (0,0) {not gridded};
  \node (b1) [above=of a1] {fooy};
  \node (c1) [above=of b1] {a};

  % gridded
  \begin{scope}[on grid]
    \node (a2) at (2,0) {gridded};
    \node (b2) [above=of a2] {fooy};
    \node (c2) [above=of b2] {a};
  \end{scope}
\end{tikzpicture}
```

`/tikz/below=<specification>`

(no default)

This key is redefined in the same manner as `above`.

`/tikz/left=<specification>`

(no default)

This key is redefined in the same manner as `above`, only all vertical shifts are replaced by horizontal shifts.

`/tikz/right=<specification>`

(no default)

This key is redefined in the same manner as `left`.

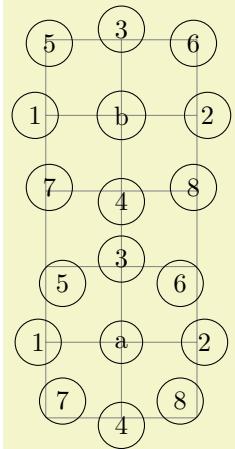
`/tikz/above left=<specification>`

(no default)

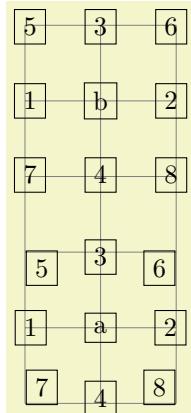
This key is also redefined in a manner similar to the above, but behaviour of the `<shifting part>` is more complicated:

- When the `<shifting part>` is of the form `<number or dimension> and <number or dimension>`, it has (essentially) the effect of shifting the node vertically upwards by the first `<number or dimension>` and to the left by the second. To be more precise, the coordinate `(<second number or dimension>, <first number or dimension>)` is computed and then the node is shifted vertically by the *y*-part of the resulting coordinate and horizontally by the negated *x*-part of the result. (This is exactly what you expect, except possibly when you have used the `x` and `y` options to modify the `xy`-coordinate system so that the unit vectors no longer point in the expected directions.)
- When the `<shifting part>` is of the form `<number or dimension>`, the node is shifted by this `<number or dimension>` in the direction of 135°. This means that there is a difference between a `<shifting part>` of 1cm and of 1cm and 1cm: In the second case, the node is shifted by 1cm upward and 1cm to the left; in the first case it is shifted by $\frac{1}{2}\sqrt{2}$ cm upward and by the same amount to the left. A more mathematical way of phrasing this is the following: A plain `<dimension>` is measured in the l_2 -norm, while a `<dimension> and <dimension>` is measured in the l_1 -norm.

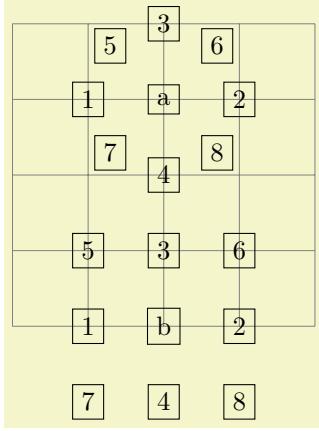
The following example should help to illustrate the difference:



```
\usetikzlibrary {positioning}
\begin{tikzpicture}[every node/.style={draw,circle}]
\draw[help lines] (0,0) grid (2,5);
\begin{scope}[node distance=5mm and 5mm]
\node (b) at (1,4) {b};
\node [left=of b] {1}; \node [right=of b] {2};
\node [above=of b] {3}; \node [below=of b] {4};
\node [above left=of b] {5}; \node [above right=of b] {6};
\node [below left=of b] {7}; \node [below right=of b] {8};
\end{scope}
\begin{scope}[node distance=5mm]
\node (a) at (1,1) {a};
\node [left=of a] {1}; \node [right=of a] {2};
\node [above=of a] {3}; \node [below=of a] {4};
\node [above left=of a] {5}; \node [above right=of a] {6};
\node [below left=of a] {7}; \node [below right=of a] {8};
\end{scope}
\end{tikzpicture}
```



```
\usetikzlibrary {positioning}
\begin{tikzpicture}[every node/.style={draw,rectangle}]
\draw[help lines] (0,0) grid (2,5);
\begin{scope}[node distance=5mm and 5mm]
\node (b) at (1,4) {b};
\node [left=of b] {1}; \node [right=of b] {2};
\node [above=of b] {3}; \node [below=of b] {4};
\node [above left=of b] {5}; \node [above right=of b] {6};
\node [below left=of b] {7}; \node [below right=of b] {8};
\end{scope}
\begin{scope}[node distance=5mm]
\node (a) at (1,1) {a};
\node [left=of a] {1}; \node [right=of a] {2};
\node [above=of a] {3}; \node [below=of a] {4};
\node [above left=of a] {5}; \node [above right=of a] {6};
\node [below left=of a] {7}; \node [below right=of a] {8};
\end{scope}
\end{tikzpicture}
```



```
\usetikzlibrary {positioning}
\begin{tikzpicture}[every node/.style={draw,rectangle},on grid]
  \draw[help lines] (0,0) grid (4,4);
  \begin{scope}[node distance=1]
    \node (a) at (2,3) {a};
    \node [left=of a] {1}; \node [right=of a] {2};
    \node [above=of a] {3}; \node [below=of a] {4};
    \node [above left=of a] {5}; \node [above right=of a] {6};
    \node [below left=of a] {7}; \node [below right=of a] {8};
  \end{scope}
  \begin{scope}[node distance=1 and 1]
    \node (b) at (2,0) {b};
    \node [left=of b] {1}; \node [right=of b] {2};
    \node [above=of b] {3}; \node [below=of b] {4};
    \node [above left=of b] {5}; \node [above right=of b] {6};
    \node [below left=of b] {7}; \node [below right=of b] {8};
  \end{scope}
\end{tikzpicture}
```

/tikz/below left=⟨specification⟩

(no default)

Works similar to `above left`.

/tikz/above right=⟨specification⟩

(no default)

Works similar to `above left`.

/tikz/below right=⟨specification⟩

(no default)

Works similar to `above left`.

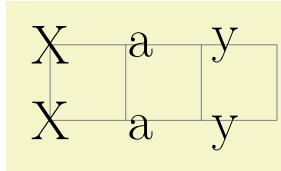
The `positioning` package also introduces the following new placement keys:

/tikz/base left=⟨specification⟩

(no default)

This key works like the `left` key, only instead of the `east` anchor, the `base east` anchor is used and, when the second form of an `⟨of-part⟩` is used, the corresponding `base west` anchor.

This key is useful for chaining together nodes so that their base lines are aligned.



```
\usetikzlibrary {positioning}
\begin{tikzpicture}[node distance=1ex]
  \draw[help lines] (0,0) grid (3,1);
  \huge
  \node (X) at (0,1) {X};
  \node (a) [right=of X] {a};
  \node (y) [right=of a] {y};

  \node (X) at (0,0) {X};
  \node (a) [base right=of X] {a};
  \node (y) [base right=of a] {y};
\end{tikzpicture}
```

/tikz/base right=⟨specification⟩

(no default)

Works like `base left`.

/tikz/mid left=⟨specification⟩

(no default)

Works like `base left`, but with `mid east` and `mid west` anchors instead of `base east` and `base west`.

/tikz/mid right=⟨specification⟩

(no default)

Works like `mid left`.

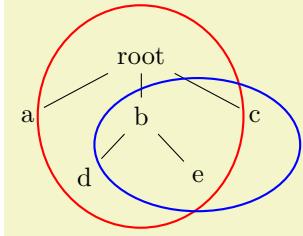
17.5.4 Advanced Arrangements of Nodes

The simple `above` and `right` options may not always suffice for arranging a large number of nodes. For such situations TikZ offers libraries that make positioning easier: The `graphdrawing` library and the `matrix` library. These libraries for positioning nodes are described in two separate Sections 20 and 27.

17.6 Fitting Nodes to a Set of Coordinates

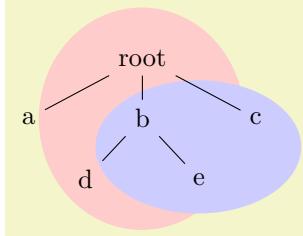
It is sometimes desirable that the size and position of a node is not given using anchors and size parameters, rather one would sometimes have a box be placed and be sized such that it “is just large enough to contain this, that, and that point”. This situation typically arises when a picture has been drawn and, afterwards, parts of the picture are supposed to be encircled or highlighted.

In this situation the `fit` option from the `fit` library is useful, see Section 55 for the details. The idea is that you may give the `fit` option to a node. The `fit` option expects a list of coordinates (one after the other without commas) as its parameter. The effect will be that the node’s text area has exactly the necessary size so that it contains all the given coordinates. Here is an example:



```
\node[draw=red,inner sep=0pt,thick,ellipse,fit=(root) (b) (d) (e)] {};
\node[draw=blue,inner sep=0pt,thick,ellipse,fit=(b) (c) (e)] {};
\end{tikzpicture}
```

If you want to fill the fitted node you will usually have to place it on a background layer.



```
\usetikzlibrary {backgrounds,fit,shapes.geometric}
\begin{tikzpicture}[level distance=8mm]
\node (root) {root}
child { node (a) {a} }
child { node (b) {b}
    child { node (d) {d} }
    child { node (e) {e} } }
child { node (c) {c} };

\begin{scope}[on background layer]
\node[fill=red!20,inner sep=0pt,ellipse,fit=(root) (b) (d) (e)] {};
\node[fill=blue!20,inner sep=0pt,ellipse,fit=(b) (c) (e)] {};
\end{scope}
\end{tikzpicture}
```

17.7 Transformations

It is possible to transform nodes, but, by default, transformations do not apply to nodes. The reason is that you usually do *not* want your text to be scaled or rotated even if the main graphic is transformed. Scaling text is evil, rotating slightly less so.

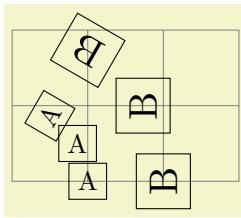
However, sometimes you *do* wish to transform a node, for example, it certainly sometimes makes sense to rotate a node by 90 degrees. There are two ways to achieve this:

1. You can use the following option:

`/tikz/transform shape` (no value)

Causes the current “external” transformation matrix to be applied to the shape. For example, if you said `\tikz[scale=3]` and then say `node[transform shape] {X}`, you will get a “huge” X in your graphic.

2. You can give transformation options *inside* the option list of the node. *These* transformations always apply to the node.

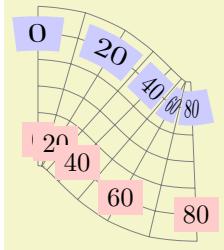


```
\usepgfmodule {nonlineartransformations}\usetikzlibrary {curvilinear}
\begin{tikzpicture}[every node/.style={draw}]
\draw[help lines] (0,0) grid (3,2);
\draw
(1,0) node[A]
(2,0) node[rotate=90,scale=1.5] {B};
\draw[rotate=30] (1,0) node[A]
(2,0) node[rotate=90,scale=1.5] {B};
\draw[rotate=60] (1,0) node[transform shape] {A}
(2,0) node[transform shape,rotate=90,scale=1.5] {B};
\end{tikzpicture}
```

Even though TikZ currently does not allow you to configure so-called *nonlinear transformations*, see Section ??, there is an option that influences how nodes are transformed when nonlinear transformations are in force:

`/tikz/transform shape nonlinear=<true or false>` (no default, initially `false`)

When set to true, TikZ will try to apply any current nonlinear transformation also to nodes. Typically, for the text in nodes this is not possible in general, in such cases a linear approximation of the nonlinear transformation is used. For more details, see Section ??.



```
\usepgfmodule{nonlineartransformations}\usetikzlibrary{curvilinear}
\begin{tikzpicture}
% Install a nonlinear transformation:
\pgfsetcurvilinearbeziercurve
{\pgfpoint{0mm}{20mm}}
{\pgfpoint{10mm}{20mm}}
{\pgfpoint{10mm}{10mm}}
{\pgfpoint{20mm}{10mm}}
\pgftransformnonlinear{\pgfpointcurvilinearbezierorthogonal{\pgf@x\pgf@y}%
}

% Draw something:
\draw [help lines] (0,-30pt) grid [step=10pt] (80pt,30pt);

\foreach \x in {0,20,...,80}
\node [fill=red!20] at (\x pt, -20pt) {\x};

\foreach \x in {0,20,...,80}
\node [fill=blue!20, transform shape nonlinear] at (\x pt, 20pt) {\x};
\end{tikzpicture}
```

17.8 Placing Nodes on a Line or Curve Explicitly

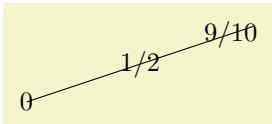
Until now, we always placed node on a coordinate that is mentioned in the path. Often, however, we wish to place nodes on “the middle” of a line and we do not wish to compute these coordinates “by hand”. To facilitate such placements, TikZ allows you to specify that a certain node should be somewhere “on” a line. There are two ways of specifying this: Either explicitly by using the `pos` option or implicitly by placing the node “inside” a path operation. These two ways are described in the following.

`/tikz/pos=<fraction>` (no default)

When this option is given, the node is not anchored on the last coordinate. Rather, it is anchored on some point on the line from the previous coordinate to the current point. The `<fraction>` dictates how “far” on the line the point should be. A `<fraction>` of 0 is the previous coordinate, 1 is the current one, everything else is in between. In particular, 0.5 is the middle.

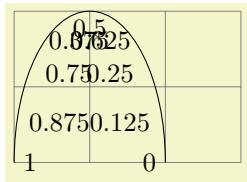
Now, what is “the previous line”? This depends on the previous path construction operation.

In the simplest case, the previous path operation was a “line-to” operation, that is, a `--<coordinate>` operation:



```
\tikz \draw (0,0) -- (3,1)
node[pos=0]{0} node[pos=0.5]{1/2} node[pos=0.9]{9/10};
```

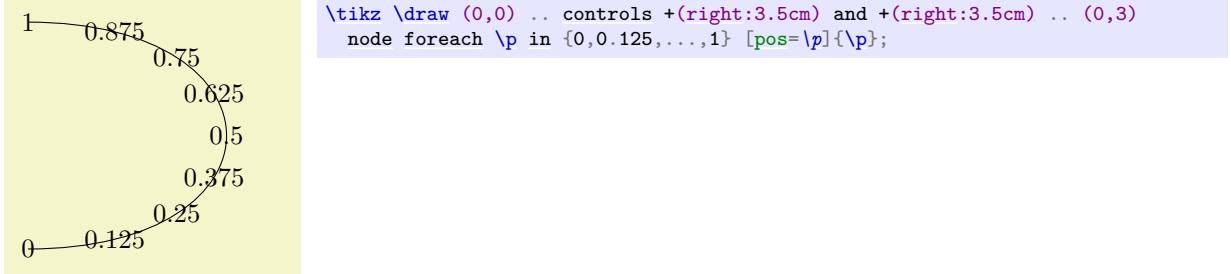
For the `arc` operation, the position is simply the corresponding position on the arc:



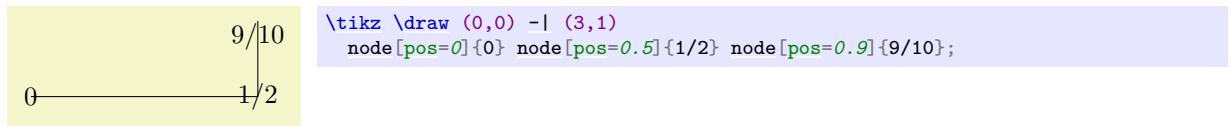
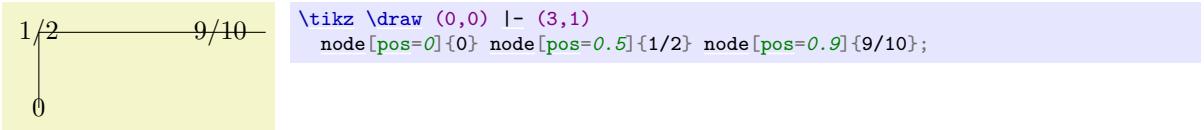
```
\tikz {
\draw [help lines] (0,0) grid (3,2);
\draw (2,0) arc [x radius=1, y radius=2, start angle=0, end angle=180]
node foreach \t in {0,0.125,...,1} [pos=\t,auto] {\t};
}
```

The next case is the curve-to operation (the `..` operation). In this case, the “middle” of the curve, that is, the position 0.5 is not necessarily the point at the exact half distance on the line. Rather, it is some

point at “time” 0.5 of a point traveling from the start of the curve, where it is at time 0, to the end of the curve, which it reaches at time 0.5. The “speed” of the point depends on the length of the support vectors (the vectors that connect the start and end points to the control points). The exact math is a bit complicated (depending on your point of view, of course); you may wish to consult a good book on computer graphics and Bézier curves if you are intrigued.



Another interesting case are the horizontal/vertical line-to operations `|-` and `-|`. For them, the position (or time) 0.5 is exactly the corner point.



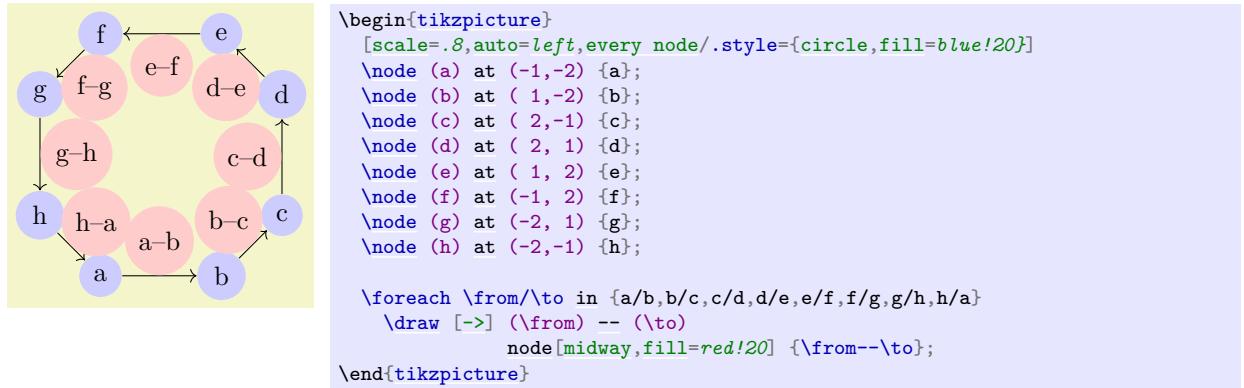
For all other path construction operations, *the position placement does not work*, currently.

/tikz/auto=<direction>

(default is scope’s setting)

This option causes an anchor positions to be calculated automatically according to the following rule. Consider a line between to points. If the `<direction>` is `left`, then the anchor is chosen such that the node is to the left of this line. If the `<direction>` is `right`, then the node is to the right of this line. Leaving out `<direction>` causes automatic placement to be enabled with the last value of `left` or `right` used. A `<direction>` of `false` disables automatic placement. This happens also whenever an anchor is given explicitly by the `anchor` option or by one of the `above`, `below`, etc. options.

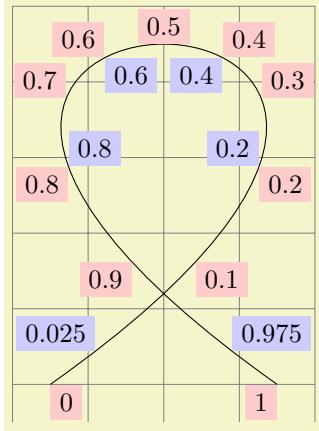
This option only has an effect for nodes that are placed on lines or curves.



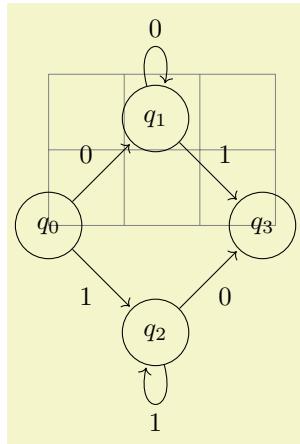
/tikz/swap

(no value)

This option exchanges the roles of `left` and `right` in automatic placement. That is, if `left` is the current `auto` placement, `right` is set instead and the other way round.



```
\usetikzlibrary {automata}
\begin{tikzpicture}[auto]
  \draw[help lines,use as bounding box] (0,-.5) grid (4,5);
  \draw (0.5,0) .. controls (9,6) and (-5,6) .. (3.5,0)
    node foreach \pos in {0,0.1,0.2,0.3,0.4,0.5,0.6,0.7,0.8,0.9,1}
      [pos=\pos,swap,fill=red!20] {\pos}
    node foreach \pos in {0.025,0.2,0.4,0.6,0.8,0.975}
      [pos=\pos,fill=blue!20] {\pos};
\end{tikzpicture}
```



```
\usetikzlibrary {automata}
\begin{tikzpicture}[shorten >=1pt,node distance=2cm,auto]
  \draw[help lines] (0,0) grid (3,2);
  \node[state] (q_0)          {$q_0$};
  \node[state] (q_1) [above right of=q_0] {$q_1$};
  \node[state] (q_2) [below right of=q_0] {$q_2$};
  \node[state] (q_3) [below right of=q_1] {$q_3$};
  \path[->] (q_0) edge      node {0} (q_1)
             edge      node {1} (q_2)
             edge      node {1} (q_3);
  \path[->] (q_1) edge      node {0} (q_0)
             edge [swap] node {1} (q_2)
             edge [loop above] node {0} ();
  \path[->] (q_2) edge      node {1} (q_0)
             edge [swap] node {0} (q_3)
             edge [loop below] node {1} ();
\end{tikzpicture}
```

/tikz/

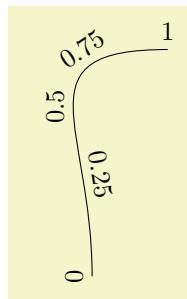
(no value)

This is a very short alias for `swap`.

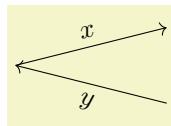
/tikz/sloped

(no value)

This option causes the node to be rotated such that a horizontal line becomes a tangent to the curve. The rotation is normally done in such a way that text is never “upside down”. To get upside-down text, use can use `[rotate=180` or `[allow upside down]`, see below.



```
\tikz \draw (0,0) .. controls +(up:2cm) and +(left:2cm) .. (1,3)
  node foreach \p in {0,0.25,...,1} [sloped,above, pos=\p]{\p};
```

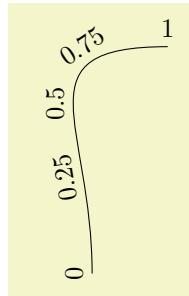


```
\begin{tikzpicture}[->]
  \draw (0,0) -- (2,0.5) node[midway,sloped,above] {$x$};
  \draw (2,-.5) -- (0,0) node[midway,sloped,below] {$y$};
\end{tikzpicture}
```

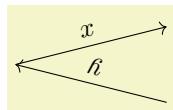
`/tikz/allow upside down=<boolean>`

(default `true`, initially `false`)

If set to `true`, TikZ will not “righten” upside down text.



```
\tikz [allow upside down]
  \draw (0,0) .. controls +(up:2cm) and +(left:2cm) .. (1,1)
    node foreach \p in {0,0.25,...,1} [sloped,above,pos=\p]{\p};
```



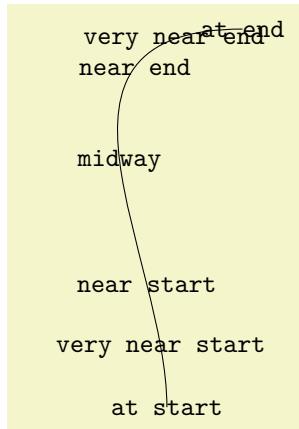
```
\begin{tikzpicture}[->,allow upside down]
  \draw (0,0) -- (2,0.5) node[midway,sloped,above] {$x$};
  \draw (2,-.5) -- (0,0) node[midway,sloped,below] {$y$};
\end{tikzpicture}
```

There exist styles for specifying positions a bit less “technically”:

`/tikz/midway`

(style, no value)

This has the same effect as `pos=0.5`.



```
\tikz \draw (0,0) .. controls +(up:2cm) and +(left:3cm) .. (1,1)
  node[at end] {\texttt{at end}}
  node[very near end] {\texttt{very near end}}
  node[near end] {\texttt{near end}}
  node[midway] {\texttt{midway}}
  node[near start] {\texttt{near start}}
  node[very near start] {\texttt{very near start}}
  node[at start] {\texttt{at start}};
```

`/tikz/near start`

(style, no value)

Set to `pos=0.25`.

`/tikz/near end`

(style, no value)

Set to `pos=0.75`.

`/tikz/very near start`

(style, no value)

Set to `pos=0.125`.

`/tikz/very near end`

(style, no value)

Set to `pos=0.875`.

`/tikz/at start`

(style, no value)

Set to `pos=0`.

`/tikz/at end`

(style, no value)

Set to `pos=1`.

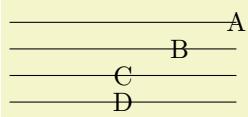
17.9 Placing Nodes on a Line or Curve Implicitly

When you wish to place a node on the line $(0,0) \text{ -- } (1,1)$, it is natural to specify the node not following the $(1,1)$, but “somewhere in the middle”. This is, indeed, possible and you can write $(0,0) \text{ -- node}\{a\} (1,1)$ to place a node midway between $(0,0)$ and $(1,1)$.

What happens is the following: The syntax of the line-to path operation is actually $\text{-- node}\langle\text{node specification}\rangle\langle\text{coordinate}\rangle$. (It is even possible to give multiple nodes in this way.) When the optional `node` is encountered, that is, when the `--` is directly followed by `node`, then the specification(s) are read and “stored away”. Then, after the `\langle coordinate\rangle` has finally been reached, they are inserted again, but with the `pos` option set.

There are two things to note about this: When a node specification is “stored”, its catcodes become fixed. This means that you cannot use overly complicated verbatim text in them. If you really need, say, a verbatim text, you will have to put it in a normal node following the coordinate and add the `pos` option.

Second, which `pos` is chosen for the node? The position is inherited from the surrounding scope. However, this holds only for nodes specified in this implicit way. Thus, if you add the option `[near end]` to a scope, this does not mean that *all* nodes given in this scope will be put on near the end of lines. Only the nodes for which an implicit `pos` is added will be placed near the end. Typically, this is what you want. Here are some examples that should make this clearer:



```
\begin{tikzpicture}[near end]
  \draw (0cm,4em) -- (3cm,4em) node{A};
  \draw (0cm,3em) -- node{B} (3cm,3em);
  \draw (0cm,2em) -- node[midway]{C} (3cm,2em);
  \draw (0cm,1em) -- (3cm,1em) node[midway]{D};
\end{tikzpicture}
```

Like the line-to operation, the curve-to operation `..` also allows you to specify nodes “inside” the operation. After both the first `..` and also after the second `..` you can place node specifications. Like for the `--` operation, these will be collected and then reinserted after the operation with the `pos` option set.

17.10 The Label and Pin Options

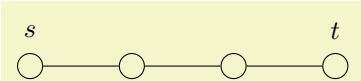
17.10.1 Overview

In addition to the `node` path operation, the two options `label` and `pin` can be used to “add a node next to another node”. As an example, suppose we want to draw a graph in which the nodes are small circles:



```
\usetikzlibrary {positioning}
\tikz [circle] {
  \node [draw] (s) {};
  \node [draw] (a) [right=of s] {} edge (s);
  \node [draw] (b) [right=of a] {} edge (a);
  \node [draw] (t) [right=of b] {} edge (b);
}
```

Now, in the above example, suppose we wish to indicate that the first node is the start node and the last node is the target node. We could write `\node (s) {s};`, but this would enlarge the first node. Rather, we want the “`s`” to be placed next to the node. For this, we need to create *another* node, but next to the existing node. The `label` and `pin` option allow us to do exactly this without having to use the cumbersome `node` syntax:



```
\usetikzlibrary {positioning}
\tikz [circle] {
  \node [draw] (s) [label=$s$] {};
  \node [draw] (a) [right=of s] {} edge (s);
  \node [draw] (b) [right=of a] {} edge (a);
  \node [draw] (t) [right=of b, label=$t$] {} edge (b);
}
```

17.10.2 The Label Option

`/tikz/label=[<options>] <angle>:<text>`

(no default)

When this option is given to a `node` operation, it causes *another* node to be added to the path after the current node has been finished. This extra node will have the text `<text>`. It is placed, in principle, in the direction `<angle>` relative to the main node, but the exact rules are a bit complex. Suppose the `node` currently under construction is called `main node` and let us call the label node `label node`. Then the following happens:

1. The `<angle>` is used to determine a position on the border of the `main node`. If the `<angle>` is missing, the value of the following key is used instead:

`/tikz/label position=<angle>`

(no default, initially `above`)

Sets the default position for labels.

The `<angle>` determines the position on the border of the shape in two different ways. Normally, the border position is given by `main node.<angle>`. This means that the `<angle>` can either be a number like 0 or -340, but it can also be an anchor like `north`. Additionally, the special angles `above`, `below`, `left`, `right`, `above left`, and so on are automatically replaced by the corresponding angles 90, 270, 180, 0, 135, and so on.

A special case arises when the following key is set:

`/tikz/absolute=<true or false>`

(default `true`)

When this key is set, the `<angle>` is interpreted differently: We still use a point on the border of the `main node`, but the angle is measured “absolutely”, that is, an angle of 0 refers to the point on the border that lies on a straight line from the `main node`’s center to the right (relative to the paper, not relative to the local coordinate system of either the node or the scope).

The difference can be seen in the following example:



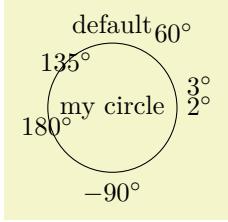
```
\tikz [rotate=-80,every label/.style={draw,red}]
\node [transform shape,rectangle,draw,label=right:label] {main node};
```



```
\tikz [rotate=-80,every label/.style={draw,red},absolute]
\node [transform shape,rectangle,draw,label=right:label] {main node};
```

2. Then, an anchor point for the `label node` is computed. It is determined in such a way that the `label node` will “face away” from the border of the `main node`. The anchor that is chosen depends on the position of the border point that is chosen and its position relative to the center of the `main node` and on whether the `transform shape` option is set. In detail, when the computed border point is at 0°, the anchor `west` will be used. Similarly, when the border point is at 90°, the anchor `south` will be used, and so on for 180° and 270°.

For angles between these “major” angles, like 30° or 110°, combined anchors, like `south west` for 30° or `south east` for 110°, are used. However, for angles close to the major angles, (differing by up to 2° from the major angle), the anchor for the major angle is used. Thus, a label at a border point for 2° will have the anchor `west`, while a label for 3° will have the anchor `south west`, resulting in a “jump” of the anchor. You can set the anchor “by hand” using the `anchor` key or indirect keys like `left`.



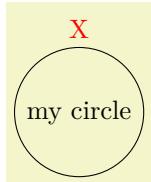
```
\tikz
\node [circle, draw,
      label=default,
      label=60:$60^\circ\circ\circ\circ$,
      label=below:$-90^\circ\circ\circ\circ$,
      label=3:$3^\circ\circ\circ$,
      label=2:$2^\circ\circ\circ$,
      label={[below]180:$180^\circ\circ\circ\circ$},
      label={[centered]135:$135^\circ\circ\circ$}] {my circle};
```

3. One $\langle angle \rangle$ is special: If you set the $\langle angle \rangle$ to `center`, then the label will be placed on the center of the main node. This is mainly useful for adding a label text to an existing node, especially if it has been rotated.

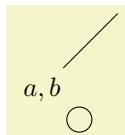


```
\tikz \node [transform shape,rotate=90,
           rectangle,draw,label={[red]center:R}] {main node};
```

You can pass $\langle options \rangle$ to the node `label` node. For this, you provide the options in square brackets before the $\langle angle \rangle$. If you do so, you need to add braces around the whole argument of the `label` option and this is also the case if you have brackets or commas or semicolons or anything special in the `text`.



```
\tikz \node [circle,draw,label={[red]above:X}] {my circle};
```



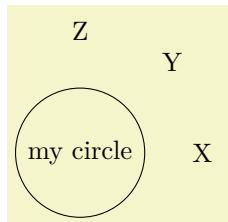
```
\begin{tikzpicture}
\node [circle,draw,label={[name=label_node]above left:$a,b$}] {};
\draw (label_node) -- +(1,1);
\end{tikzpicture}
```

If you provide multiple `label` options, then multiple extra label nodes are added in the order they are given.

The following styles influence how labels are drawn:

`/tikz/label distance=(distance)` (no default, initially 0pt)

The $\langle distance \rangle$ is additionally inserted between the main node and the label node.



```
\tikz [label distance=5mm]
\node [circle,draw,label=right:X,
       label=above right:Y,
       label=above:Z] {my circle};
```

`/tikz/every label` (style, initially empty)

This style is used in every node created by the `label` option. The default is `draw=none,fill=none`.

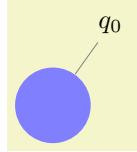
See Section 17.10.4 for an easier syntax for specifying nodes.

17.10.3 The Pin Option

`/tikz/pin=[⟨options⟩]⟨angle⟩:⟨text⟩`

(no default)

This option is quite similar to the `label` option, but there is one difference: In addition to adding an extra node to the picture, it also adds an edge from this node to the main node. This causes the node to look like a pin that has been added to the main node:



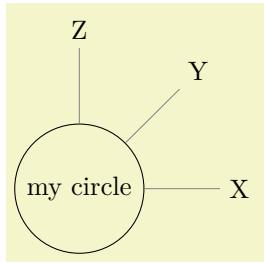
```
\tikz \node [circle,fill=blue!50,minimum size=1cm,pin=60:$q_0$] {};
```

The meaning of the `⟨options⟩` and the `⟨angle⟩` and the `⟨text⟩` is exactly the same as for the `node` option. Only, the options and styles the influence the way pins look are different:

`/tikz/pin distance=⟨distance⟩`

(no default, initially 3ex)

This `⟨distance⟩` is used instead of the `label distance` for the distance between the main node and the label node.



```
\tikz[pin distance=1cm]
\node [circle,draw,pin=right:X,
pin=above right:Y,
pin=above:Z] {my circle};
```

`/tikz/every pin`

(style, initially `draw=none,fill=None`)

This style is used in every node created by the `pin` option.

`/tikz/pin position=⟨angle⟩`

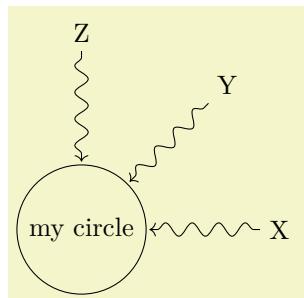
(no default, initially `above`)

The default pin position. Works like `label position`.

`/tikz/every pin edge`

(style, initially `help lines`)

This style is used in every edge created by the `pin` options.

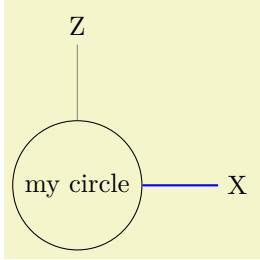


```
\usetikzlibrary {decorations.pathmorphing}
\tikz [pin distance=15mm,
every pin edge/.style={<-,shorten <=1pt,decorate,
decoration={snake,pre length=4pt}}]
\node [circle,draw,pin=right:X,
pin=above right:Y,
pin=above:Z] {my circle};
```

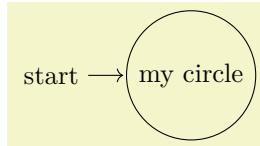
`/tikz/pin edge=⟨options⟩`

(no default, initially empty)

This option can be used to set the options that are to be used in the edge created by the `pin` option.



```
\tikz [pin distance=10mm]
\node [circle,draw,pin={[pin edge={blue,thick}]right:X},
pin=above:Z] {my circle};
```



```
\tikz [every pin edge/.style={},
initial/.style={pin={[pin distance=5mm,
pin edge={<-,shorten <=1pt}]left:start}}]
\node [circle,draw,initial] {my circle};
```

17.10.4 The Quotes Syntax

The `label` and `pin` options provide a syntax for creating nodes next to existing nodes, but this syntax is often a bit too verbose. By including the following library, you get access to an even more concise syntax:

TikZ Library `quotes`

```
\usetikzlibrary{quotes} % LATEX and plain TEX
\usetikzlibrary[quotes] % ConTEXt
```

Enables the quotes syntax for labels, pins, edge nodes, and pic texts.

Let us start with the basics of what this library does: Once loaded, inside the options of a `node` command, instead of the usual $\langle key \rangle = \langle value \rangle$ pairs, you may also provide strings of the following form (the actual syntax slightly more general, see the detailed descriptions later on):

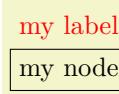
`"⟨text⟩"⟨options⟩`

The $\langle options \rangle$ must be surrounded in curly braces when they contain a comma, otherwise the curly braces are optional. The $\langle options \rangle$ may be preceded by an optional space.

When a $\langle string \rangle$ of the above form is encountered inside the options of a `node`, then it is internally transformed to

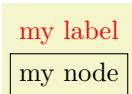
`label={[⟨options⟩]}⟨text⟩`

Let us have a look at an example:



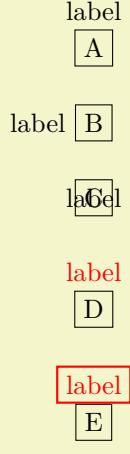
```
\usetikzlibrary {quotes}
\tikz \node ["my label" red, draw] {my node};
```

The above has the same effect as the following:



```
\tikz \node [label={{red}my label}, draw] {my node};
```

Here are further examples, one where no $\langle options \rangle$ are added to the `label`, one where a position is specified, and examples with more complicated options in curly braces:



```
\usetikzlibrary {quotes}
\begin{tikzpicture}
  \matrix [row sep=5mm] {
    \node [draw, "label"] {A}; \\
    \node [draw, "label" left] {B}; \\
    \node [draw, "label" centered] {C}; \\
    \node [draw, "label" color=red] {D}; \\
    \node [draw, "label" {red,draw,thick}] {E}; \\
  };
\end{tikzpicture}
```

Let us now have a more detailed look at what which commands this library provides:

`/tikz/quotes mean label`

(no value)

When this option is used (which is the default when this library is loaded), then, as described above, inside the options of a node a special syntax check is done.

The syntax. For each string in the list of options it is tested whether it starts with a quotation mark (note that this will never happen for normal keys since the normal keys of TikZ do not start with quotation marks). When this happens, the *<string>* should not be a key–value pair, but, rather, must have the form:

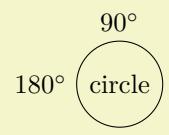
`"<text>"'` *<options>*

(We will discuss the optional apostrophe in a moment. It is not really important for the current option, but only for edge labels, which are discussed later).

Transformation to a label option. When a *<string>* has the above form, it is treated (almost) as if you had written

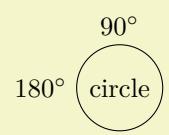
`label={["<text>">']}<options>`

instead. The “almost” refers to the following additional feature: In reality, before the *<options>* are executed inside the `label` command, the direction keys `above`, `left`, `below` `right` and so on are redefined so that `above` is a shorthand for `label position=90` and similarly for the other keys. The net effect is that in order to specify the position of the *<text>* relative to the main node you can just put something like `left` or `above right` inside the *<options>*:



```
\usetikzlibrary {quotes}
\tikz
  \node ["$90^\circ \circlearrowleft$" above, "$180^\circ \circlearrowleft$" left, circle, draw] {circle};
```

Alternatively, you can also use *<direction>:<actual text>* as your *<text>*. This works since the `label` command allows you to specify a direction at the beginning when it is separated by a colon:

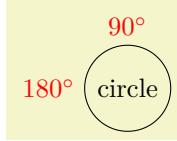


```
\usetikzlibrary {quotes}
\tikz
  \node ["90:$90^\circ \circlearrowleft$", "left:$180^\circ \circlearrowleft$", circle, draw] {circle};
```

Arguably, placing `above` or `left` behind the *<text>* seems more natural than having it inside the *<text>*. In addition to the above, before the *<options>* are executed, the following style is also executed:

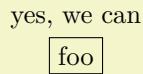
`/tikz/every label quotes`

(style, no value)



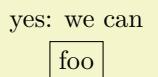
```
\usetikzlibrary {quotes}
\tikz [every label quotes/.style=red]
\node ["90:$90^\circ\circ", "left:$180^\circ\circ", circle, draw] {circle};
```

Handling commas and colons inside the text. The $\langle text \rangle$ may not contain a comma, unless it is inside curly braces. The reason is that the key handler separates the total options of a `node` along the commas it finds. So, in order to have text containing a comma, just add curly braces around either the comma or just around the whole $\langle text \rangle$:



```
\usetikzlibrary {quotes}
\tikz \node ["{yes, we can}", draw] {foo};
```

The same is true for a colon, only in this case you may need to surround specifically the colon by curly braces to stop the `label` option from interpreting everything before the colon as a direction:



```
\usetikzlibrary {quotes}
\tikz \node ["{yes{:} we can}", draw] {foo};
```

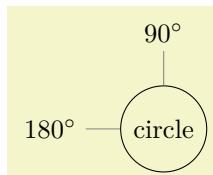
The optional apostrophe. Following the closing quotation marks in a $\langle string \rangle$ there may (but need not) be a single quotation mark (an apostrophe), possibly surrounded by whitespaces. If it is present, it is simply added to the $\langle options \rangle$ as another option (and, indeed, a single apostrophe is a legal option in TikZ, it is a shorthand for `swap`):

String	has the same effect as
"foo"	"foo" {'}
"foo", red	"foo" {',red}
"foo",{red}	"foo" {',red}
"foo'{,red}	"foo" {',red}
"foo"{red,'}	"foo" {red,'}
"foo'{red}	"foo" {'red} (illegal; there is no key 'red)
"foo" red'	"foo" {red'} (illegal; there is no key red')

`/tikz/quotes mean pin`

(no value)

This option has exactly the same effect as `quotes mean label`, only instead of transforming quoted text to the `label` option, they get transformed to the `pin` option:



```
\usetikzlibrary {quotes}
\tikz [quotes mean pin]
\node ["$90^\circ\circ" above, "$180^\circ\circ" left, circle, draw] {circle};
```

Instead of `every label quotes`, the following style is executed with each such pin:

`/tikz/every pin quotes`

(style, no value)

If instead of `labels` or `pins` you would like quoted strings to be interpreted in a different manner, you can also define your own handlers:

`/tikz/node quotes mean=<replacement>`

(no default)

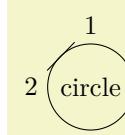
This key allows you to define your own handler for quotes options. Inside the options of a `node`, whenever a key-value pair with the syntax

" $\langle text \rangle$ " , $\langle options \rangle$

is encountered, the following happens: The above string gets replaced by $\langle replacement \rangle$ where inside the $\langle replacement \rangle$ the parameter #1 is $\langle text \rangle$ and #2 is $\langle options \rangle$. If the apostrophe is present (see also the discussion of `quotes mean label`), the $\langle options \rangle$ start with ',.

The $\langle replacement \rangle$ is then parsed normally as options (using `\pgfkeys`).

Here is an example, where the quotes are used to define labels that are automatically named according to the `text`:



```
\usetikzlibrary {quotes}
\begin{tikzset}{node quotes mean={label={[#2,name={#1}]#1}}}

\tikz {
    \node ["1", "2" label position=left, circle, draw] {circle};
    \draw (1) -- (2);
}
```

Some further options provided by the `quotes` library concern labels next to edges rather than nodes and they are described in Section 17.12.2.

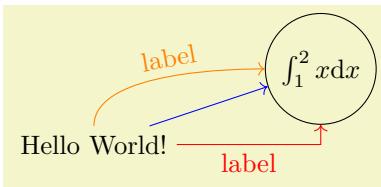
17.11 Connecting Nodes: Using Nodes as Coordinates

Once you have defined a node and given it a name, you can use this name to reference it. This can be done in two ways, see also Section 13.2.3. Suppose you have said `\path(0,0) node(x) {Hello World!};` in order to define a node named `x`.

1. Once the node `x` has been defined, you can use `(x.<anchor>)` wherever you would normally use a normal coordinate. This will yield the position at which the given `<anchor>` is in the picture. Note that transformations do not apply to this coordinate, that is, `(x.north)` will be the northern anchor of `x` even if you have said `scale=3` or `xshift=4cm`. This is usually what you would expect.
2. You can also just use `(x)` as a coordinate. In most cases, this gives the same coordinate as `(x.center)`. Indeed, if the `shape` of `x` is `coordinate`, then `(x)` and `(x.center)` have exactly the same effect.

However, for most other shapes, some path construction operations like `--` try to be “clever” when they are asked to draw a line from such a coordinate or to such a coordinate. When you say `(x)--(1,1)`, the `--` path operation will not draw a line from the center of `x`, but *from the border* of `x` in the direction going towards `(1,1)`. Likewise, `(1,1)--(x)` will also have the line end on the border in the direction coming from `(1,1)`.

In addition to `--`, the curve-to path operation `..` and the path operations `-|` and `|-` will also handle nodes without anchors correctly. Here is an example, see also Section 13.2.3:



```
\begin{tikzpicture}
\path (0,0) node (x) {Hello World!}
(3,1) node[circle,draw](y) {$\int_1^2 x \, dx$};

\draw[->,blue] (x) -- (y);
\draw[->,red] (x) -| node[near start,below]{label} (y);
\draw[->,orange] (x) .. controls +(up:1cm) and +(left:1cm) .. node[above,sloped]{label} (y);
\end{tikzpicture}
```

17.12 Connecting Nodes: Using the Edge Operation

17.12.1 Basic Syntax of the Edge Operation

The `edge` operation works like a `to` operation that is added after the main path has been drawn, much like a node is added after the main path has been drawn. This allows each `edge` to have a different

appearance. As the `node` operation, an `edge` temporarily suspends the construction of the current path and a new path p is constructed. This new path p will be drawn after the main path has been drawn. Note that p can be totally different from the main path with respect to its options. Also note that if there are several `edge` and/or `node` operations in the main path, each creates its own path(s) and they are drawn in the order that they are encountered on the main path.

```
\path ... edge[<options>] <nodes> (<coordinate>) ...;
```

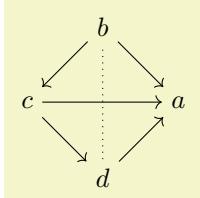
The effect of the `edge` operation is that after the main path the following path is added to the picture:

```
\path[every edge,<options>] (\tikztostart) <path>;
```

Here, $\langle path \rangle$ is the `to` path. Note that, unlike the path added by the `to` operation, the (\tikztostart) is added before the $\langle path \rangle$ (which is unnecessary for the `to` operation, since this coordinate is already part of the main path).

The `\tikztostart` is the last coordinate on the path just before the `edge` operation, just as for the `node` or `to` operations. However, there is one exception to this rule: If the `edge` operation is directly preceded by a `node` operation, then this just-declared node is the start coordinate (and not, as would normally be the case, the coordinate where this just-declared node is placed – a small, but subtle difference). In this regard, `edge` differs from both `node` and `to`.

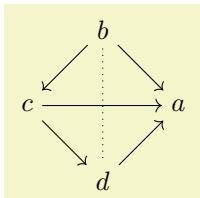
If there are several `edge` operations in a row, the start coordinate is the same for all of them as their target coordinates are not, after all, part of the main path. The start coordinate is, thus, the coordinate preceding the first `edge` operation. This is similar to nodes insofar as the `edge` operation does not modify the current path at all. In particular, it does not change the last coordinate visited, see the following example:



```
\begin{tikzpicture}
\node (a) at (0:1) {$a$};
\node (b) at (90:1) {$b$};
\node (c) at (180:1) {$c$};
\node (d) at (270:1) {$d$};

\path[>] (b) edge (a);
\path[>] (c) edge (a);
\path[<-,dotted] (c) edge (d);
\path[<-] (a) edge (b);
\path[<-] (a) edge (d);
\path[<-] (b) edge (d);
\end{tikzpicture}
```

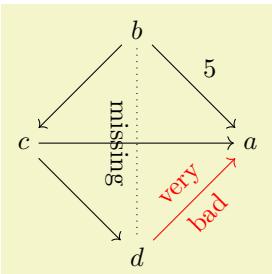
A different way of specifying the above graph using the `edge` operation is the following:



```
\begin{tikzpicture}
\node foreach \name/\angle in {a/0,b/90,c/180,d/270}{\name} at (\angle:1) {$\name$};

\path[>] (b) edge (a)
        edge (c)
        edge [-,dotted] (d);
(c) edge (a)
        edge (d);
(d) edge (a);
\end{tikzpicture}
```

As can be seen, the path of the `edge` operation inherits the options from the main path, but you can locally overrule them.



```
\begin{tikzpicture}
\node foreach \name/\angle in {a/0,b/90,c/180,d/270}{\name} at (\angle:1.5) {$\name$};

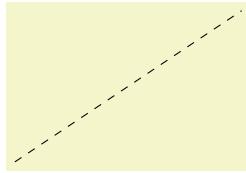
\path[>] (b) edge [node[above right]{$5$}] (a)
        edge [node[below,sloped]{missing}] (c)
        edge [-,dotted] node[below,sloped]{missing} (d);
(c) edge [red] [node[above,sloped]{very}] (a)
        edge [red] [node[below,sloped]{bad}] (d);
\end{tikzpicture}
```

Instead of `every to`, the style `every edge` is installed at the beginning of the main path.

`/tikz/every edge`

(style, initially `draw`)

Executed for each `edge`.



```
\begin{tikzpicture}[every edge/.style={draw,dashed}]
\path (0,0) edge (3,2);
\end{tikzpicture}
```

17.12.2 Nodes on Edges: Quotes Syntax

The standard way of specifying nodes that are placed “on” an edge (or on a to-path; all of the following is also true for to-paths) is to put node specifications after the `edge` keyword, but before the target coordinate. Another way is to use the `edge node` option and its friends. Yet another way is to use the quotes syntax.

The syntax is essentially the same as for labels added to nodes as described in Section 17.10.4 and you also need to load the `quotes` library.

In detail, when the `quotes` library is loaded, each time a key–value pair in a list of options passed to an `edge` or a `to` path command starts with ", the key–value pair must actually be a string of the following form:

"*text*" '*options*'

This string is transformed into the following:

`edge node=node [every edge quotes<i>optionstext}`

As described in Section 17.10.4, the apostrophe becomes part of the *options*, when present.

The following style is important for the placement of the labels:

`/tikz/every edge quotes`

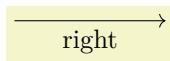
(style, initially `auto`)

This style is `auto` by default, which causes labels specified using the quotes-syntax to be placed next to the edges. Unless the setting of `auto` has been changed, they will be placed to the left.



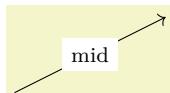
```
\usetikzlibrary {quotes}
\tikz \draw (0,0) edge ["left", ->] (2,0);
```

In order to place all labels to the right by default, change this style to `auto=right`:



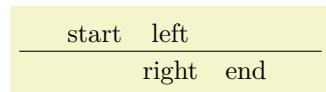
```
\usetikzlibrary {quotes}
\tikz [every edge quotes/.style={auto=right}]
\draw (0,0) edge ["right", ->] (2,0);
```

To place all nodes “on” the edge, just make this style empty (and, possibly, make your labels opaque):



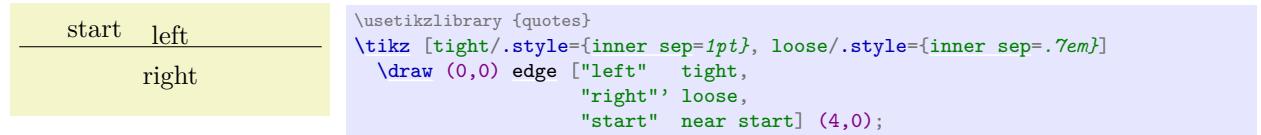
```
\usetikzlibrary {quotes}
\tikz [every edge quotes/.style={fill=white,font=\footnotesize}]
\draw (0,0) edge ["mid", ->] (2,1);
```

You may often wish to place some edge nodes to the right of edges and some to the left. For this, the special treatment of the apostrophe is particularly convenient: Recall that in TikZ there is an option just called ', which is a shorthand for `swap`. Now, following the closing quotation mark come the options of an edge node. Thus, if the closing quotation mark is followed by an apostrophe, the `swap` option will be added to the edge label, causing it to be placed on the other side. Because of the special treatment, you can even add another option like `near end` after the apostrophe without having to add curly braces and commas:



```
\usetikzlibrary {quotes}
\tikz
\draw (0,0) edge ["left", "right",
"start" near start,
"end" near end] (4,0);
```

In order to modify the distance between the edge labels and the edge, you should consider introducing some styles:



17.13 Referencing Nodes Outside the Current Picture

17.13.1 Referencing a Node in a Different Picture

It is possible (but not quite trivial) to reference nodes in pictures other than the current one. This means that you can create a picture and a node therein and, later, you can draw a line from some other position to this node.

To reference nodes in different pictures, proceed as follows:

1. You need to add the `remember picture` option to all pictures that contain nodes that you wish to reference and also to all pictures from which you wish to reference a node in another picture.
2. You need to add the `overlay` option to paths or to whole pictures that contain references to nodes in different pictures. (This option switches the computation of the bounding box off.)
3. You need to use a driver that supports picture remembering and you need to run T_EX twice.

(For more details on what is going on behind the scenes, see Section ??.)

Let us have a look at the effect of these options.

`/tikz/remember picture=<boolean>` (no default, initially `false`)

This option tells TikZ that it should attempt to remember the position of the current picture on the page. This attempt may fail depending on which backend driver is used. Also, even if remembering works, the position may only be available on a second run of T_EX.

Provided that remembering works, you may consider saying

```
\tikzset{every picture/.append style={remember picture}}
```

to make TikZ remember all pictures. This will add one line in the `.aux` file for each picture in your document – which typically is not very much. Then, you do not have to worry about remembered pictures at all.

`/tikz/overlay=<boolean>` (default `true`)

This option is mainly intended for use when nodes in other pictures are referenced, but you can also use it in other situations. The effect of this option is that everything within the current scope is not taken into consideration when the bounding box of the current picture is computed.

You need to specify this option on all paths (or at least on all parts of paths) that contain a reference to a node in another picture. The reason is that, otherwise, TikZ will attempt to make the current picture large enough to encompass *the node in the other picture*. However, on a second run of T_EX this will create an even bigger picture, leading to larger and larger pictures. Unless you know what you are doing, I suggest specifying the `overlay` option with all pictures that contain references to other pictures.

Let us now have a look at a few examples. These examples work only if this document is processed with a driver that supports picture remembering.

Inside the current text we place two pictures, containing nodes named `n1` and `n2`, using

```
\tikz[remember picture] \node[circle,fill=red!50] (n1) {};
```

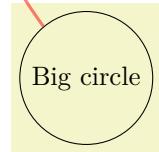
which yields 

```
\tikz[remember picture] \node[fill=blue!50] (n2) {};
```

yielding the node . To connect these nodes, we create another picture using the `overlay` option and also the `remember picture` option.

```
\begin{tikzpicture}[remember picture,overlay]
\draw[->,very thick] (n1) -- (n2);
\end{tikzpicture}
```

Note that the last picture is seemingly empty. What happens is that it has zero size and contains an arrow that lies well outside its bounds. As a last example, we connect a node in another picture to the first two nodes. Here, we provide the `overlay` option only with the line that we do not wish to count as part of the picture.



```
\begin{tikzpicture}[remember picture]
\node (c) [circle,draw] {Big circle};
\draw [overlay,->,very thick,red,opacity=.5]
(c) to[bend left] (n1) (n1) -| (n2);
\end{tikzpicture}
```

17.13.2 Referencing the Current Page Node – Absolute Positioning

There is a special node called `current page` that can be used to access the current page. It is a node of shape rectangle whose `south west` anchor is the lower left corner of the page and whose `north east` anchor is the upper right corner of the page. While this node is handled in a special way internally, you can reference it as if it were defined in some remembered picture other than the current one. Thus, by giving the `remember picture` and the `overlay` options to a picture, you can position nodes *absolutely* on a page.

The first example places some text in the lower left corner of the current page:

```
\begin{tikzpicture}[remember picture,overlay]
\node [xshift=1cm,yshift=1cm] at (current page.south west)
[text width=7cm,fill=red!20,rounded corners,above right]
{
This is an absolutely positioned text in the
lower left corner. No shipout-hackery is used.
};
\end{tikzpicture}
```

The next example adds a circle in the middle of the page.

```
\begin{tikzpicture}[remember picture,overlay]
\draw [line width=1mm,opacity=.25]
(current page.center) circle (3cm);
\end{tikzpicture}
```

The final example overlays some text over the page (depending on where this example is found on the page, the text may also be behind the page).

```
\begin{tikzpicture}[remember picture,overlay]
\node [rotate=60,scale=10,text opacity=0.2]
at (current page.center) {Example};
\end{tikzpicture}
```

17.14 Late Code and Late Options

All options given to a node only locally affect this one node. While this is a blessing in most cases, you may sometimes want to cause options to have effects “later” on. The other way round, you may sometimes note “only later” that some options should be added to the options of a node. For this, the following version of the `node path` command can be used:

This is an absolutely positioned text in the lower left corner. No shipout-hackery is used.

```
\path ... node also[⟨late options⟩](<name>) ... ;
```

Note that the `<name>` is compulsory and that *no* text may be given. Also, the ordering of options and node label must be as above.

The effect of the above is the following effect: The node `<name>` must already be existing. Now, the `⟨late options⟩` are executed in a local scope. Most of these options will have no effect since you *cannot change the appearance of the node*, that is, you cannot change a red node into a green node using these “late” options. However, giving the `append after command` and `prefix after command` options inside the `⟨late options⟩` (directly or indirectly) does have the desired effect: The given path gets executed with the `\tikzlastnode` set to the determined node.

The net effect of all this is that you can provide, say, the `label` option inside the `⟨options⟩` to add a label to a node that has already been constructed.



```
\begin{tikzpicture}
  \node [draw,circle] (a) {Hello};
  \node also [label=above:world] (a);
\end{tikzpicture}
```

As explained in Section 14, you can use the options `append after command` and `prefix after command` to add a path after a node. The following macro may be useful there:

\tikzlastnode

Expands to the last node on the path.

Instead of the `node also` syntax, you can also the following option:

```
/tikz/late options=<options>
```

 (no default)

This option can be given on a path (but not as an argument to a `node` path command) and has the same effect as the `node also` path command. Inside the `⟨options⟩`, you should use the `name` option to specify the node for which you wish to add late options:



```
\begin{tikzpicture}
  \node [draw,circle] (a) {Hello};
  \path [late options={name=a, label=above:world}];
\end{tikzpicture}
```

18 Pics: Small Pictures on Paths

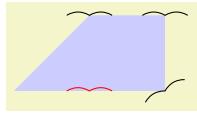
18.1 Overview

A “pic” is a “short picture” (hence the short name...) that can be inserted anywhere in TikZ picture where you could also insert a node. Similarly to nodes, pics have a “shape” (called *type* to avoid confusion) that someone has defined. Each time a pic of a specified type is used, the type’s code is executed, resulting in some drawings to be added to the current picture. The syntax for adding nodes and adding pics to a picture are also very similar. The core difference is that pics are typically more complex than nodes and may consist of a whole bunch of nodes themselves together with complex paths joining them.

As a very simple example, suppose we want to define a pic type `seagull` that just draw “two bumps”. The code for this definition is quite easy:

```
\tikzset{
  seagull/.pic={
    % Code for a "seagull". Do you see it?...
    \draw (-3mm,0) to [bend left] (0,0) to [bend left] (3mm,0);
  }
}
```

The first line just tells TeX that you set some TikZ options for the current scope (which is the whole document); you could put `seagull/.pic=...` anywhere else where TikZ options are allowed (which is just about anywhere). We have now defined a `seagull` pic type and can use it as follows:



```
\tikz \fill [fill=blue!20]
  (1,1)
  -- (2,2) pic      {seagull}
  -- (3,2) pic      {seagull}
  -- (3,1) pic [rotate=30] {seagull}
  -- (2,1) pic [red] {seagull};
```

As can be seen, defining new types of pics is much easier than defining new shapes for nodes; but see Section 18.3 for the fine details.

Since defining new pics types is easier than defining new node shapes and since using pics is as easy as using nodes, why should you use nodes at all? There are chiefly two reasons:

1. Unlike nodes, pics cannot be referenced later on. You *can* reference nodes that are inside a pic, but not “the pic itself”. In particular, you cannot draw lines between pics the way you can draw them between nodes. In general, whenever it makes sense that some drawing could conceivably be connected to other node-like-things, then a node is better than a pic.
2. If pics are used to emulate the full power of a node (which is possible, in principle), they will be slower to construct and take up more memory than a node achieving the same effect.

Despite these drawbacks, pics are an excellent choice for creating highly configurable reusable pieces of drawings that can be inserted into larger contexts.

18.2 The Pic Syntax

\pic

Inside `\tikzpicture` this is an abbreviation for `\path pic`.

The syntax for adding a pic to a picture is very similar to the syntax used for nodes (indeed, internally the same parser code is used). The main difference is that instead of a node contents you provide the picture’s type between the braces:

```
\path ... pic <foreach statements> [<options>] (<prefix>) at(<coordinate>) :<animation
attribute>=<options> {<pic type>} ...;
```

Adds a pic to the current TikZ picture of the specified `<pic type>`. The effect is, basically, that some code associated with the `<pic type>` is executed (how this works, exactly, is explained later). This code can consist of arbitrary TikZ code. As for nodes, the current path will not be modified by this path command, all drawings produced by the code are “external” to the path the same way neither a node nor its border are part of the path on which they are specified.

Just like the `node` command, this path operation is somewhat complex and we go over it step by step.

Order of the parts of the specification. Just like for nodes, everything between “pic” and the opening brace of the `<pic type>` is optional and can be given in any order. If there are `<foreach statements>`, they must come first, directly following “pic”. As for nodes, the “end” of the pic specification is normally detected by the presence of the opening brace. You can, however, use the `pic type` option to specify the pic type as an option.

`/tikz/pic type=<pic type>`

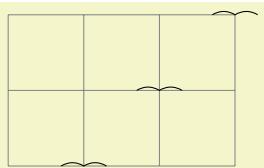
(no default)

This key sets the pic type of the current `pic`. When this option is used inside an option block of a `pic`, the parsing of the `pic` ends immediately and no pic type in braces is expected. (In other words, this option behaves exactly like the `node contents` option and, indeed, the two are interchangeable.)



```
\tikz {
  \path (0,0) pic [pic type = seagull]
    (1,0) pic
      {seagull};
}
```

The location of a pic. Just like nodes, pics are placed at the last position mentioned on the path or, when `at` is used, at a specified position. “Placing” a pic somewhere actually means that the coordinate system is translated (shifted) to this last position. This means that inside of the pic type’s code any mentioning of the origin refers to the last position used on the path or to the specified `at`.



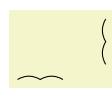
```
\tikz { % different ways of placing pics
  \draw [help lines] (0,0) grid (3,2);
  \pic at (1,0) {seagull};
  \path (2,1) pic {seagull};
  \pic [at={(3,2)}] {seagull};
}
```

As for nodes, except for the described shifting, the coordinate system of a pic is reset prior to executing the pic type’s code. This can be changed using the `transform shape` option, which has the same effect as for nodes: The “outer” transformation gets applied to the node:



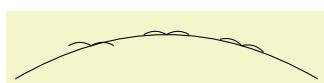
```
\tikz [scale=2] {
  \pic at (0,0) {seagull};
  \pic at (1,0) [transform shape] {seagull};
}
```

When the `<options>` contain transformation commands like `scale` or `rotate`, these transformations always apply to the pic:



```
\tikz [rotate=30] {
  \pic at (0,0) {seagull};
  \pic at (1,0) [rotate=90] {seagull};
}
```

Just like nodes, pics can also be positioned implicitly and, somewhat unsurprisingly, the same rules concerning positioning and sloping apply:



```
\tikz \draw
  (0,0) to [bend left]
    pic [near start] {seagull}
    pic {seagull}
    pic [sloped, near end] {seagull} (4,0);
```

The options of a node. As always, any given `<options>` apply only to the pic and have no effect outside. As for nodes, most “outside” options also apply to the pics, but not the “action” options like `draw` or `fill`. These must be given in the `<options>` of the pic.

The code of a pic. As stated earlier, the main job of a pic is to execute some code in a scope that is shifted according to the last point on the path or to the `at` position specified in the pic. It was also claimed that this code is specified by the `<pic type>`. However, this specification is somewhat indirect.

What really happens is the following: When a `pic` is encountered, the current path is suspended and a new internal scope is started. The `<options>` are executed and also the `<pic type>` (as explained in a moment). After all this is done, the code stored in the following key gets executed:

`/tikz/pics/code=<code>` (no default)

This key stores the `<code>` that should be drawn in the current pic. Normally, setting this key is done by the `<pic type>`, but you can also set it in the `<options>` and leave the `<pic type>` empty:

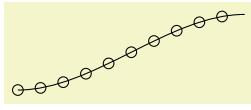


```
\tikz \pic [pics/code={\draw (-3mm,0) to[bend left] (0,0)
to[bend left] (3mm,0);}]

{}; % no pic type specified
```

Now, how does the `<pic type>` set `pics/code`? It turns out that the `<pic type>` is actually just a list of keys that are executed with the prefix `/tikz/pics/`. In the above examples, this “list of keys” just consisted of the single key “`seagull`” that did not take any arguments, but, in principle, you could provide any arbitrary text understood by `\pgfkeys` here. This means that when we write `pic{seagull}`, TikZ will execute the key `/tikz/pics/seagull`. It turns out, see Section 18.3, that this key is just a style set to `code={\draw(-3mm,0)...;}`. Thus, `pic{seagull}` will cause the `pics/code` key to be set to the text needed to draw the seagull.

Indeed, you can also use the `<pic type>` simply to set the `code` of the pic. This is useful for cases when you have some code that you “just want to execute, but do not want to define a new pic type”. Here is a typical example where we use `pics` to add some markings to a path:



```
\tikz \draw (0,0) .. controls(1,0) and (2,1) .. (3,1)
foreach \t in {0, 0.1, ..., 1} {
  pic [pos=\t] {code={\draw circle [radius=2pt];}}
};
```

In our seagull example, we always explicitly used `\draw` to draw the seagull. This implies that when a user writes something `pic[fill]{seagull}` in the hope of having a “filled” seagull, nothing special will happen: The `\draw` inside the `pic` explicitly states that the path should be drawn, not filled, and the fact that in the surrounding scope the `fill` option is set has no effect. The following key can be used to change this:

`/tikz/pic actions` (no value)

This key is a style that can be used (only) inside the code of a `pic`. There, it will set the “action” keys set inside the `<options>` of the `pic` (“actions” are drawing, filling, shading, and clipping or any combination thereof).

To see how this key works, let us define the following `pic`:

```
\tikzset{
  my pic/.pic = {
    \path [pic actions] (0,0) circle[radius=3mm];
    \draw (-3mm,-3mm) rectangle (3mm,3mm);
  }
}
```

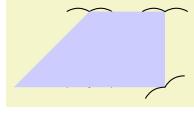
In the code, whether or not the circle gets drawn/filled/shaded depends on which options were given to the `pic` command when it is used. In contrast, the rectangle will always (just) be drawn.



```
\tikz \pic {my pic}; \space
\tikz \pic [red] {my pic}; \space
\tikz \pic [draw] {my pic}; \space
\tikz \pic [draw=red] {my pic}; \space
\tikz \pic [draw, shading=ball] {my pic}; \space
\tikz \pic [fill=red!50] {my pic}; \space
```

Code executed behind or in front of the path. As for nodes, a `pic` can be “behind” the current path or “in front of it” and, just as for nodes, the two options `behind path` and `in front of path` are used to specify which is meant. In detail, if `node` and `pic` are both used repeatedly on a path, in

the resulting picture we first see all nodes and pics with the `behind path` option set in the order they appear on the path (nodes and pics are interchangeable in this regard), then comes the path, and then come all nodes and pics that are in front of the path in the order they appeared.



```
\tikz \fill [fill=blue!20]
  (1,1)
  -- (2,2) pic [behind path] {seagull}
  -- (3,2) pic {seagull}
  -- (3,1) pic [rotate=30] {seagull}
  -- (2,1) pic [red, behind path] {seagull};
```

In contrast to nodes, a pic need not only be completely behind the path or in front of the path as specified by the user. Instead, a pic type may also specify that a certain part of the drawing should always be behind the path and it may specify that a certain other part should always be before the path. For this, the values of the following keys are relevant:

`/tikz/pics/foreground code=<code>`

(no default)

This key stores `<code>` that will always be drawn in front of the current path, even when `behind path` is used. If `behind path` is not used and `code` is (also) set, the code of `code` is drawn first, following by the foreground `<code>`.

`/tikz/pics/background code=<code>`

(no default)

Like `foreground code`, only that the `<code>` is always put behind the path, except when the `behind path` option is applied to the pic, then the background code is drawn in front of the “behind path” code.

The `foreach` statement for pics. As for nodes, a pic specification may start with `foreach`. The effect and semantics are the same as for nodes.



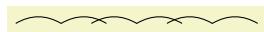
```
\tikz \pic foreach \x in {1,2,3} at (\x,0) {seagull};
```

Styles for pics. The following styles influence how nodes are rendered:

`/tikz/every pic`

(style, initially empty)

This style is installed at the beginning of every pic.



```
\begin{tikzpicture}[every pic/.style={scale=2,transform shape}]
  \pic foreach \x in {1,2,3} at (\x,0) {seagull};
\end{tikzpicture}
```

Name scopes. You can specify a `<name>` for a pic using the key `name=<name>` or by giving the name in parenthesis inside the pic’s specification. The effect of this is, for once, quite different from what happens for nodes: All that happens is that `name prefix` is set to `<name>` at the beginning of the pic. The `name prefix` key was already introduced in the description of the `node` command: It allows you to set some text that is prefixed to all nodes in a scope. For pics this makes particular sense: All nodes defined by a pic’s code can be referenced from outside the pic with the prefix provided.

To see how this works, let us add some nodes to the code of the seagull:

```
\tikzset{
  seagull/.pic={
    % Code for a "seagull". Do you see it?...
    \coordinate (-left wing) at (-3mm,0);
    \coordinate (-head)      at (0,0);
    \coordinate (-right wing) at (3mm,0);

    \draw (-left wing) to [bend left] (0,0) (-head) to [bend left] (-right wing);
  }
}
```

Now, we can use it as follows:

```
\tikz {
  \pic (Emma) [seagull];
  \pic (Alexandra) at (0,1) [seagull];

  \draw (Emma-left wing) -- (Alexandra-right wing);
}
```

Sometimes, you may also wish your pic to access nodes outside the pic (typically, because they are given as parameters). In this case, the name prefix gets in the way since the nodes outside the picture do not have this prefix. The trick is to locally reset the name prefix to the value it had outside the picture, which is achieved using the following style:

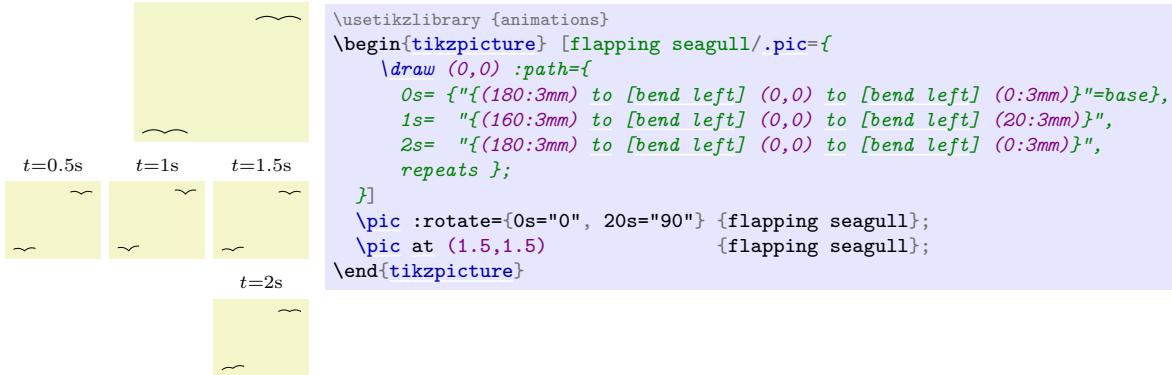
`/tikz/name prefix ..` (no value)

This key is available only inside the code of a pic. There, it (locally) changes the name prefix to the value it had outside the pic. This allows you to access nodes outside the current pic.

Animations for pics. Just as for nodes, you can use the attribute–colon syntax to add an animation to a pic:



Naturally, you can also use animations in the code of a picture:



There are two general purpose keys that pics may find useful:

`/tikz/pic text=<text>` (no default)

This macro stores the `<text>` in the macro `\tikzpictext`, which is `\let` to `\relax` by default. Setting the `pic text` to some value is the “preferred” way of communicating a (single) piece of text that should become part of a pic (typically of a node). In particular, the `quotes` library maps quoted parameters to this key.

`/tikz/pic text options=<options>` (no default)

This macro stores the `<options>` in the macro `\tikzpictextoptions`, which is `\let` to the empty string by default. The `quotes` library maps options for quoted parameters to this key.

18.2.1 The Quotes Syntax

When you load the `quotes` library, you can use the “quotes syntax” inside the options of a pic. Recall that for nodes this syntax is used to add a label to a node. For pics, the quotes syntax is used to set the `pic text` key. Whether or not the pic type’s code takes this key into consideration is, however, up to the key.

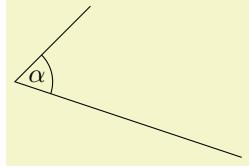
In detail, when the `quotes` library is loaded, each time a key–value pair in a list of options passed to an `pic` starts with `",` the key–value pair must actually be a string of the following form:

```
"<text>"'<options>'
```

This string is transformed into the following:

```
every pic quotes/.try, pic text=<text>, pic text options={<options>}
```

As example of a pic type that takes these values into account is the `angle` pic type:



```
\usetikzlibrary {angles,quotes}
\begin{tikzpicture}
\draw (3,0) coordinate (A)
    -- (0,1) coordinate (B)
    -- (1,2) coordinate (C)
    pic [draw, "\$\alpha\$"] {angle};
\end{tikzpicture}
```

As described in Section 17.10.4, the apostrophe becomes part of the `<options>`, when present. As can be seen above, the following style is executed:

```
/tikz/every pic quotes
```

(style, initially empty)

18.3 Defining New Pic Types

As explained in the description of the `pic` command, in order to define a new pic type you need to

1. define a key with the path prefix `/tikz/pics` that
2. sets the key `/tikz/pics/code` to the code of the pic.

It turns out that this is easy enough to achieve using styles:

```
\tikzset{
  pics/seagull/.style ={
    % Ok, this is the key that should, when
    % executed, set the code key:
    code = {%
      \draw (... ) ... ;
    }
}
```

Even though the above pattern is easy enough, there is a special so-called key handler that allows us to write even simpler code, namely:

```
\tikzset{
  seagull/.pic = {
    \draw (... ) ... ;
  }
}
```

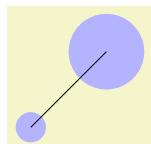
Key handler `<key>/ .pic=<some code>`

This handler can only be used with a key with the prefix `/tikz/`, so just should normally use it only as an option to a TikZ command or to the `\tikzset` command. It takes the `<key>`'s path and, inside that path, it replaces `/tikz/` by `/tikz/pics/` (so, basically, it adds the “missing” `pics` part of the path). Then, it sets up things so that the resulting name to key is a style that executes `code=<some code>`.

In almost all cases, the `.pic` key handler will suffice to setup keys. However, there are cases where you really need to use the first version using `.style` and `code=`:

- Whenever your pic type needs to set the foreground or the background code.
- In case of complicated arguments given to the keys.

As an example, let us define a simple pic that draws a filled circle behind the path. Furthermore, we make the circle's radius a parameter of the pic:



```
\tikzset{
  pics/my circle/.style = {
    background code = { \fill circle [radius=#1]; }
  }
}
\tikz [fill=blue!30]
\draw (0,0) pic {my circle=2mm} -- (1,1) pic {my circle=5mm};
```

19 Specifying Graphs

19.1 Overview

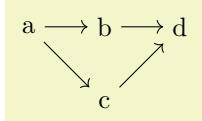
TikZ offers a powerful path command for specifying how the nodes in a graph are connected by edges and arcs: The `graph` path command, which becomes available when you load the `graphs` library.

TikZ Library `graphs`

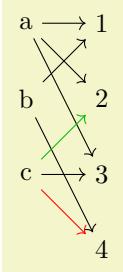
```
\usetikzlibrary{graphs} % LATEX and plain TEX
\usetikzlibrary[graphs] % ConTeXt
```

The package must be loaded to use the `graph` path command.

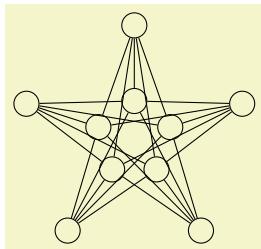
In this section, by *graph* we refer to a set of nodes together with some edges (sometimes also called arcs, in case they are directed) such as the following:



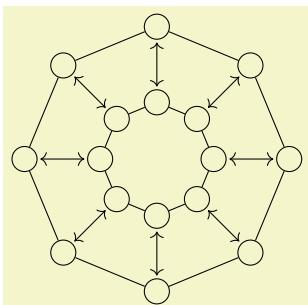
```
\usetikzlibrary {graphs}
\tikz \graph { a -> {b, c} -> d };
```



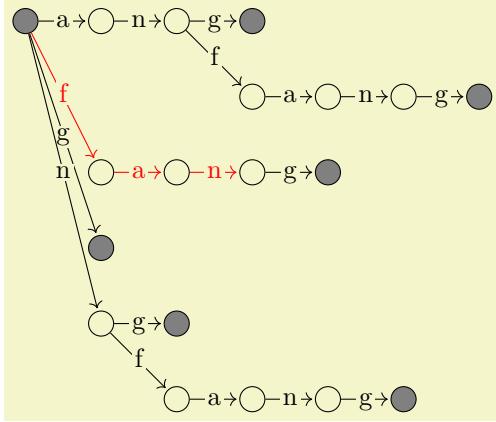
```
\usetikzlibrary {graphs.standard}
\tikz \graph {
    subgraph I_nm [V={a, b, c}, W={1,...,4}];
    a -> { 1, 2, 3 };
    b -> { 1, 4 };
    c -> { 2 [>green!75!black], 3, 4 [>red] }
};
```



```
\usetikzlibrary {graphs}
\tikz
\graph [nodes={draw, circle}, clockwise, radius=.5cm, empty nodes, n=5] {
    subgraph I_n [name=inner] --[complete bipartite]
    subgraph O_n [name=outer]
};
```



```
\usetikzlibrary {graphs}
\tikz
\graph [nodes={draw, circle}, clockwise, radius=.75cm, empty nodes, n=8] {
    subgraph C_n [name=inner] <->[shorten <=1pt, shorten >=1pt]
    subgraph C_n [name=outer]
};
```



```
\usetikzlibrary {graphs}
\begin{tikz} [x=1cm, y=1cm, rotate=90, xscale=-1,
mark/.style={fill=black!50}, mark/.default=]
\graph [trie, simple,
nodes={circle, draw},
edges={nodes=t,
inner sep=1pt, anchor=mid,
fill=graphicbackground}], % yellowish background
put node text on incoming edges]
{
root[mark] --> {
  a --> n --> {
    g [mark],
    f --> a --> n --> g [mark]
  },
  f --> a --> n --> g [mark],
  g [mark],
  n --> {
    g [mark],
    f --> a --> n --> g [mark]
  }
},
{ [edges=red] % highlight one path
root --> f --> a --> n
}
};
}
```

The nodes of a graph are normal TikZ nodes, the edges are normal lines drawn between nodes. There is nothing in the `graphs` library that you cannot do using the normal `\node` and the `edge` commands. Rather, its purpose is to offer a concise and powerful way of *specifying* which nodes are present and how they are connected. The `graphs` library only offers simple methods for specifying *where* the nodes should be shown, its main strength is in specifying which nodes and edges are present in principle. The problem of finding “good positions on the canvas” for the nodes of a graph is left to *graph drawing algorithms*, which are covered in Part IV of this manual and which are not part of the `graphs` library; indeed, these algorithms can be used also with graphs specified using `node` and `edge` commands.

The `graphs` library uses a syntax that is quite different from the normal TikZ syntax for specifying nodes. The reason for this is that for many medium-sized graphs it can become quite cumbersome to specify all the nodes using `\node` repeatedly and then using a great number of `edge` command; possibly with complicated `\foreach` statements. Instead, the syntax of the `graphs` library is loosely inspired by the DOT format, which is quite useful for specifying medium-sized graphs, with some extensions on top.

19.2 Concepts

The present section aims at giving a quick overview of the main concepts behind the `graph` command. The exact syntax is explained in more detail in later sections.

19.2.1 Concept: Node Chains

The basic way of specifying a graph is to write down a *node chain* as in the following example:

foo → bar → blub

```
\usetikzlibrary {graphs}
\tikz [every node/.style = draw]
\graph { foo -> bar -> blub };
```

As can be seen, the text `foo -> bar -> my node` creates three nodes, one with the text `foo`, one with `bar` and one with the text `blub`. These nodes are connected by arrows, which are caused by the `->` between the node texts. Such a sequence of node texts and arrows between them is called a *chain* in the following.

Inside a graph there can be more than one chain:

a → b → c d → e → f g → f

```
\usetikzlibrary {graphs}
\tikz \graph {
  a -> b -> c;
  d -> e -> f;
  g -> f;
};
```

Multiple chains are separated by a semicolon or a comma (both have exactly the same effect). As the example shows, when a node text is seen for the second time, instead of creating a new node, a connection is created to the already existing node.

When a node like `f` is created, both the node name and the node text are identical by default. This is not always desirable and can be changed by using the `as` key or by providing another text after a slash:

```
\usetikzlibrary {graphs}
\begin{tikz} \graph {
    x1-> x2 -> x3, x4;
    x1 -> [bend left] x34;
};
```

When you wish to use a node name that contains special symbols like commas or dashes, you must surround the node name by quotes. This allows you to use quite arbitrary text as a “node name”:

```
\usetikzlibrary {graphs}
\begin{tikz} \graph {
    "$x_1$"-> "$x_2$"["red"] -> "$x_3,x_4$";
    "$x_1$"->[bend left] "$x_3,x_4$";
};
```

19.2.2 Concept: Chain Groups

Multiple chains that are separated by a semicolon or a comma and that are surrounded by curly braces form what will be called a *chain group* or just a *group*. A group in itself has no special effect. However, things get interesting when you write down a node or even a whole group and connect it to another group. In this case, the “exit points” of the first node or group get connected to the “entry points” of the second node or group:

```
\usetikzlibrary {graphs}
\begin{tikz} \graph {
    a-> b-> c-> f;
    a-> d-> e;
    {b,c}-> f;
};
```

Chain groups make it easy to create tree structures:

```
\begin{tikz} \graph [grow down, branch right=2.5cm] {
    root-> child 1;
    root-> child 2;
    root-> child 3;
    child 2-> grand child 1;
    child 2-> grand child 2;
    child 3-> grand child 1;
    child 3-> grand child 2;
    child 3-> grand child 3;
};
```

```
\usetikzlibrary {graphs}
\begin{tikz} \graph [grow down, branch right=2.5cm] {
    root-> child 1;
    child 2-> {
        grand child 1,
        grand child 2
    },
    child 3-> {
        grand child 1,
        grand child 2,
        grand child 3
    }
};
```

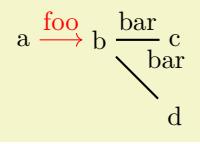
As can be seen, the placement is not particularly nice by default, use the algorithms from the graph drawing libraries to get a better layout. For instance, adding `tree layout` to the above code results in the following somewhat more pleasing rendering: (You need to use LuaTeX to typeset this graphic.)

19.2.3 Concept: Edge Labels and Styles

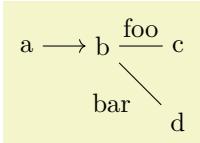
When connectors like `->` or `--` are used to connect nodes or whole chain groups, one or more edges will typically be created. These edges can be styled easily by providing options in square brackets directly after these connectors:

```
\usetikzlibrary {graphs}
\begin{tikz} \graph {
    a->[red] b --[thick] {c, d};
};
```

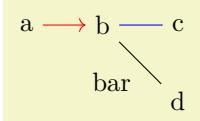
Using the quotes syntax, see Section 17.10.4, you can even add labels to the edges easily by putting the labels in quotes:



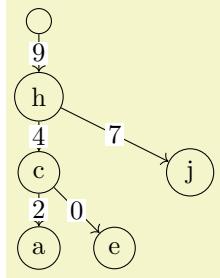
For the first edge, the effect is as desired, however between `b` and the group `{c, d}` two edges are inserted and the options `thick` and the label option `"bar"` is applied to both of them. While this is the correct and consistent behaviour, we typically might wish to specify different labels for the edge going from `b` to `c` and the edge going from `b` to `d`. To achieve this effect, we can no longer specify the label as part of the options of `--`. Rather, we must pass the desired label to the nodes `c` and `d`, but we must somehow also indicate that these options actually “belong” to the edge “leading to” to nodes. This is achieved by preceding the options with a greater-than sign:



Symmetrically, preceding the options by `<` causes the options and labels to apply to the “outgoing” edges of the node:



This syntax allows you to easily create trees with special edge labels as in the following example of a treap:



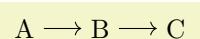
```
\usetikzlibrary {graphs,quotes}
\begin{tikzpicture}
\graph [edge quotes={fill=white, inner sep=1pt}, grow down, branch right, nodes={circle, draw}] {
    "" -> h [>"9"] -> {
        c [>"4"] -> {
            a [>"2"],
            e [>"0"]
        },
        j [>"7"]
    };
};

```

19.2.4 Concept: Node Sets

When you write down some node text inside a `graph` command, a new node is created by default unless this node has already been created inside the same `graph` command. In particular, if a node has already been declared outside of the current `graph` command, a new node of the same name gets created.

This is not always the desired behaviour. Often, you may wish to make nodes part of a graph than have already been defined prior to the use of the `graph` command. For this, simply surround a node name by parentheses. This will cause a reference to be created to an already existing node:

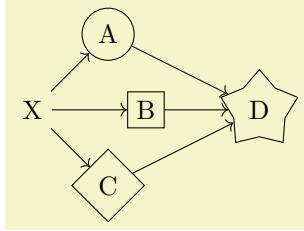


```
\usetikzlibrary {graphs}
\begin{tikzpicture}
\node (a) at (0,0) {A};
\node (b) at (1,0) {B};
\node (c) at (2,0) {C};

\graph { (a) -> (b) -> (c) };

```

You can even go a step further: A whole collection of nodes can all be flagged to belong to a *node set* by adding the option `set=<node set name>`. Then, inside a `graph` command, you can collectively refer to these nodes by surrounding the node set name in parentheses:

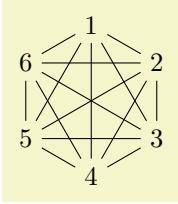


```
\usetikzlibrary {graphs,shapes.geometric}
\begin{tikzpicture}
\node [set=my nodes, circle, draw] at (1,1) {A};
\node [set=my nodes, rectangle, draw] at (1.5,0) {B};
\node [set=my nodes, diamond, draw] at (1,-1) {C};
\node (d) [star, draw] at (3,0) {D};

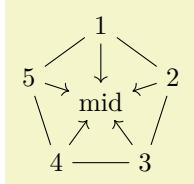
\graph { X --> (my nodes) --> (d) };
\end{tikzpicture}
```

19.2.5 Concept: Graph Macros

Often, a graph will consist – at least in parts – of standard parts. For instance, a graph might contain a cycle of certain size or a path or a clique. To facilitate specifying such graphs, you can define a *graph macro*. Once a graph macro has been defined, you can use the name of the graph to make a copy of the graph part of the graph currently being specified:



```
\usetikzlibrary {graphs.standard}
\begin{tikzpicture}
\graph { subgraph K_n [n=6, clockwise] };
\end{tikzpicture}
```



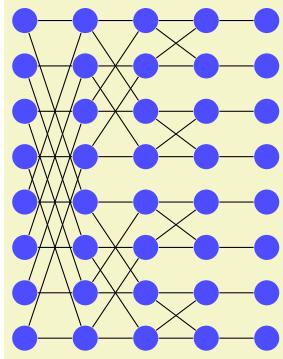
```
\usetikzlibrary {graphs.standard}
\begin{tikzpicture}
\graph { subgraph C_n [n=5, clockwise] -> mid };
\end{tikzpicture}
```

The library `graphs.standard` defines a number of such graphs, including the complete clique K_n on n nodes, the complete bipartite graph $K_{n,m}$ with shores sized n and m , the cycle C_n on n nodes, the path P_n on n nodes, and the independent set I_n on n nodes.

19.2.6 Concept: Graph Expressions and Color Classes

When a graph is being constructed using the `graph` command, it is constructed recursively by uniting smaller graphs to larger graphs. During this recursive union process the nodes of the graph get implicitly *colored* (conceptually) and you can also explicitly assign colors to individual nodes and even change the colors as the graph is being specified. All nodes having the same color form what is called a *color class*.

The power of color class is that special *connector operators* allow you to add edges between nodes having certain colors. For instance, saying `clique=red` at the beginning of a group will cause all nodes that have been flagged as being (conceptually) “red” to be connected as a clique. Similarly, saying `complete bipartite={red}{green}` will cause edges to be added between all red and all green nodes. More advanced connectors, like the `butterfly` connector, allow you to add edges between color classes in a fancy manner.



```
\usetikzlibrary {graphs}
\tikz [x=8mm, y=6mm, circle]
\graph [nodes={fill=blue!70}, empty nodes, n=8] {
    subgraph I_n [name=A] --[butterfly={level=4}]
    subgraph I_n [name=B] --[butterfly={level=2}]
    subgraph I_n [name=C] --[butterfly]
    subgraph I_n [name=D] --
    subgraph I_n [name=E]
};
```

19.3 Syntax of the Graph Path Command

19.3.1 The Graph Command

In order to construct a graph, you should use the `graph` path command, which can be used anywhere on a path at any place where you could also use a command like, say, `plot` or `--`.

\graph

Inside a `{tikzpicture}` this is an abbreviation for `\path graph`.

```
\path ... graph[<options>]<group specification> ...;
```

When this command is encountered on a path, the construction of the current path is suspended (similarly to an `edge` command or a `node` command). In a local scope, the `<options>` are first executed with the key path `/tikz/graphs` using the following command:

\tikzgraphsset{<options>}

Executes the `<options>` with the path prefix `/tikz/graphs`.

Apart from the keys explained in the following, further permissible keys will be listed during the course of the rest of this section.

/tikz/graphs/every graph

(style, no value)

This style is executed at the beginning of every `graph` path command prior to the `<options>`.

Once the scope has been set up and once the `<options>` have been executed, a parser starts to parse the `<group specification>`. The exact syntax of such a group specification is explained in detail in Section 19.3.2. Basically, a group specification is a list of chain specifications, separated by commas or semicolons.

Depending on the content of the `<group specification>`, two things will happen:

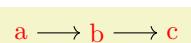
1. A number of new nodes may be created. These will be inserted into the picture in the same order as if they had been created using multiple `node` path commands at the place where the `graph` path command was used. In other words, all nodes created in a `graph` path command will be painted on top of any nodes created earlier in the path and behind any nodes created later in the path. Like normal nodes, the newly created nodes always lie on top of the path that is currently being created (which is often empty, for instance when the `\graph` command is used).
2. Edges between the nodes may be added. They are added in the same order as if the `edge` command had been used at the position where the `graph` command is being used.

Let us now have a look at some common keys that may be used inside the `<options>`:

/tikz/graphs/nodes=<options>

(no default)

This option causes the `<options>` to be applied to each newly created node inside the `<group specification>`.



```
\usetikzlibrary {graphs}
\tikz \graph [nodes=red] { a -> b -> c };
```

Multiple uses of this key accumulate.

`/tikz/graphs/edges=(options)` (no default)

This option causes the `(options)` to be applied to each newly created edge inside the `(group specification)`.

```
\usetikzlibrary {graphs}
\begin{tikzpicture}
\graph [edges={red, thick}] { a -> b -> c };
\end{tikzpicture}
```

Again, multiple uses of this key accumulate.

`/tikz/graphs/edge=(options)` (no default)

This is an alias for `edges`.

`/tikz/graphs/edge node=(node specification)` (no default)

This key specifies that the `(node specification)` should be added to each newly created edge as an implicitly placed node.

```
\usetikzlibrary {graphs}
\begin{tikzpicture}
\graph [edge node={node [red, near end] {X}}] { a -> b -> c };
\end{tikzpicture}
```

Again, multiple uses of this key accumulate.

```
\usetikzlibrary {graphs}
\begin{tikzpicture}
\graph [edge node={node [red, near end] {X}, node [red, near start] {Y}}] { a -> b -> c };
\end{tikzpicture}
```

`/tikz/graphs/edge label=(text)` (no default)

This key is an abbreviation for `edge node=node[auto]{<text>}`. The net effect is that the `text` is placed next to the newly created edges.

```
\usetikzlibrary {graphs}
\begin{tikzpicture}
\graph [edge label=x] { a -> b -> c };
\end{tikzpicture}
```

`/tikz/graphs/edge label'=(text)` (no default)

This key is an abbreviation for `edge node=node[auto,swap]{<text>}`.

```
\usetikzlibrary {graphs.standard}
\begin{tikzpicture}
\graph [edge label=out, edge label'=in]
{ subgraph C_n [clockwise, n=5] };
\end{tikzpicture}
```

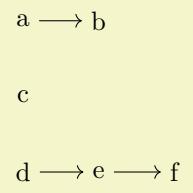
19.3.2 Syntax of Group Specifications

A `(group specification)` inside a `graph` path command has the following syntax:

`{[(options)](list of chain specifications)}`

The `(chain specifications)` must contain chain specifications, whose syntax is detailed in the next section, separated by either commas or semicolons; you can freely mix them. It is permissible to use empty lines (which are mapped to `\par` commands internally) to structure the chains visually, they are simply ignored by the parser.

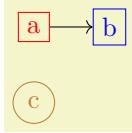
In the following example, the group specification consists of three chain specifications, namely of `a -> b`, then `c` alone, and finally `d -> e -> f`:



```
\usetikzlibrary {graphs}
\begin{tikz} \graph {
    a -> b,
    c;
    d -> e -> f
};
```

The above has the same effect as the more compact group specification `{a->b,c,d->e->f}`.

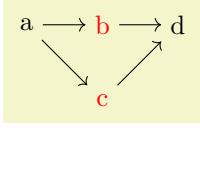
Commas are used to detect where chain specifications end. However, you will often wish to use a comma also inside the options of a single node like in the following example:



```
\usetikzlibrary {graphs}
\begin{tikz} \graph {
    a [red, draw] -> b [blue, draw],
    c [brown, draw, circle]
};
```

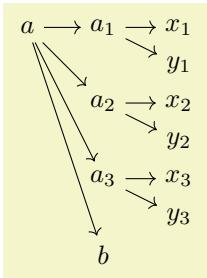
Note that the above example works as expected: The first comma inside the option list of `a` is *not* interpreted as the end of the chain specification “`a [red]`”. Rather, commas inside square brackets are “protected” against being interpreted as separators of group specifications.

The `<options>` that can be given at the beginning of a group specification are local to the group. They are executed with the path prefix `/tikz/graphs`. Note that for the outermost group specification of a graph it makes no difference whether the options are passed to the `graph` command or whether they are given at the beginning of this group. However, for groups nested inside other groups, it does make a difference:



```
\usetikzlibrary {graphs}
\begin{tikz} \graph {
    a -> { [nodes=red] % the option is local to these nodes:
        b, c
    } ->
    d
};
```

Using foreach. There is special support for the `\foreach` statement inside groups: You may use the statement inside a group specification at any place where a `<chain specification>` would normally go. In this case, the `\foreach` statement is executed and for each iteration the content of the statement’s body is treated and parsed as a new chain specification.



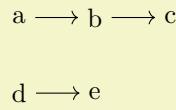
```
\usetikzlibrary {graphs}
\begin{tikz} \graph [math nodes, branch down=5mm] {
    a -> {
        \foreach \i in {1,2,3} {
            a_\i -> { x_\i, y_\i }
        },
        b
    };
};
```

Using macros. In some cases you may wish to use macros and TeX code to compute which nodes and edges are present in a group. You cannot use macros in the normal way inside a graph specification since the parser does not expand macros as it scans for the start and end of groups and node names. Rather, only after commas, semicolons, and hyphens have already been detected and only after all other parsing decisions have been made will macros be expanded. At this point, when a macro expands to, say `a,b`, this will not result in two nodes to be created since the parsing is already done. For these reasons, a special key is needed to make it possible to “compute” which nodes should be present in a group.

`/tikz/graph/parse=<text>`

(no default)

This key can only be used inside the $\langle options \rangle$ of a $\langle group specification \rangle$. Its effect is that the $\langle text \rangle$ is inserted at the beginning of the current group as if you had entered it there. Naturally, it makes little sense to just write down some static $\langle text \rangle$ since you could just as well directly place it at the beginning of the group. The real power of this command stems from the fact that the keys mechanism allows you to say, for instance, `parse/.expand once` to insert the text stored in some macro into the group.



```

\usetikzlibrary {graphs}
\def\mychain{ a -> b -> c; }
\tikz \graph { [parse/.expand once=\mychain] d -> e };
  
```

In the following, more fancy example we use a loop to create a chain of dynamic length.



```

\usetikzlibrary {graphs}
\def\mychain#1{
  \def\mytext{#1}
  \foreach \i in {2,...,#1} {
    \xdef\mytext{\mytext -> \i}
  }
  \tikzgraphsset{my chain/.style={
    /utils/exec=\mychain{#1},
    parse/.expand once=\mytext
  }}
  \tikz \graph { [my chain=4] };
  
```

Multiple uses of this key accumulate, that is, all the texts given in the different uses is inserted in the order it is given.

19.3.3 Syntax of Chain Specifications

A $\langle chain specification \rangle$ has the following syntax: It consists of a sequence of $\langle node specifications \rangle$, where subsequent node specifications are separated by $\langle edge specifications \rangle$. Node specifications, which typically consist of some text, are discussed in the next section in more detail. They normally represent a single node that is either newly created or exists already, but they may also specify a whole set of nodes.

An $\langle edge specification \rangle$ specifies *which* of the node(s) to the left of the edge specification should be connected to which node(s) to the right of it and it also specifies in which direction the connections go. In the following, we only discuss how the direction is chosen, the powerful mechanism behind choosing which nodes should be connect is detailed in Section 19.7.

The syntax of an edge specification is always one of the following five possibilities:

```

-> [(options)]
-- [(options)]
<- [(options)]
<-> [(options)]
-!- [(options)]
  
```

The first four correspond to a directed edge, an undirected edge, a “backward” directed edge, and a bidirected edge, respectively. The fifth edge specification means that there should be no edge (this specification can be used together with the `simple` option to remove edges that have previously been added, see Section 19.5).

Suppose the nodes $\langle left nodes \rangle$ are to the left of the $\langle edge specification \rangle$ and $\langle right nodes \rangle$ are to the right and suppose we have written `->` between them. Then the following happens:

1. The $\langle options \rangle$ are executed (inside a local scope) with the path `/tikz/graphs`. These options may setup the connector algorithm (see below) and may also use keys like `edge` or `edge label` to specify how the edge should look like. As a convenience, whenever an unknown key is encountered for the path `/tikz/graphs`, the key is passed to the `edge` key. This means that you can directly use options like `thick` or `red` inside the $\langle options \rangle$ and they will apply to the edge as expected.
2. The chosen connector algorithm, see Section 19.7, is used to compute from which of the $\langle left nodes \rangle$ an edge should lead to which of the $\langle right nodes \rangle$. Suppose that $(l_1, r_1), \dots, (l_n, r_n)$ is the list of node pairs that result (so there should be an edge between l_1 and r_1 and another edge between l_2 and r_2 and so on).

3. For each pair (l_i, r_i) an edge is created. This is done by calling the following key (for the edge specification \rightarrow , other keys are executed for the other kinds of specifications):

`/tikz/graphs/new ->={⟨left node⟩}{⟨right node⟩}{⟨edge options⟩}{⟨edge nodes⟩}` (no default)

This key will be called for a \rightarrow edge specification with the following four parameters:

- (a) $⟨left node⟩$ is the name of the “left” node, that is, the name of l_i .
- (b) $⟨right node⟩$ is the name of the right node.
- (c) $⟨edge options⟩$ are the accumulated options from all calls of `/tikz/graph/edges` in groups that surround the edge specification.
- (d) $⟨edge nodes⟩$ is text like `node {A} node {B}` that specifies some nodes that should be put as labels on the edge using TikZ’s implicit positioning mechanism.

By default, the key executes the following code:

```
\path [->,every new ->]
  (<left node>)\tikzgraphleftanchor) edge [⟨edge options⟩] ⟨edge nodes⟩
  (<right node>)\tikzgraphrightanchor);
```

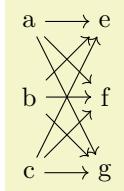
You are welcome to change the code underlying the key.

`/tikz/every new ->` (style, no value)

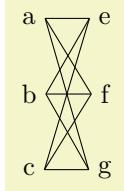
This key gets executed by default for a `new ->`.

`/tikz/graphs/left anchor=(anchor)` (no default)

This anchor is used for the node that is to the left of an edge specification. Setting this anchor to the empty string means that no special anchor is used (which is the default). The $⟨anchor⟩$ is stored in the macro `\tikzgraphleftanchor` with a leading dot.



```
\usetikzlibrary {graphs}
\tikz \graph {
  {a,b,c} -> [complete bipartite] {e,f,g}
};
```



```
\usetikzlibrary {graphs}
\tikz \graph [left anchor=east, right anchor=west] {
  {a,b,c} -- [complete bipartite] {e,f,g}
};
```

`/tikz/graphs/right anchor=(anchor)` (no default)

Works like `left anchor`, only for `\tikzgraphrightanchor`.

For the other three kinds of edge specifications, the following keys will be called:

`/tikz/graphs/new -=={⟨left node⟩}{⟨right node⟩}{⟨edge options⟩}{⟨edge nodes⟩}` (no default)

This key is called for $--$ with the same parameters as above. The only difference in the definition is that in the `\path` command the \rightarrow gets replaced by $-$.

`/tikz/every new -` (style, no value)

`/tikz/graphs/new <->={⟨left node⟩}{⟨right node⟩}{⟨edge options⟩}{⟨edge nodes⟩}` (no default)

Called for $<->$ with the same parameters as above. The \rightarrow is replaced by $<-$.

`/tikz/every new <->` (style, no value)

```
/tikz/graphs/new <-={⟨left node⟩}{⟨right node⟩}{⟨edge options⟩}{⟨edge nodes⟩}      (no default)
Called for <- with the same parameters as above.4
/tikz/every new <-
/tikz/graphs/new -!-={⟨left node⟩}{⟨right node⟩}{⟨edge options⟩}{⟨edge nodes⟩}      (no default)
Called for -!- with the same parameters as above. Does nothing by default.
```

Here is an example that shows the default rendering of the different edge specifications:

```
\usetikzlibrary {graphs}
\begin{tikzpicture}
\graph [branch down=5mm] {
    a -> b;
    c -- d;
    e <-- f;
    g <--> h;
    i -!- j;
};

```

19.3.4 Syntax of Node Specifications

Node specifications are the basic building blocks of a graph specification. There are three different possible kinds of node specifications, each of which has a different syntax:

Direct Node Specification

"⟨node name⟩"/"⟨text⟩" [⟨options⟩]

(note that the quotation marks are optional and only needed when the ⟨node name⟩ contains special symbols)

Reference Node Specification

(⟨node name or node set name⟩)

Group Node Specification

⟨group specification⟩

The rule for determining which of the possible kinds is meant is as follows: If the node specification starts with an opening parenthesis, a reference node specification is meant; if it starts with an opening curly brace, a group specification is meant; and in all other cases a direct node specification is meant.

Direct Node Specifications. If after reading the first symbol of a node specification is has been detected to be *direct*, TikZ will collect all text up to the next edge specification and store it as the ⟨node name⟩; however, square brackets are used to indicate options and a slash ends the ⟨node name⟩ and start a special ⟨text⟩ that is used as a rendering text instead of the original ⟨node name⟩.

Due to the way the parsing works and due to the restrictions on node names, most special characters are forbidding inside the ⟨node name⟩, including commas, semicolons, hyphens, braces, dots, parentheses, slashes, dashes, and more (but spaces, single underscores, and the hat character `#` are allowed). To use special characters in the name of a node, you can optionally surround the ⟨node name⟩ and/or the ⟨text⟩ by quotation marks. In this case, you can use all of the special symbols once more. The details of what happens, exactly, when the ⟨node name⟩ is surrounded by quotation marks is explained later; surrounding the ⟨text⟩ by quotation marks has essentially the same effect as surrounding it by curly braces.

Once the node name has been determined, it is checked whether the same node name was already used inside the current graph. If this is the case, then we say that the already existing node is *referenced*; otherwise we say that the node is *fresh*.

```
\usetikzlibrary {graphs}
\begin{tikzpicture}
\graph {
    a -> b; % both are fresh
    c -> a; % only c is fresh, a is referenced
};

```

⁴You might wonder why this key is needed: It seems more logical at first sight to just call `newedge[directed]` with swapped first parameters. However, a positioning algorithm might wish to take the fact into account that an edge is “backward” rather than “forward” in order to improve the layout. Also, different arrow heads might be used.

This behaviour of deciding whether a node is fresh or referenced can, however, be modified by using the following keys:

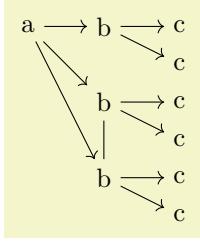
`/tikz/graphs/use existing nodes=<true or false>` (default `true`)

When this key is set to `true`, all nodes will be considered to be referenced, no node will be fresh. This option is useful if you have already created all the nodes of a graph prior to using the `graph` command and you now only wish to connect the nodes. It also implies that an error is raised if you reference a node which has not been defined previously.

`/tikz/graphs/fresh nodes=<true or false>` (default `true`)

When this key is set to `true`, all nodes will be considered to be fresh. This option is useful when you create for instance a tree with many identical nodes.

When a node name is encountered that was already used previously, a new name is chosen as follows: An apostrophe (') is appended repeatedly until a node name is found that has not yet been used:

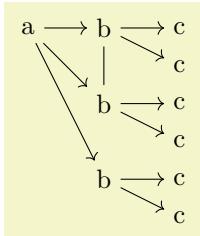


```
\usetikzlibrary {graphs}
\tikz \graph [branch down=5mm] {
  [fresh nodes]
  a -> {
    b -> {c, c},
    b -> {c, c},
    b -> {c, c},
  }
},
b' -- b"
};
```

`/tikz/graphs/number nodes=<start number>` (default 1)

When this key is used in a scope, each encountered node name will get appended a new number, starting with `<start>`. Typically, this ensures that all node names are different. Between the original node name and the appended number, the setting of the following will be inserted:

`/tikz/graphs/number nodes sep=<text>` (no default, initially space)



```
\usetikzlibrary {graphs}
\tikz \graph [branch down=5mm] {
  [number nodes]
  a -> {
    b -> {c, c},
    b -> {c, c},
    b -> {c, c},
  }
},
b 2 -- b 5
};
```

When a fresh node has been detected, a new node is created in the inside a protecting scope. For this, the current placement strategy is asked to compute a default position for the node, see Section 19.9 for details. Then, the command

```
\node (<full node name>) [<node options>] {<text>};
```

is called. The different parameters are as follows:

- The `<full node name>` is normally the `<node name>` that has been determined as described before. However, there are two exceptions:

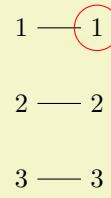
First, if the `<node name>` is empty (which happens when there is no `<node name>` before the slash), then a fresh internal node name is created and used as `<full node name>`. This name is guaranteed to be different from all node names used in this or any other graph. Thus, a direct node starting with a slash represents an anonymous fresh node.

Second, you can use the following key to prefix the `<node name>` inside the `<full node name>`:

/tikz/graphs/name=⟨text⟩

(no default)

This key prepends the ⟨text⟩, followed by a separating symbol (a space by default), to all ⟨node name⟩s inside a ⟨full node name⟩. Repeated calls of this key accumulate, leading to ever-longer “name paths”:



```
\usetikzlibrary {graphs}
\begin{tikzpicture}
\graph {
  { [name=first] 1, 2, 3} --
  { [name=second] 1, 2, 3}
};
\draw [red] (second 1) circle [radius=3mm];
\end{tikzpicture}
```

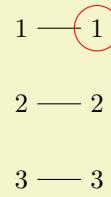
Note that, indeed, in the above example six nodes are created even though the first and second set of nodes have the same ⟨node name⟩. The reason is that the full names of the six nodes are all different. Also note that only the ⟨node name⟩ is used as the node text, not the full name. This can be changed as described later on.

This key can be used repeatedly, leading to ever longer node names.

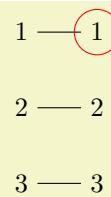
/tikz/graphs/name separator=⟨symbols⟩

(no default, initially \space)

Changes the symbol that is used to separate the ⟨text⟩ from the ⟨node name⟩. The default is \space, resulting in a space.



```
\usetikzlibrary {graphs}
\begin{tikzpicture}
\graph [name separator=] { % no separator
  { [name=first] 1, 2, 3} --
  { [name=second] 1, 2, 3}
};
\draw [red] (second1) circle [radius=3mm];
\end{tikzpicture}
```



```
\usetikzlibrary {graphs}
\begin{tikzpicture}
\graph [name separator=-] {
  { [name=first] 1, 2, 3} --
  { [name=second] 1, 2, 3}
};
\draw [red] (second-1) circle [radius=3mm];
\end{tikzpicture}
```

- The ⟨node options⟩ are

1. The options that have accumulated in calls to `nodes` from the surrounding scopes.
2. The local ⟨options⟩.

The options are executed with the path prefix `/tikz/graphs`, but any unknown key is executed with the prefix `/tikz`. This means, in essence, that some esoteric keys are more difficult to use inside the options and that any key with the prefix `/tikz/graphs` will take precedence over a key with the prefix `/tikz`.

- The ⟨text⟩ that is passed to the `\node` command is computed as follows: First, you can use the following key to directly set the ⟨text⟩:

/tikz/graphs/as=⟨text⟩

(no default)

The ⟨text⟩ is used as the text of the node. This allows you to provide a text for the node that differs arbitrarily from the name of the node.

x — y₅ → a–b

```
\usetikzlibrary {graphs}
\tikz \graph { a [as=$x$] -- b [as=$y_5$] -> c [red, as={a--b}] };
```

This key always takes precedence over all of the mechanisms described below.

In case the `as` key is not used, a default text is chosen as follows: First, when a direct node specification contains a slash (or, for historical reasons, a double underscore), the text to the right of the slash (or double underscore) is stored in the macro `\tikzgraphnodetext`; if there is no slash, the `<node name>` is stored in `\tikzgraphnodetext`, instead. Then, the current value of the following key is used as `<text>`:

`/tikz/graphs/typeset=(code)` (no default)

The macro or code stored in this key is used as the `<text>` if the node. Inside the `(code)`, the following macros are available:

`\tikzgraphnodetext`

This macro expands to the `<text>` to the right of the double underscore or slash in a direct node specification or, if there is no slash, to the `<node name>`.

`\tikzgraphnodename`

This macro expands to the name of the current node without the path.

`\tikzgraphnodepath`

This macro expands to the current path of the node. These paths result from the use of the `name` key as described above.

`\tikzgraphnodefullname`

This macro contains the concatenation of the above two.

By default, the typesetter is just set to `\tikzgraphnodetext`, which means that the default text of a node is its name. However, it may be useful to change this: For instance, you might wish that the text of all graph nodes is, say, surrounded by parentheses:

(a) → (b) → (c)

```
\usetikzlibrary {graphs}
\tikz \graph [typeset=(\tikzgraphnodetext)]
{ a -> b -> c };
```

A more advanced macro might take apart the node text and render it differently:

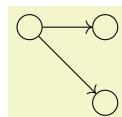
a_1, \dots, n
↓
 b_2, \dots, m
↓
 c_4, \dots, nm

```
\usetikzlibrary {graphs}
\def\mytypesetter{\expandafter\myparser\tikzgraphnodetext\relax}
\def\myparser#1 #2 #3\relax{%
$#1_{\{#2,\dots,#3\}}$}
}
\tikz \graph [typeset=\mytypesetter, grow down]
{ a 1 n -> b 2 m -> c 4 nm };
```

The following styles install useful predefined typesetting macros:

`/tikz/graphs/empty nodes` (no value)

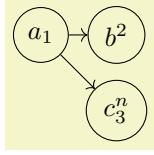
Just sets `typeset` to nothing, which causes all nodes to have an empty text (unless, of course, the `as` option is used):



```
\usetikzlibrary {graphs}
\tikz \graph [empty nodes, nodes={circle, draw}] { a -> {b, c} };
```

`/tikz/graphs/math nodes` (no value)

Sets `typeset` to `\tikzgraphnodetext`, which causes all nodes names to be typeset in math mode:

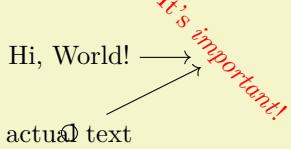


```
\usetikzlibrary {graphs}
\begin{tikzpicture}
\graph [math nodes, nodes={circle, draw}] { a_1 -> {b^2, c_3^n} };

```

If a node is referenced instead of fresh, then this node becomes the node that will be connected by the preceding or following edge specification to other nodes. The *<options>* are executed even for a referenced node, but they cannot be used to change the appearance of the node (because the node exists already). Rather, the *<options>* can only be used to change the logical coloring of the node, see Section 19.7 for details.

Quoted Node Names. When the *<node name>* and/or the *<text>* of a node is surrounded by quotation marks, you can use all sorts of special symbols as part of the text that are normally forbidden:



```
\usetikzlibrary {graphs}
\begin{tikzpicture}
\graph [grow right=2cm] {
    "Hi, World!" --> "It's \emph{important}!"[red,rotate=-45];
    "name"/actual text --> "It's \emph{important}!";
};
\draw (name) circle [radius=3pt];
\end{tikzpicture}
```

In detail, for the following happens when quotation marks are encountered at the beginning of a node name or its text:

- Everything following the quotation mark up to the next single quotation mark is collected into a macro *<collected>*. All sorts of special characters, including commas, square brackets, dashes, and even backslashes are allowed here. Basically, the only restriction is that braces must be balanced.
- A double quotation mark ("") does not count as the “next single quotation mark”. Rather, it is replaced by a single quotation mark. For instance, "He said, ""Hello world."" would be stored inside *<collected>* as He said, "Hello world." However, this rule applies only on the outer-most level of braces. Thus, in

```
"He {said, ""Hello world."}"
```

we would get He {said, "Hello world."} as *<collected>*.

- “The next single quotation mark” refers to the next quotation mark on the current level of braces, so in "hello {" world", the next quotation mark would be the one following world.

Now, once the *<collected>* text has been gathered, it is used as follows: When used as *<text>* (what is actually displayed), it is just used “as is”. When it is used as *<node name>*, however, the following happens: Every “special character” in *<collected>* is replaced by its Unicode name, surrounded by @-signs. For instance, if *<collected>* is Hello, world!, the *<node name>* is the somewhat longer text Hello@COMMA@world@EXCLAMATION MARK@. Admittedly, referencing such a node from outside the graph is cumbersome, but when you use exactly the same *<collected>* text once more, the same *<node name>* will result. The following characters are considered “special”:

```
!$&^~_[]{}()/.-,+*'`!":;,<=>?@#%\{\}
```

These are exactly the Unicode character with a decimal code number between 33 and 126 that are neither digits nor letters.

Reference Node Specifications. A reference node specification is a node specification that starts with an opening parenthesis. In this case, parentheses must surround a *<name>* as in (foo), where foo is the *<name>*. The following will now happen:

1. It is tested whether *<name>* is the name of a currently active *node set*. This case will be discussed in a moment.

2. Otherwise, the `<name>` is interpreted and treated as a referenced node, but independently of whether the node has already been fresh in the current graph or not. In other words, the node must have been defined either already inside the graph (in which case the parenthesis are more or less superfluous) or it must have been defined outside the current picture.

The way the referenced node is handled is the same way as for a direct node that is a referenced node.

If the node does not already exist, an error message is printed.

Let us now have a look at node sets. Inside a `\tikzpicture` you can locally define a *node set* by using the following key:

`/tikz/new set=<set name>` (no default)

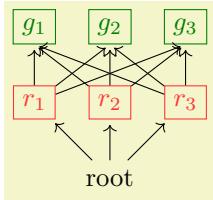
This will setup a node set named `<set name>` within the current scope. Inside the scope, you can add nodes to the node set using the `set` key. If a node set of the same name already exists in the current scope, it will be reset and made empty for the current scope.

Note that this command has the path `/tikz` and is normally used *outside* the `graph` command.

`/tikz/set=<set name>` (no default)

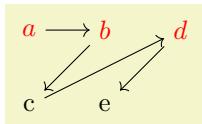
This key can be used as an option with a `node` command. The `<set name>` must be the name of a node set that has previously been created inside some enclosing scope via the `new set` key. The effect is that the current node is added to the node set.

When you use a `graph` command inside a scope where some node set called `<set name>` is defined, then inside this `graph` command you use `(<set name>)` to reference *all* of the nodes in the node set. The effect is the same as if instead of the reference to the set name you had created a group specification containing a list of references to all the nodes that are part of the node set.



```
\usetikzlibrary {graphs}
\begin{tikzpicture} [new set=red, new set=green, shorten >=2pt]
\foreach \i in {1,2,3} {
  \node [draw, red!80, set=red] (r\i) at (\i,1) {$r_{\i}$};
  \node [draw, green!50!black, set=green] (g\i) at (\i,2) {$g_{\i}$};
}
\graph {
  root [xshift=2cm] --> (red)
  (red) --> [complete bipartite, right anchor=south]
  (green)
};
\end{tikzpicture}
```

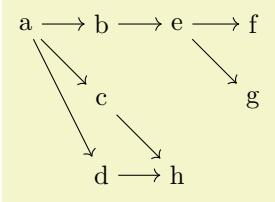
There is an interesting caveat with referencing node sets: Suppose that at the beginning of a graph you just say `(foo)`; where `foo` is a set name. Unless you have specified special options, this will cause the following to happen: A group is created whose members are all the nodes of the node set `foo`. These nodes become referenced nodes, but otherwise nothing happens since, by default, the nodes of a group are not connected automatically. However, the referenced nodes have now been referenced inside the graph, you can thus subsequently access them as if they had been defined inside the graph. Here is an example showing how you can create nodes outside a `graph` command and then connect them inside as if they had been declared inside:



```
\usetikzlibrary {graphs}
\begin{tikzpicture} [new set=import nodes]
\begin{scope} [nodes={set=import nodes}] % make all nodes part of this set
  \node [red] (a) at (0,1) {$a$};
  \node [red] (b) at (1,1) {$b$};
  \node [red] (d) at (2,1) {$d$};
\end{scope}

\graph {
  (import nodes); % "import" the nodes
  a -> b -> c -> d -> e; % only c and e are new
};
\end{tikzpicture}
```

Group Node Specifications. At a place where a node specification should go, you can also instead provide a group specification. Since nodes specifications are part of chain specifications, which in turn are part of group specifications, this is a recursive definition.



```
\usetikzlibrary {graphs}
\begin{tikzpicture}
\graph [group sep=1cm]
    {
        a --> b --> e --> f
        a -.-> c
        a -.-> d
        c -.-> d
        e -.-> g
        f -.-> g
        f -.-> h
    };

```

As can be seen in the above example, when two groups of nodes are connected via an edge specification, it is not immediately obvious which connecting edges are added. This is detailed in Section 19.7.

19.3.5 Specifying Tries

In computer science, a *trie* is a special kind of tree, where for each node and each symbol of an alphabet, there is at most one child of the node labeled with this symbol.

The `trie` key is useful for drawing tries, but it can also be used in other situations. What it does, essentially, is to prepend the node names of all nodes *before* the current node of the current chain to the node's name. This will often make it easier or more natural to specify graphs in which several nodes have the same label.

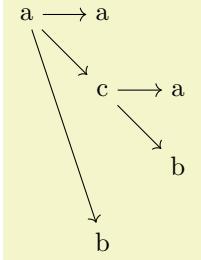
`/tikz/graphs/trie=<true or false>`

(default `true`, initially `false`)

If this key is set to `true`, after a node has been created on a chain, the `name` key is executed with the node's *<node name>*. Thus, all nodes later on this chain have the “path” of nodes leading to this node as their name. This means, in particular, that

1. two nodes of the same name but in different parts of a chain will be different,
2. while if another chain starts with the same nodes, no new nodes get created.

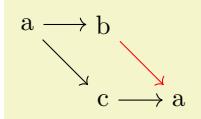
In total, this is exactly the behaviour you would expect of a trie:



```
\usetikzlibrary {graphs}
\begin{tikzpicture}
\graph [trie]
    {
        a --> a
        a -.-> c
        a -.-> b
        c --> a
        c --> b
    };

```

You can even “reiterate” over a path in conjunction with the `simple` option. However, in this case, the default placement strategies will not work and you will need options like `layered layout` from the graph drawing libraries, which need LuaTeX. You can also use the `trie` key locally and later reference nodes using their full name:



```
\usetikzlibrary {graphs}
\begin{tikzpicture}
\graph [trie, simple]
    {
        a --> b
        a -.-> c
        c --> a
    };

```

19.4 Quick Graphs

The graph syntax is powerful, but this power comes at a price: parsing the graph syntax, which is done by TeX, can take some time. Normally, the parsing is fast enough that you will not notice it, but it can be bothersome when you have graphs with hundreds of nodes as happens frequently when nodes are generated

algorithmically by some other program. Fortunately, when another program generated a graph specification, we typically do not need the full power of the graph syntax. Rather, a small subset of the graph syntax would suffice that allows to specify nodes and edges. For these reasons, there is a special “quick” version of the graph syntax.

Note, however, that using this syntax will usually at most halve the time needed to parse a graph. Thus, it really mostly makes sense in conjunction with large, algorithmically generated graphs.

/tikz/graphs/quick

(no value)

When you provide this key with a graph, the syntax of graph specifications gets restricted. You are no longer allowed to use certain features of the graph syntax; but all features that are still allowed are also allowed in the same way when you do not provide the `quick` option. Thus, leaving out the `quick` option will never hurt.

Since the syntax is so severely restricted, it is easier to explain which aspects of the graph syntax *will* still work:

1. A quick graph consists of a sequence of either nodes, edges sequences, or groups. These are separated by commas or semicolons.

2. Every node is of the form

`"<node name>" / "<node text>" [<options>]`

The quotation marks are mandatory. The part `"/<node text>"` may be missing, in which case the node name is used as the node text. The `<options>` may also be missing. The `<node name>` may not contain any “funny” characters (unlike in the normal graph command).

3. Every chain is of the form

`<node spec> <connector> <node spec> <connector> ... <connector> <node spec>;`

Here, the `<node spec>` are node specifications as described above, the `<connector>` is one of the four connectors `->`, `<-`, `--`, and `<->` (the connector `-!-` is not allowed since the `simple` option is also not allowed). Each connector may be followed by options in square brackets. The semicolon may be replaced by a comma.

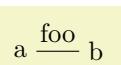
4. Every group is of the form

`{ [<options>] <chains and groups>} ;`

The `<options>` are compulsory. The semicolon can, again, be replaced by a comma.

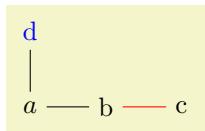
5. The `number nodes` option will work as expected.

Here is a typical way this syntax might be used:



```
\usepackage{tikz}
\usetikzlibrary{graphs, quotes}
\begin{tikzpicture}
\graph [quick] {
    "a" --["foo"] --> "b";
}

```



```
\usepackage{tikz}
\usetikzlibrary{graphs}
\begin{tikzpicture}
\graph [quick] {
    "a"/"$a$" -- "b"["x=1"] --> "c"["x=2"];
    { [nodes=blue] "a" -- "d"["y=1"]; };
}

```

Let us now have a look at the most important things that will *not* work when the `quick` option is used:

- Connecting a node and a group as in `a->{b,c}`.
- Node names without quotation marks as in `a--b`.
- Everything described in subsequent subsections, which includes subgraphs (graph macros), graph sets, graph color classes, anonymous nodes, the `fresh nodes` option, sublayouts, simple graphs, edge annotations.
- Placement strategies – you either have to define all node positions explicitly using `at=` or `x=` and `y=` or you must use a graph drawing algorithm like `layered layout`.

19.5 Simple Versus Multi-Graphs

The `graphs` library allows you to construct both simple graphs and multi-graphs. In a simple graph there can be at most one edge between any two vertices, while in a multi-graph there can be multiple edges (hence the name). The two keys `multi` and `simple` allow you to switch (even locally inside on of the graph's scopes) between which kind of graph is being constructed. By default, the `graph` command produces a multi-graph since these are faster to construct.

/tikz/graphs/multi (no value)

When this edge is set for a whole graph (which is the default) or just for a group (which is useful if the whole graph is simple in general, but a part is a multi-graph), then when you specify an edge between two nodes several times, several such edges get created:



```
\usetikzlibrary{graphs}
\begin{tikzpicture}
\graph [multi] { a ->[bend left, red] b;
  a ->[bend right, blue] b;
};
\end{tikzpicture}
```

In case `multi` is used for a scope inside a larger scope where the `simple` option is specified, then inside the local `multi` scope edges are immediately created and they are completely ignored when it comes to deciding which kind of edges should be present in the surrounding simple graph. From the surrounding scope's point of view it is as if the local `multi` graph contained no edges at all.

This means, in particular, that you can use the `multi` option with a single edge to “enforce” this edge to be present in a simple graph.

/tikz/graphs/simple (no value)

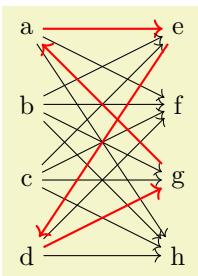
In contrast a multi-graph, in a simple graph, at most one edge gets created for every pair of vertices:



```
\usetikzlibrary{graphs}
\begin{tikzpicture}
\graph [simple] {
  a ->[bend left, red] b;
  a ->[bend right, blue] b;
};
\end{tikzpicture}
```

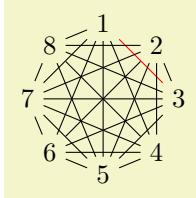
As can be seen, the second edge “wins” over the first edge. The general rule is as follows: In a simple graph, whenever an edge between two vertices is specified multiple times, only the very last specification and its options will actually be executed.

The real power of the `simple` option lies in the fact that you can first create a complicated graph and then later redirect and otherwise modify edges easily:



```
\usetikzlibrary{graphs}
\begin{tikzpicture}
\graph [simple, grow right=2cm] {
  {a,b,c,d} ->[complete bipartite] {e,f,g,h};
  {edges={red, thick}} a -> e -> d -> g -> a ;
};
\end{tikzpicture}
```

One particularly interesting kind of edge specification for a simple graph is `-!-`. Recall that this is used to indicate that “no edge” should be added between certain nodes. In a multi-graph, this key usually has no effect (unless the key `new -!-` has been redefined) and is pretty superfluous. In a simple graph, however, it counts as an edge kind and you can thus use it to remove an edge that been added previously:



```
\usetikzlibrary {graphs.standard}
\begin{tikzpicture}
\graph [simple] {
    subgraph K_n [n=8, clockwise];
        % Get rid of the following edges:
        1 !-- 2;
        3 !-- 4;
        6 !-- 8;
        % And make one edge red:
        1 --[red] 3;
    };

```

Creating a graph such as the above in other fashions is pretty awkward.

For every unordered pair $\{u, v\}$ of vertices at most one edge will be created in a simple graph. In particular, when you say $a \rightarrow b$ and later also $a \leftarrow b$, then only the edge $a \leftarrow b$ will be created. Similarly, when you say $a \rightarrow b$ and later $b \rightarrow a$, then only the edge $b \rightarrow a$ will be created.

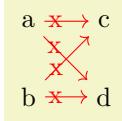
The power of the `simple` command comes at a certain cost: As the graph is being constructed, a (sparse) array is created that keeps track for each edge of the last edge being specified. Then, at the end of the scope containing the `simple` command, for every pair of vertices the edge is created. This is implemented by two nested loops iterating over all possible pairs of vertices – which may take quite a while in a graph of, say, 1000 vertices. Internally, the `simple` command is implemented as an operator that adds the edges when it is called, but this should be unimportant in normal situations.

19.6 Graph Edges: Labeling and Styling

When the `graphs` library creates an edge between two nodes in a graph, the appearance (called “styling” in TikZ) can be specified in different ways. Sometimes you will simply wish to say “the edges between these two groups of node should be red”, but sometimes you may wish to say “this particular edge going into this node should be red”. In the following, different ways of specifying such styling requirements are discussed. Note that adding labels to edges is, from TikZ’s point of view, almost the same as styling edges, since they are also specified using options.

19.6.1 Options For All Edges Between Two Groups

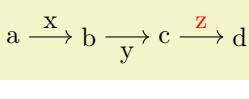
When you write $\dots \rightarrow[\text{options}] \dots$ somewhere inside your graph specification, this typically cause one or more edges to be created between the nodes in the chain group before the `\rightarrow` and the nodes in the chain group following it. The `options` are applied to all of them. In particular, if you use the `quotes` library and you write some text in quotes inside the `options`, this text will be added as a label to each edge:



```
\usetikzlibrary {graphs,quotes}
\begin{tikzpicture}
\graph [edge quotes=near start] {
    {a, b} -> [red, "x", complete bipartite] {c, d};
};

```

As documented in the `quotes` library in more detail, you can easily modify the appearance of edge labels created using the `quotes` syntax by adding options after the closing quotes:



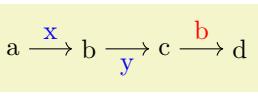
```
\usetikzlibrary {graphs,quotes}
\begin{tikzpicture}
\graph {
    a ->["x"] b ->["y"] c ->["z" red] d;
};

```

The following options make it easy to setup the styling of nodes created in this way:

`/tikz/graphs/edge quotes=<options>` (no default)

A shorthand for setting the style `every edge quotes` to `<options>`.



```
\usetikzlibrary {graphs,quotes}
\begin{tikzpicture}
\graph [edge quotes={blue,auto}] {
    a ->["x"] b ->["y"] c ->["z" red] d;
};

```

`/tikz/graphs/edge quotes center`

(no value)

A shorthand for `edge quotes` to `anchor=center`.

a \xrightarrow{x} b \xrightarrow{y} c \xrightarrow{z} d

```
\usetikzlibrary {graphs,quotes}
\begin{tikzpicture}
\graph [edge quotes center] {
    a ->["x"] b ->["y"] c ->["z" red] d;
};

```

`/tikz/graphs/edge quotes mid`

(no value)

A shorthand for `edge quotes` to `anchor=mid`.

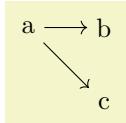
a \xrightarrow{x} b \xrightarrow{y} c \xrightarrow{z} d

```
\usetikzlibrary {graphs,quotes}
\begin{tikzpicture}
\graph [edge quotes mid] {
    a ->["x"] b ->["y"] c ->["z" red] d;
};

```

19.6.2 Changing Options For Certain Edges

Consider the following tree-like graph:

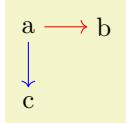


```
\usetikzlibrary {graphs}
\begin{tikzpicture}
\graph {
    a -> {b,c};
};

```

Suppose we wish to specify that the edge from `a` to `b` should be red, while the edge from `a` to `c` should be blue. The difficulty lies in the fact that *both* edges are created by the single `->` operator and we can only add one of these option `red` or `blue` to the operator.

There are several ways to solve this problem. First, we can simply split up the specification and specify the two edges separately:

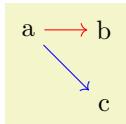


```
\usetikzlibrary {graphs}
\begin{tikzpicture}
\graph {
    a -> [red] b;
    a -> [blue] c;
};

```

While this works quite well, we can no longer use the nice chain group syntax of the `graphs` library. For the rather simple graph `a->{b,c}` this is not a big problem, but if you specify a tree with, say, 30 nodes it is really worthwhile being able to specify the tree “in its natural form in the T_EX code” rather than having to list all of the edges explicitly. Also, as can be seen in the above example, the node placement is changed, which is not always desirable.

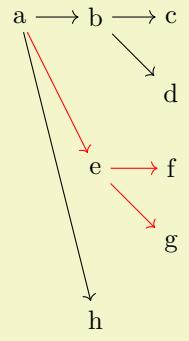
One can sidestep this problem using the `simple` option: This option allows you to first specify a graph and then, later on, replace edges by other edges and, thereby, provide new options:



```
\usetikzlibrary {graphs}
\begin{tikzpicture}
\graph [simple] {
    a -> {b,c};
    a -> [red] b;
    a -> [blue] c;
};

```

The first line is the original specification of the tree, while the following two lines replace some edges of the tree (in this case, all of them) by edges with special options. While this method is slower and in the above example creates even longer code, it is very useful if you wish to, say, highlight a path in a larger tree: First specify the tree normally and, then, “respecify” the path or paths with some other edge options in force. In the following example, we use this to highlight a whole subtree of a larger tree:



```
\usetikzlibrary {graphs}
\begin{tikz} \graph [simple] {
    % The larger tree, no special options in force
    a -> {
        b -> {c,d},
        e -> {f,g},
        h
    },
    { [edges=red] % Now highlight a part of the tree
        a -> e -> {f,g}
    }
};
```

19.6.3 Options For Incoming and Outgoing Edges

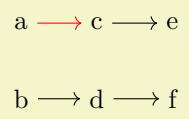
When you use the syntax `... ->[options] ...` to specify options, you specify options for the “connections between two sets of nodes”. In many cases, however, it will be more natural to specify options “for the edges lead to or coming from a certain node” and you will want to specify these options “at the node”. Returning to the example of the graph `a->{b,c}` where we want a red edge between `a` and `b` and a blue edge between `a` and `c`, this could also be phrased as follows: “Make the edge leading to `b` red and make the edge leading to `c` blue”.

For this situation, the `graphs` library offers a number of special keys, which are documented in the following. However, most of the time you will not use these keys directly, but, rather, use a special syntax explained in Section 19.6.4.

`/tikz/graphs/target edge style=<options>`

(no default)

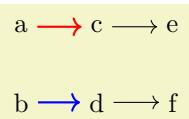
This key can (only) be used with a `node` inside a graph specification. When used, the `<options>` will be added to every edge that is created by a connector like `->` in which the node is a `target`. Consider the following example:



```
\usetikzlibrary {graphs}
\begin{tikz} \graph {
    { a, b } ->
    { c [target edge style=red], d } ->
    { e, f }
};
```

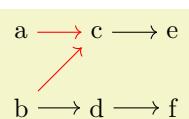
In the example, only when the edge from `a` to `c` is created, `c` is the “target” of the edge. Thus, only this edge becomes red.

When an edge already has options set directly, the `<options>` are executed after these direct options, thus, they “overrule” them:



```
\usetikzlibrary {graphs}
\begin{tikz} \graph {
    { a, b } -> [blue, thick]
    { c [target edge style=red], d } ->
    { e, f }
};
```

The `<options>` set in this way will stay attached to the node, so also for edges created later on that lead to the node will have these options set:

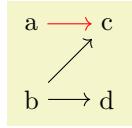


```
\usetikzlibrary {graphs}
\begin{tikz} \graph {
    { a, b } ->
    { c [target edge style=red], d } ->
    { e, f },
    b -> c
};
```

Multiple uses of this key accumulate. However, you may sometimes also wish to “clear” these options for a key since at some later point you no longer wish the `<options>` to be added when some further edges are added. This can be achieved using the following key:

`/tikz/graphs/target edge clear` (no value)

Clears all *(options)* for edges with the node as a target and also edge labels (see below) for this node.

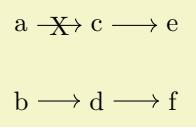


```
\usetikzlibrary {graphs}
\begin{tikzpicture}
\graph {
    {a, b} ->
    {c [target edge style=red], d},
    b -> c[target edge clear]
};

```

`/tikz/graphs/target edge node=<node specification>` (no default)

This key works like `target edge style`, only the *<node specification>* will not be added as options to any newly created edges with the current node as their target, but rather it will be added as a node specification.



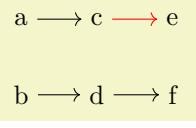
```
\usetikzlibrary {graphs}
\begin{tikzpicture}
\graph {
    {a, b} ->
    {c [target edge node=node[X]], d} ->
    {e, f}
};

```

As for `target edge style` multiple uses of this key accumulate and the key `target edge clear` will (also) clear all target edge nodes that have been set for a node earlier on.

`/tikz/graphs/source edge style=<options>` (no default)

Works exactly like `target edge style`, only now the *(options)* are only added when the node is a source of a newly created edge:



```
\usetikzlibrary {graphs}
\begin{tikzpicture}
\graph {
    {a, b} ->
    {c [source edge style=red], d} ->
    {e, f}
};

```

If both for the source and also for the target of an edge *(options)* have been specified, the options are applied in the following order:

1. First come the options from the edge itself.
2. Then come the options contributed by the source node using this key.
3. Then come the options contributed by the target node using `target node style`.



```
\usetikzlibrary {graphs}
\begin{tikzpicture}
\graph {
    a [source edge style=red] ->[green] b [target edge style=blue] % blue wins
};

```

`/tikz/graphs/source edge node=<node specification>` (no default)

Works like `source edge style` and `target edge node`.

`/tikz/graphs/source edge clear=<node specification>` (no default)

Works like `target edge clear`.

19.6.4 Special Syntax for Options For Incoming and Outgoing Edges

The keys `target node style` and its friends are powerful, but a bit cumbersome to write down. For this reason, the `graphs` library introduces a special syntax that is based on what I call the “first-char syntax” of keys. Inside the options of a node inside a graph, the following special rules apply:

- Whenever an option starts with `>`, the rest of the options are passed to `target edge style`. For instance, when you write `a[>red]`, then this has the same effect as if you had written

```
a[target edge style={red}]
```

- Whenever an option starts with `<`, the rest of the options are passed to `source edge style`.
- In both of the above case, in case the options following the `>` or `<` sign start with a quote, the created edge label is passed to `source edge node` or `target edge node`, respectively.

This is exactly what you want to happen.

Additionally, the following styles provide shorthands for “clearing” the target and source options:

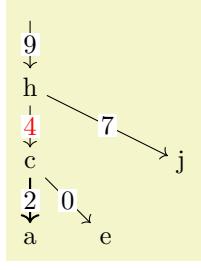
```
/tikz/graphs/clear > (no value)
```

A more easy-to-remember shorthand for `target edge clear`.

```
/tikz/graphs/clear < (no value)
```

A more easy-to-remember shorthand for `source edge clear`.

These mechanisms make it especially easy to create trees in which the edges are labeled in some special way:

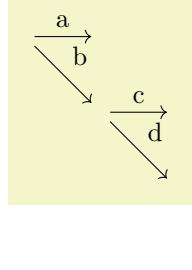


```
\usetikzlibrary {graphs, quotes}
\begin{tikzpicture}
\graph [edge quotes={fill=white, inner sep=1pt},
        grow down, branch right] {
    / -> h [>"9"] -> {
        c [>"4" text=red] -> {
            a [>"2", >thick],
            e [>"0"]
        },
        j [>"7"]
    }
};

```

19.6.5 Placing Node Texts on Incoming Edges

Normally, the text of a node is shown (only) inside the node. In some case, for instance when drawing certain kind of trees, the nodes themselves should not get any text, but rather the edge leading to the node should be labeled as in the following example:



```
\usetikzlibrary {graphs, quotes}
\begin{tikzpicture}
\graph [empty nodes]
{
    root -> {
        a [>"a"],
        b [>"b"] -> {
            c [>"c"],
            d [>"d"]
        }
    }
};

```

As the example shows, it is a bit cumbersome that we have to label the nodes and then specify the same text once more using the incoming edge syntax.

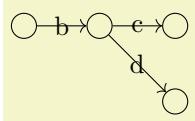
For these cases, it would be better if the text of the node where not used with the node but, rather, be passed directly to the incoming or the outgoing edge. The following styles do exactly this:

```
/tikz/graphs/put node text on incoming edges=<options> (no default)
```

When this key is used with a node or a group, the following happens:

- The command `target edge node={node[<options>]}{\tikzgraphnodetext}` is executed. This means that all incoming edges of the node get a label with the text that would usually be displayed in the node. You can use keys like `math nodes` normally.
- The command `as={}` is executed. This means that the node itself will display nothing.

Here is an example that show how this command is used.



```
\usetikzlibrary {graphs}
\begin{tikzpicture}
\graph [put node text on incoming edges,
        math nodes, nodes={circle, draw}]
{ a -> b -> {c, d} };

```

`/tikz/graphs/put node text on outgoing edges=<options>`

(no default)

Works like the previous key, only with `target` replaced by `source`.

19.7 Graph Operators, Color Classes, and Graph Expressions

TikZ's `graph` command employs a powerful mechanism for adding edges between nodes and sets of nodes. To a graph theorist, this mechanism may be known as a *graph expression*: A graph is specified by starting with small graphs and then applying *operators* to them that form larger graphs and that connect and recolor colored subsets of the graph's node in different ways.

19.7.1 Color Classes

TikZ keeps track of a (*multi*)coloring of the graph as it is being constructed. This does not mean that the actual color of the nodes on the page will be different, rather, in the following we refer to “logical” colors in the way graph theoreticians do. These “logical” colors are only important while the graph is being constructed and they are “thrown away” at the end of the construction. The actual (“physical”) colors of the nodes are set independently of these logical colors.

As a graph is being constructed, each node can be part of one or more overlapping *color classes*. So, unlike what is sometimes called a *legal coloring*, the logical colorings that TikZ keeps track of may assign multiple colors to the same node and two nodes connected by an edge may well have the same color.

Color classes must be declared prior to use. This is done using the following key:

`/tikz/graphs/color class=<color class name>`

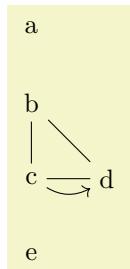
(no default)

This sets up a new color class called `<color class name>`. Nodes and whole groups of nodes can now be colored with `<color class name>`. This is done using the following keys, which become available inside the current scope:

`/tikz/graphs/<color class name>`

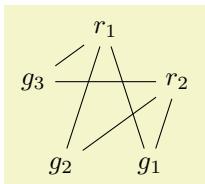
(no value)

This key internally uses the `operator` command to setup an operator that will cause all nodes of the current group to get the “logical color” `<color class name>`. Nodes retain this color in all encompassing scopes, unless it is explicitly changed (see below) or unset (again, see below).



```
\usetikzlibrary {graphs}
\begin{tikzpicture}
\graph [color class=red] {
    [cycle=red] % causes all "logically" red nodes to be connected in
                % a cycle
    a,
    b [red],
    { [red] c -> [bend right] d },
    e
};

```



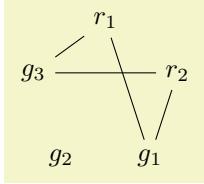
```
\usetikzlibrary {graphs}
\begin{tikzpicture}
\graph [color class=red, color class=green,
        math nodes, clockwise, n=5] {
    [complete bipartite={red}{green}]
    { [red] r_1, r_2 },
    { [green] g_1, g_2, g_3 }
};

```

`/tikz/graphs/not <color class name>`

(no value)

Sets up an operator for the current scope so that all nodes in it loose the color `<color class name>`. You can also use `!(color class name)` as an alias for this key.

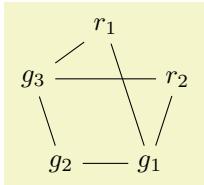


```
\usetikzlibrary {graphs}
\tikz \graph [color class=red, color class=green,
              math nodes, clockwise, n=5] {
    [complete bipartite={red}{green}]
    { [red]   r_1, r_2 },
    { [green] g_1, g_2, g_3 },
    g_2 [not green]
};
```

`/tikz/graphs/recolor <color class name> by=<new color>`

(no default)

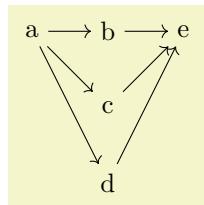
Causes all keys having color `<color class name>` to get `<new color>` instead. They loose having color `<color class name>`, but other colors are not affected.



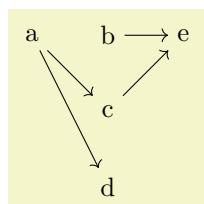
```
\usetikzlibrary {graphs}
\tikz \graph [color class=red, color class=green,
              math nodes, clockwise, n=5] {
    [complete bipartite={red}{green}]
    { [red]   r_1, r_2 },
    { [green] g_1, g_2, g_3 },
    g_2 [recolor green by=red]
};
```

The following color classes are available by default:

- Color class `all`. Every node is part of this class by default. This is useful to access all nodes of a (sub)graph, since you can simply access all nodes of this color class.
- Color classes `source` and `target`. These classes are used to identify nodes that lead “into” a group of nodes and nodes from which paths should “leave” the group. Details on how these colors are assigned are explained in Section 19.7.3. By saying `not source` or `not target` with a node, you can influence how it is connected:



```
\usetikzlibrary {graphs}
\tikz \graph { a -> { b, c, d } -> e };
```



```
\usetikzlibrary {graphs}
\tikz \graph { a -> { b[not source], c, d[not target] } -> e };
```

- Color classes `source'` and `target'`. These are temporary colors that are also explained in Section 19.7.3.

19.7.2 Graph Operators on Groups of Nodes

Recall that the `graph` command constructs graphs recursively from nested `<group specifications>`. Each such `<group specification>` describes a subset of the nodes of the final graph. A *graph operator* is an algorithm that gets the nodes of a group as input and (typically) adds edges between these nodes in some sensible way. For instance, the `clique` operator will simply add edges between all nodes of the group.

`/tikz/graphs/operator=<code>`

(no default)

This key has an effect in three places:

1. It can be used in the `<options>` of a `<direct node specification>`.
2. It can be used in the `<options>` of a `<group specification>`.
3. It can be used in the `<options>` of an `<edge specification>`.

The first case is a special case of the second, since it is treated like a group specification containing a single node. The last case is more complicated and discussed in the next section. So, let us focus on the second case.

Even though the `<options>` of a group are given at the beginning of the `<group specification>`, the `<code>` is only executed when the group has been parsed completely and all its nodes have been identified. If you use the `operator` multiple times in the `<options>`, the effect accumulates, that is, all code passed to the different calls of `operator` gets executed in the order it is encountered.

The `<code>` can do “whatever it wants”, but it will typically add edges between certain nodes. You can configure what kind of edges (directed, undirected, etc.) are created by using the following keys:

`/tikz/graphs/default edge kind=<value>`

(no default, initially `--`)

This key stores one of the five edge kinds `--`, `<-`, `->`, `<->`, and `-!-`. When an operator wishes to create a new edge, it should typically set

```
\tikzgraphsset{new \pkeysvalueof{/tikz/graphs/default edge kind}={...}}
```

While this key can be set explicitly, it may be more convenient to use the abbreviating keys listed below. Also, this key is automatically set to the current value of `<edge specification>` when a joining operator is called, see the discussion of joining operators in Section 19.7.3.

`/tikz/graphs/-`

(no value)

Sets the `default edge kind` to `--`.

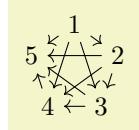


```
\usetikzlibrary {graphs.standard}
\tikz \graph [n=5, clockwise, radius=6mm] {
```

`/tikz/graphs/->`

(no value)

Sets the `default edge kind` to `->`.

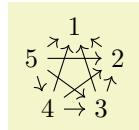


```
\usetikzlibrary {graphs.standard}
\tikz \graph [n=5, clockwise, radius=6mm] {
```

`/tikz/graphs/<-`

(no value)

Sets the `default edge kind` to `<-`.

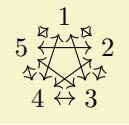


```
\usetikzlibrary {graphs.standard}
\tikz \graph [n=5, clockwise, radius=6mm] {
```

`/tikz/graphs/<->`

(no value)

Sets the `default edge kind` to `<->`.



```
\usetikzlibrary {graphs.standard}
```

```
\tikz \graph [ subgraph K_n [ <-> , n=5, clockwise, radius=6mm ] ];
```

/tikz/graphs/-!-

(no value)

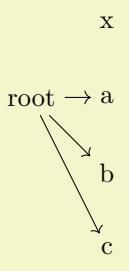
Sets the `default edge kind` to `-!-`.

When the `<code>` of an operator is executed, the following commands can be used to find the nodes that should be connected:

\tikzgraphforeachcolorednode{<color name>}{<macro>}

When this command is called inside `<code>`, the following will happen: TikZ will iterate over all nodes inside the just-specified group that have the color `<color name>`. The order in which they are iterated over is the order in which they appear inside the group specification (if a node is encountered several times inside the specification, only the first occurrence counts). Then, for each node the `<macro>` is executed with the node's name as the only argument.

In the following example we use an operator to connect every node colored `all` inside the subgroup to the node `root`.



```
\usetikzlibrary {graphs}
\def\myconnect#1{\tikzset{graphs/new ->={#1}{}{}{}}
```

```
\begin{tikzpicture}
\node (root) at (-1,-1) {root};
\graph {
  x,
  {
    [operator=\tikzgraphforeachcolorednode{all}{\myconnect}]
    a, b, c
  }
};
\end{tikzpicture}
```

\tikzgraphpreparecolor{<color name>}{<counter>}{<prefix>}

This command is used to “prepare” the nodes of a certain color for random access. The effect is the following: It is counted how many nodes there are having color `<color name>` in the current group and the result is stored in `<counter>`. Next, macros named `<prefix>1`, `<prefix>2`, and so on are defined, that store the names of the first, second, third, and so on node having the color `<color name>`.

The net effect is that after you have prepared a color, you can quickly iterate over them. This is especially useful when you iterate over several colors at the same time.

As an example, let us create an operator that adds a zig-zag path between two color classes:



Naturally, in order to turn the above code into a usable operator, some more code would be needed (like default values and taking care of shores of different sizes).

There are a number of predefined operators, like `clique` or `cycle`, see the reference Section 19.10 for a complete list.

19.7.3 Graph Operators for Joining Groups

When you join two nodes `foo` and `bar` by the edge specification `->`, it is fairly obvious, what should happen: An edge from `(foo)` to `(bar)` should be created. However, suppose we use an edge specification between two node sets like `{a,b,c}` and `{d,e,f}`. In this case, it is not so clear which edges should be created. One might argue that all possible edges from any node in the first set to any node in the second set should be added. On the other hand, one might also argue that only a matching between these two sets should be created. Things get even more muddy when a longer chain of node sets are joined.

Instead of fixing how edges are created between two node sets, TikZ takes a somewhat more general, but also more complicated approach, which can be broken into two parts. In the following, assume that the following chain specification is given:

$\langle spec_1 \rangle \langle edge specification \rangle \langle spec_2 \rangle$

An example might be `{a,b,c} -> {d, e->f}`.

The source and target vertices. Let us start with the question of which vertices of the first node set should be connected to vertices in the second node set.

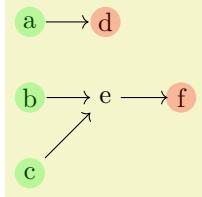
There are two predefined special color classes that are used for this: `source` and `target`. For every group specification, some vertices are colored as `source` vertices and some vertices are `target` vertices (a node can both be a target and a source). Initially, every vertex is both a source and a target, but that can change as we will see in a moment.

The intuition behind source and target vertices is that, in some sense, edges “from the outside” lead into the group via the source vertices and lead out of the group via the target vertices. To be more precise, the following happens:

1. The target vertices of the first group are connected to the source vertices of the second group.
2. In the group resulting from the union of the nodes from $\langle spec_1 \rangle$ and $\langle spec_2 \rangle$, the source vertices are only those from the first group, and the target vertices are only those from the second group.

Let us go over the effect of these rules for the example $\{a, b, c\} \rightarrow \{d, e \rightarrow f\}$. First, each individual node is initially both a **source** and a **target** vertex. Then, in $\{a, b, c\}$ all nodes are still both source and target vertices since just grouping vertices does not change their colors. Now, in $e \rightarrow f$ something interesting happens for the first time: the target vertices of the “group” e (which is just the node e) are connected to the source vertices of the “group” f . This means, that an edge is added from e to f . Then, in the resulting group $e \rightarrow f$ the only source vertex is e and the only target vertex is f . This implies that in the group $\{d, e \rightarrow f\}$ the sources are d and e and the targets are d and f .

Now, in $\{a, b, c\} \rightarrow \{d, e \rightarrow f\}$ the targets of $\{a, b, c\}$ (which are all three of them) are connected to the sources of $\{d, e \rightarrow f\}$ (which are just d and e). Finally, in the whole graph only a, b , and c are sources while only d and f are targets.



```
\usetikzlibrary {graphs}
\def\highlightsource#1{\fill [green, opacity=.25] (#1) circle [radius=2mm]; }
\def\highlighttarget#1{\fill [red, opacity=.25] (#1) circle [radius=2mm]; }
\tikz \graph
  [operator=\tikzgraphforeachcolorednode{source}{\highlightsource},
   operator=\tikzgraphforeachcolorednode{target}{\highlighttarget}]
  { {a,b,c} -> {d, e->f} };
```

The next objective is to make more precise what it means that “the targets of the first graph” and the “sources of the second graph” should be connected. We know already of a general way of connecting nodes of a graph: operators! Thus, we use an operator for this job. For instance, the `complete bipartite` operator adds an edge from every node having a certain color to every node have a certain other color. This is exactly what we need here: The first color is “the color **target** restricted to the nodes of the first graph” and the second color is “the color **source** restricted to the nodes of the second graph”.

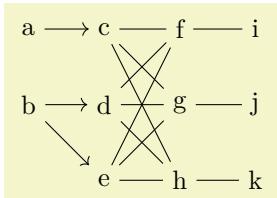
However, we cannot really specify that only nodes from a certain subgraph are meant – the `operator` machinery only operates on all nodes of the current graph. For this reason, what really happens is the following: When the `graph` command encounters $\langle spec_1 \rangle \langle edge specification \rangle \langle spec_2 \rangle$, it first computes and colors the nodes of the first and the second specification independently. Then, the `target` nodes of the first graph are recolored to `target'` and the `source` nodes of the second graph are recolored to `source'`. Then, the two graphs are united into one graph and a *joining operator* is executed, which should add edges between `target'` and `source'`. Once this is done, the colors `target'` and `source'` get erased. Note that in the resulting graph only the `source` nodes from the first graph are still `source` nodes and likewise for the `target` nodes of the second graph.

The joining operators. The job of a joining operator is to add edges between nodes colored `target'` and `source'`. The following rule is used to determine which operator should be chosen for performing this job:

1. If the $\langle edge specification \rangle$ explicitly sets the `operator` key to something non-empty (and also not to `\relax`), then the `(code)` of this `operator` call is used.
2. Otherwise, the current value of the following key is used:

`/tikz/graphs/default edge operator=(key)` (no default, initially `matching and star`)

This key stores the name of a $\langle key \rangle$ that is executed for every $\langle edge specification \rangle$ whose $\langle options \rangle$ do not contain the `operator` key.



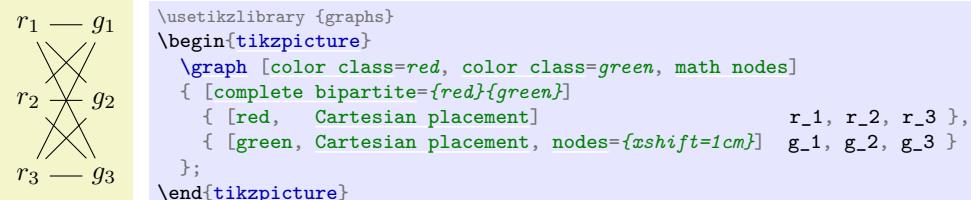
```
\usetikzlibrary {graphs}
\tikz \graph [default edge operator=matching] {
  {a, b} -->[matching and star]
  {c, d, e} --[complete bipartite]
  {f, g, h} --
  {i, j, k}
};
```

A typical joining operator is `complete bipartite`. It takes the names of two color classes as input and adds edges from all vertices of the first class to all vertices of the second class. Now, the trick is that the default value for the `complete bipartite` key is $\{\text{target}'\}\{\text{source}'\}$. Thus, if you just write $\rightarrow[\text{complete bipartite}]$, the same happens as if you had written

```
->[complete bipartite={target'}{source'}]
```

This is exactly what we want to happen. The same default values are also set for other joining operators like `matching` or `butterfly`.

Even though an operator like `complete bipartite` is typically used together with an edge specification, it can also be used as a normal operator together with a group specification. In this case, however, the color classes must be named explicitly:



A list of predefined joining operators can be found in the reference Section 19.10.

The fact that joining operators can also be used as normal operators leads to a subtle problem: A normal operator will typically use the current value of `default edge kind` to decide which kind of edges should be put between the identified vertices, while a joining operator should, naturally, use the kind of edge specified by the *<edge specification>*. This problem is solved as follows: Like a normal operator, a joining operator should also use the current value of `default edge kind` for the edges it produces. The trick is that this will automatically be set to the current *<edge specification>* when the operator explicitly in the *<options>* of the edge specification or implicitly in the `default edge` operator.

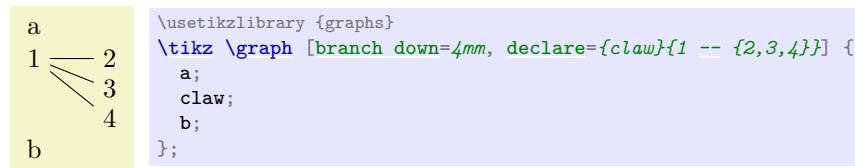
19.8 Graph Macros

A *graph macro* is a small graph that is inserted at some point into the graph that is currently being constructed. There is special support for such graph macros in TikZ. You might wonder why this is necessary – can't one use TeX's normal macro mechanism? The answer is “no”: one cannot insert new nodes into a graph using normal macros because the chains, groups, and nodes are determined prior to macro expansion. Thus, any macro encountered where some node text should go will only be expanded when this node is being named and typeset.

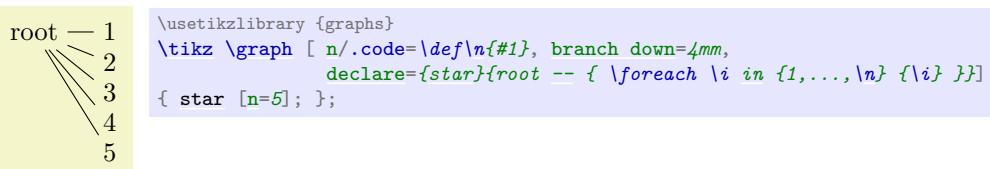
A graph macro is declared using the following key:

```
/tikz/graphs/declare={⟨graph name⟩}{⟨specification⟩} (no default)
```

This key declares that *⟨graph name⟩* can subsequently be used as a replacement for a *⟨node name⟩*. Whenever the *⟨graph name⟩* is used in the following, a graph group will be inserted instead whose content is exactly *⟨specification⟩*. In case *⟨graph name⟩* is used together with some *⟨options⟩*, they are executed prior to inserting the *⟨specification⟩*.

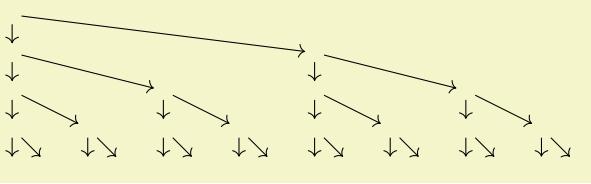


In the next example, we use a key to configure a subgraph:



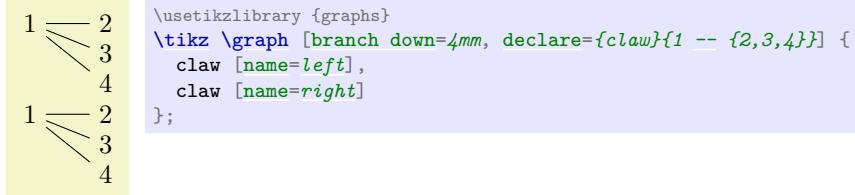
Actually, the `n` key is already defined internally for a similar purpose.

As a last example, let us define a somewhat more complicated graph macro.



```
\usetikzlibrary {graphs}
\newcount\mycount
\tikzgraphsset{
  levels/.store in=\tikzgraphlevel,
  levels=1,
  declare={bintree}{%
    [/utils/exec={%
      \ifnum\tikzgraphlevel=1\relax%
        \def\childtrees{ / }%
      \else%
        \mycount=\tikzgraphlevel%
        \advance\mycount by-1\relax%
        \edef\childtrees{%
          / -> {
            bintree[levels=\the\mycount],
            bintree[levels=\the\mycount]
          }
        }%
      \fi%
    },
    parse/.expand once=\childtrees
  ]
  % Everything is inside the \childtrees...
}
}
\tikz \graph [grow down=5mm, branch right=5mm] { bintree [levels=5] };
```

Note that when you use a graph macro several time inside the same graph, you will typically have to use the `name` option so that different copies of the subgraph are created:



You will find a list of useful graph macros in the reference section, Section 19.10.1.

19.9 Online Placement Strategies

The main job of the `graphs` library is to make it easy to specify which nodes are present in a graph and how they are connected. In contrast, it is *not* the primary job of the library to compute good positions for nodes in a graph – use for instance a `\matrix`, specify good positions “by hand” or use the graph drawing facilities. Nevertheless, some basic support for automatic node placement is provided for simple cases. The `graphs` library will provide you with information about the position of nodes inside their groups and chains.

As a graph is being constructed, a *placement strategy* is used to determine a (reasonably good) position for the nodes as they are created. These placement strategies get some information about what TikZ has already seen concerning the already constructed nodes, but it gets no information concerning the upcoming nodes. Because of this lack of information concerning the future, the strategies need to be what is called an *online strategy* in computer science. (The opposite are *offline strategies*, which get information about the whole graph and all the sizes of the nodes in it. The graph drawing libraries employ such offline strategies.)

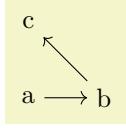
Strategies are selected using keys like `no placement` or `Cartesian placement`. It is permissible to use different strategies inside different parts of a graph, even though the different strategies do not always work together in perfect harmony.

19.9.1 Manual Placement

`/tikz/graphs/no placement`

(no value)

This strategy simply “switches off” the whole placement mechanism, causing all nodes to be placed at the origin by default. You need to use this strategy if you position nodes “by hand”. For this, you can use the `at` key, the `shift` keys:



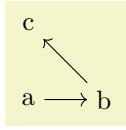
```
\usetikzlibrary {graphs}
\begin{tikz} \graph [no placement]
{
    a[at={\{0:0\}}] -> b[at={\{1,0\}}] -> c[yshift=1cm];
};
```

Since the syntax and the many braces and parentheses are a bit cumbersome, the following two keys might also be useful:

`/tikz/graphs/x=<x dimension>`

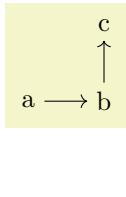
(no default)

When you use this key, it will have the same effect as if you had written `at={<x dimension>, <y dimension>}`, where `<y dimension>` is a value set using the `y` key:



```
\usetikzlibrary {graphs}
\begin{tikz} \graph [no placement]
{
    a[x=0,y=0] -> b[x=1,y=0] -> c[x=0,y=1];
};
```

Note that you can specify an `x` or a `y` key for a whole scope and then vary only the other key:



```
\usetikzlibrary {graphs}
\begin{tikz} \graph [no placement]
{
    a ->
    {
        [x=1] % group option
        b [y=0] -> c[y=1]
    };
};
```

Note that these keys have the path `/tikz/graphs/`, so they will be available inside `graphs` and will not clash with the usual `x` and `y` keys of TikZ, which are used to specify the basic lengths of vectors.

`/tikz/graphs/y=<y dimension>`

(no default)

See above.

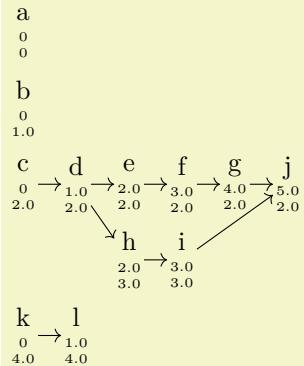
19.9.2 Placement on a Grid

`/tikz/graphs/Cartesian placement`

(no value)

This strategy is the default strategy. It works, roughly, as follows: For each new node on a chain, advance a “logical width” counter and for each new node in a group, advance a “logical depth” counter. When a chain contains a whole group, then the “logical width” taken up by the group is the maximum over the logical widths taken up by the chains inside the group; and symmetrically the logical depth of a chain is the maximum of the depths of the groups inside it.

This slightly confusing explanation is perhaps best exemplified. In the below example, the two numbers indicate the two logical width and depth of each node as computed by the `graphs` library. Just ignore the arcane code that is used to print these numbers.



```
\usetikzlibrary {graphs}
\begin{tikzpicture}
\graph [nodes={align=center, inner sep=1pt}, grow right=7mm,
    typeset={|\tikzgraphnode|[-4pt]
        \tiny\mywidth\[-6pt]\tiny\mydepth},
    placement/compute position/.append code=
        \pgfkeysgetvalue{/tikz/graphs/placement/width}{\mywidth}
        \pgfkeysgetvalue{/tikz/graphs/placement/depth}{\mydepth}]
{
    a,
    b,
    c->d->{
        e->f->g,
        h->i
    }->j,
    k->l
};

```

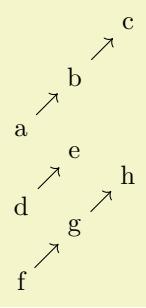
You will find a detailed description of how these logical units are computed, exactly, in Section 19.9.6.

Now, even though we talk about “widths” and “depths” and even though by default a graph “grows” to the right and down, this is by no means fixed. Instead, you can use the following keys to change how widths and heights are interpreted:

`/tikz/graphs/chain shift=<coordinate>`

(no default, initially $(1,0)$)

Under the regime of the `Cartesian placement` strategy, each node is shifted by the current logical width times this $\langle coordinate \rangle$.



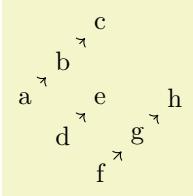
```
\usetikzlibrary {graphs}
\begin{tikzpicture}
\graph [chain shift=(45:1)] {
    a->b->c;
    d->e;
    f->g->h;
};

```

`/tikz/graphs/group shift=<coordinate>`

(no default, initially $(0,-1)$)

Like for `chain shift`, each node is shifted by the current logical depth times this $\langle coordinate \rangle$.



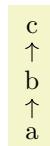
```
\usetikzlibrary {graphs}
\begin{tikzpicture}
\graph [chain shift=(45:7mm), group shift=(-45:7mm)] {
    a->b->c;
    d->e;
    f->g->h;
};

```

`/tikz/graphs/grow up=<distance>`

(default 1)

Sets the `chain shift` to $(0,\langle distance \rangle)$, so that chains “grow upward”. The distance by which the center of each new element is removed from the center of the previous one is $\langle distance \rangle$.

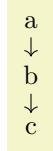


```
\usetikzlibrary {graphs}
\begin{tikzpicture}
\graph [grow up=7mm] { a -> b -> c };

```

`/tikz/graphs/grow down=(distance)` (default 1)

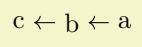
Like `grow up`.



```
\usetikzlibrary {graphs}  
\tikz \graph [grow down=7mm] { a -> b -> c};
```

`/tikz/graphs/grow left=(distance)` (default 1)

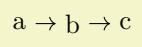
Like `grow up`.



```
\usetikzlibrary {graphs}  
\tikz \graph [grow left=7mm] { a -> b -> c};
```

`/tikz/graphs/grow right=(distance)` (default 1)

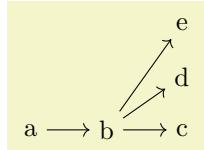
Like `grow up`.



```
\usetikzlibrary {graphs}  
\tikz \graph [grow right=7mm] { a -> b -> c};
```

`/tikz/graphs/branch up=(distance)` (default 1)

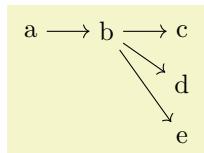
Sets the `group shift` so that groups “branch upward”. The distance by which the center of each new element is removed from the center of the previous one is `<i>distance</i>`.



```
\usetikzlibrary {graphs}  
\tikz \graph [branch up=7mm] { a -> b -> {c, d, e} };
```

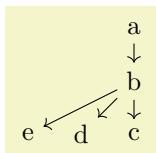
Note that when you draw a tree, the `branch ...` keys specify how siblings (or adjacent branches) are arranged, while the `grow ...` keys specify in which direction the branches “grow”.

`/tikz/graphs/branch down=(distance)` (default 1)



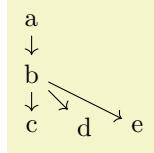
```
\usetikzlibrary {graphs}  
\tikz \graph [branch down=7mm] { a -> b -> {c, d, e} };
```

`/tikz/graphs/branch left=(distance)` (default 1)



```
\usetikzlibrary {graphs}  
\tikz \graph [branch left=7mm, grow down=7mm] { a -> b -> {c, d, e} };
```

`/tikz/graphs/branch right=(distance)` (default 1)



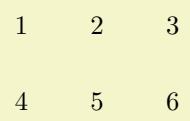
```
\usetikzlibrary {graphs}
\begin{tikzpicture}
\graph [branch right=7mm, grow down=7mm] { a -> b -> {c, d, e} };
\end{tikzpicture}
```

The following keys place nodes in a $N \times M$ grid.

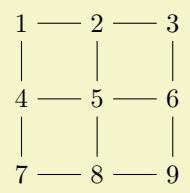
`/tikz/graphs/grid placement`

(no value)

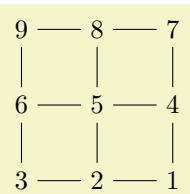
This key works similar to `Cartesian placement`. As for that placement strategy, a node has logical width and depth 1. However, the computed total width and depth are mapped to a $N \times M$ grid. The values of N and M depend on the size of the graph and the value of `wrap after`. The number of columns M is either set to `wrap after` explicitly or computed automatically as $\sqrt{|V|}$. N is the number of rows needed to lay out the graph in a grid with M columns.



```
\usetikzlibrary {graphs.standard}
% An example with 6 nodes, 3 columns and therefore 2 rows
\begin{tikzpicture}
\graph [grid placement] { subgraph I_n [n=6, wrap after=3] };
\end{tikzpicture}
```

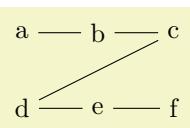


```
\usetikzlibrary {graphs.standard}
% An example with 9 nodes with columns and rows computed automatically
\begin{tikzpicture}
\graph [grid placement] { subgraph Grid_n [n=9] };
\end{tikzpicture}
```



```
\usetikzlibrary {graphs.standard}
% Directions can be changed
\begin{tikzpicture}
\graph [grid placement, branch up, grow left] { subgraph Grid_n [n=9] };
\end{tikzpicture}
```

In case a user-defined graph instead of a pre-defined `subgraph` is to be laid out using `grid placement`, `n` has to be specified explicitly:



```
\usetikzlibrary {graphs}
\begin{tikzpicture}
\graph [grid placement] {
  [n=6, wrap after=3]
  a -- b -- c
  d -- e -- f
  a -- d
  b -- e
  c -- f
};
\end{tikzpicture}
```

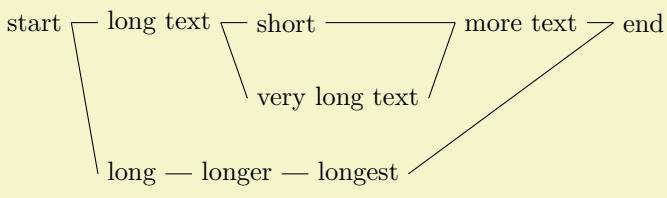
19.9.3 Placement Taking Node Sizes Into Account

Options like `grow up` or `branch right` do not take the sizes of the to-be-positioned nodes into account – all nodes are placed quite “dumbly” at grid positions. It turns out that the `Cartesian placement` can also be used to place nodes in such a way that their height and/or width is taken into account. Note, however, that while the following options may yield an adequate placement in many situations, when you need advanced alignments you should use a `matrix` or advanced offline strategies to place the nodes.

`/tikz/graphs/grow right sep=<distance>`

(default `1em`)

This key has several effects, but let us start with the bottom line: Nodes along a chain are placed in such a way that the left end of a new node is $\langle distance \rangle$ from the right end of the previous node:



```
\usetikzlibrary {graphs}
\tikz \graph [grow right sep, left anchor=east, right anchor=west] {
  start -- {
    long text -- {short, very long text} -- more text,
    long -- longer -- longest
  } -- end
};
```

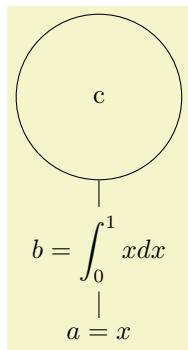
What happens internally is the following: First, the `anchor` of the nodes is set to `west` (or `north west` or `south west`, see below). Second, the logical width of a node is no longer 1, but set to the actual width of the node (which we define as the horizontal difference between the `west` anchor and the `east` anchor) in points. Third, the `chain shift` is set to `(1pt,0pt)`.

`/tikz/graphs/grow left sep=<distance>` (default `1em`)

longest — longer — long

```
\usetikzlibrary {graphs}
\tikz \graph [grow left sep] { long -- longer -- longest };
```

`/tikz/graphs/grow up sep=<distance>` (default `1em`)



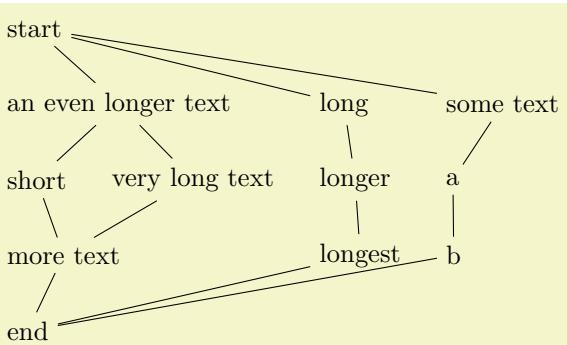
```
\usetikzlibrary {graphs}
\tikz \graph [grow up sep] {
  a / $a=x$ --
  b / {$b=\displaystyle \int_0^1 x \, dx$} --
  c [draw, circle, inner sep=7mm]
};
```

`/tikz/graphs/grow down sep=<distance>` (default `1em`)

As above.

`/tikz/graphs/branch right sep=<distance>` (default `1em`)

This key works like `grow right sep`, only it affects groups rather than chains.

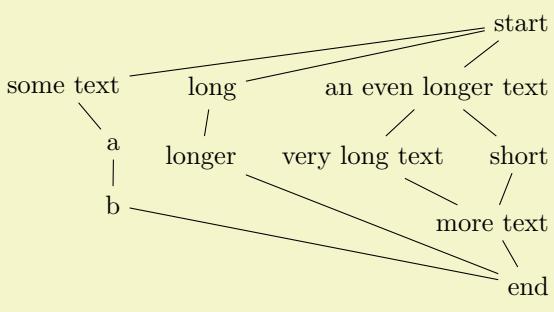


```
\usetikzlibrary {graphs}
\begin{tikz} \graph [grow down, branch right sep] {
  start -- {
    an even longer text -- {short, very long text} -- more text,
    long -- longer -- longest,
    some text -- a -- b
  } -- end
};
```

When both this key and, say, `grow down sep` are set, instead of the `west` anchor, the `north west` anchor will be selected automatically.

`/tikz/graphs/branch left sep=<distance>`

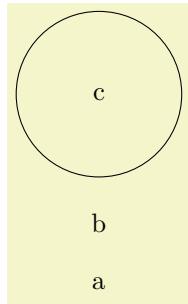
(default `1em`)



```
\usetikzlibrary {graphs}
\begin{tikz} \graph [grow down sep, branch left sep] {
  start -- {
    an even longer text -- {short, very long text} -- more text,
    long -- longer,
    some text -- a -- b
  } -- end
};
```

`/tikz/graphs/branch up sep=<distance>`

(default `1em`)



```
\usetikzlibrary {graphs}
\begin{tikz} \graph [branch up sep] { a, b, c [draw, circle, inner sep=7mm] };
```

`/tikz/graphs/branch down sep=<distance>`

(default `1em`)

19.9.4 Placement On a Circle

The following keys place nodes on circles. Note that, typically, you do not use `circular placement` directly, but rather use one of the two keys `clockwise` or `counterclockwise`.

`/tikz/graphs/circular placement`

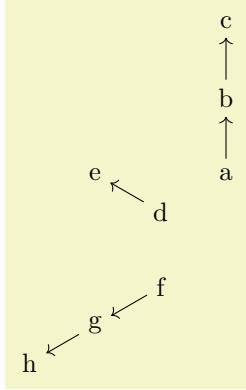
(no value)

This key works quite similar to `Cartesian placement`. As for that placement strategy, a node has logical width and depth 1. However, the computed total width and depth are mapped to polar coordinates rather than Cartesian coordinates.

`/tikz/graphs/chain polar shift=(<angle>:<radius>)`

(no default, initially `(0:1)`)

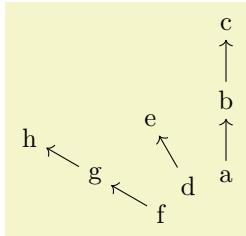
Under the regime of the `circular placement` strategy, each node on a chain is shifted by $(\langle \text{logical width} \rangle \langle \text{angle} \rangle : \langle \text{logical width} \rangle \langle \text{angle} \rangle)$.



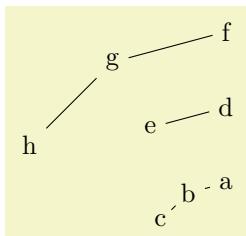
```
\usetikzlibrary {graphs}
\begin{tikz} \graph [circular placement] {
    a -> b -> c;
    d -> e;
    f -> g -> h;
};
```

`/tikz/graphs/group polar shift=(⟨angle⟩):⟨radius⟩)` (no default, initially `(45:0)`)

Like for `group shift`, each node on a chain is shifted by `((⟨logical depth⟩⟨angle⟩):⟨logical depth⟩⟨angle⟩)`.



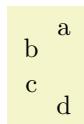
```
\usetikzlibrary {graphs}
\begin{tikz} \graph [circular placement, group polar shift=(30:0)] {
    a -> b -> c;
    d -> e;
    f -> g -> h;
};
```



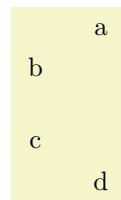
```
\usetikzlibrary {graphs}
\begin{tikz} \graph [circular placement,
    chain polar shift=(30:0),
    group polar shift=(0:1cm)] {
    a -- b -- c;
    d -- e;
    f -- g -- h;
};
```

`/tikz/graphs/radius=⟨dimension⟩)` (no default, initially `1cm`)

This is an initial value that is added to the total computed radius when the polar shift of a node has been calculated. Essentially, this key allows you to set the `⟨radius⟩` of the innermost circle.



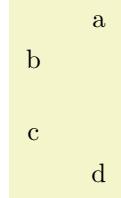
```
\usetikzlibrary {graphs}
\begin{tikz} \graph [circular placement, radius=5mm] { a, b, c, d };
```



```
\usetikzlibrary {graphs}
\begin{tikz} \graph [circular placement, radius=1cm] { a, b, c, d };
```

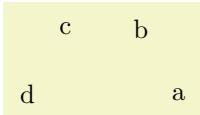
`/tikz/graphs/phase=⟨angle⟩)` (no default, initially `90`)

This is an initial value that is added to the total computed angle when the polar shift of a node has been calculated.



```
\usetikzlibrary {graphs}
\begin{tikzpicture}
\graph [circular placement] { a, b, c, d };

```



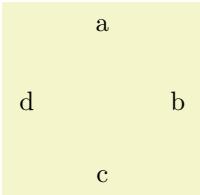
```
\usetikzlibrary {graphs}
\begin{tikzpicture}
\graph [circular placement, phase=0] { a, b, c, d };

```

/tikz/graphs/clockwise=<number>

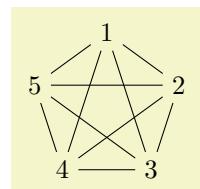
(default \tikzgraphVnum)

This key sets the `group shift` so that if there are exactly `<number>` many nodes in a group, they will form a complete circle. If you do not provide a `<number>`, the current value of `\tikzgraphVnum` is used, which is exactly what you want when you use predefined graph macros like `subgraph K_n`.



```
\usetikzlibrary {graphs}
\begin{tikzpicture}
\graph [clockwise=4] { a, b, c, d };

```



```
\usetikzlibrary {graphs.standard}
\begin{tikzpicture}
\graph [clockwise] { subgraph K_5 [n=5] };

```

/tikz/graphs/counterclockwise=<number>

(default \tikzgraphVnum)

Works like `clockwise`, only the direction is inverted.

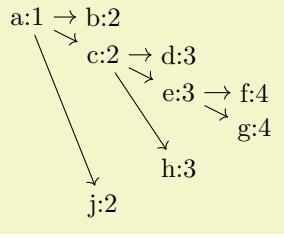
19.9.5 Levels and Level Styles

As a graph is being parsed, the `graph` command keeps track of a parameter called the *level* of a node. Provided that the graph is actually constructed in a tree-like manner, the level is exactly equal to the level of the node inside this tree.

/tikz/graphs/placement/level

(no value)

This key stores a number that is increased for each element on a chain, but gets reset at the end of a group:



```

\usetikzlibrary {graphs}
\tikz \graph [ branch down=5mm, typeset=
  \tikzgraphnodetext:\pgfkeyvalueof{/tikz/graphs/placement/level}]
{
  a -> {
    b,
    c -> {
      d,
      e -> {f,g},
      h
    },
    j
  }
};
```

Unlike the parameters `depth` and `width` described in the next section, the key `level` is always available.

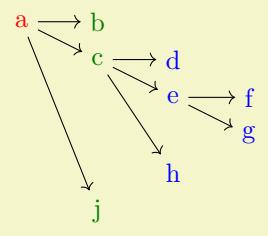
In addition to keeping track of the value of the `level` key, the `graph` command also executes the following keys whenever it creates a node:

`/tikz/graph/level=(level)` (style, no default)

This key gets executed for each newly created node with `(level)` set to the current level of the node. You can use this key to, say, reconfigure the node distance or the node color.

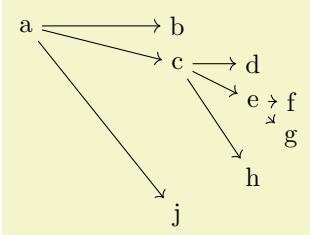
`/tikz/graph/level <level>` (style, no value)

This key also gets executed for each newly created node with `<level>` set to the current level of the node.



```

\usetikzlibrary {graphs}
\tikz \graph [
  branch down=5mm,
  level 1/.style={nodes=red},
  level 2/.style={nodes=green!50!black},
  level 3/.style={nodes=blue}]
{
  a -> {
    b,
    c -> {
      d,
      e -> {f,g},
      h
    },
    j
  }
};
```



```

\usetikzlibrary {graphs}
\tikz \graph [
  branch down=5mm,
  level 1/.style={grow right=2cm},
  level 2/.style={grow right=1cm},
  level 3/.style={grow right=5mm}]
{
  a -> {
    b,
    c -> {
      d,
      e -> {f,g},
      h
    },
    j
  }
};
```

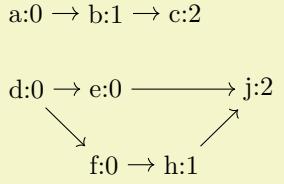
19.9.6 Defining New Online Placement Strategies

In the following the details of how to define a new placement strategy are explained. Most readers may wish to skip this section.

As a graph specification is being parsed, the `graphs` library will keep track of different numbers that identify the positions of the nodes. Let us start with what happens on a chain. First, the following counter is increased for each element of the chain:

`/tikz/graphs/placement/element count` (no value)

This key stores a number that tells us the position of the node on the current chain. However, you only have access to this value inside the code passed to the macro `compute position`, explained later on.



```
\usetikzlibrary {graphs}
\begin{tikzpicture} \graph [grow right sep, typeset=\tikzgraphnodetext:\mynum,
placement/compute position/.append code=
{\pgfkeysgetvalue{/tikz/graphs/placement/element count}\mynum}]
{
  a -> b -> c,
  d -> {e, f->h} -> j
};
```

As can be seen, each group resets the element counter.

The second value that is computed is more complicated to explain, but it also gives more interesting information:

`/tikz/graphs/placement/width` (no value)

This key stores the “logical width” of the nodes parsed up to now in the current group or chain (more precisely, parsed since the last call of `place` in an enclosing group). This is not necessarily the “total physical width” of the nodes, but rather a number representing how “big” the elements prior to the current element were. This *may* be their width, but it may also be their height or even their number (which, incidentally, is the default). You can use the `width` to perform shifts or rotations of to-be-created nodes (to be explained later).

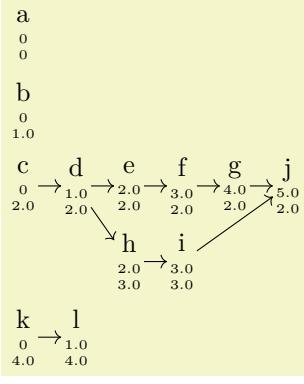
The logical width is defined recursively as follows. First, the width of a single node is computed by calling the following key:

`/tikz/graphs/placement/logical node width=<full node name>` (no default)

This key is called to compute a physical or logical width of the node `<full node name>`. You can change the code of this key. The code should return the computed value in the macro `\pgfmathresult`. By default, this key returns 1.

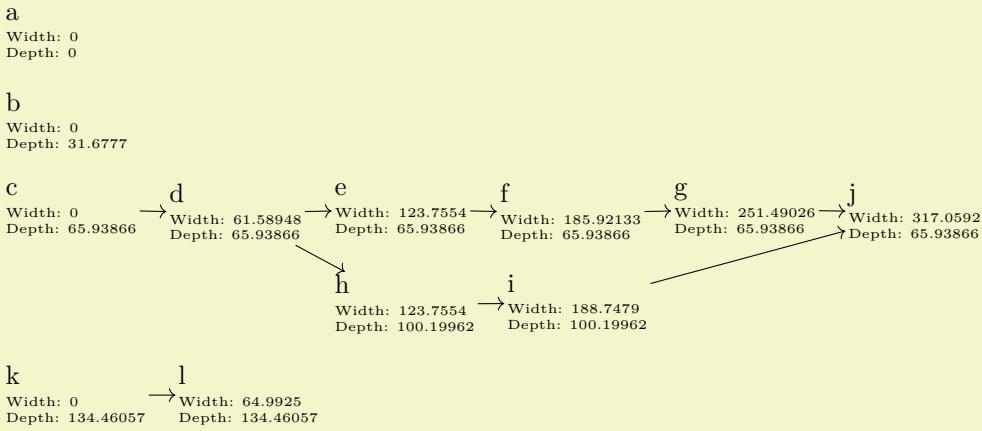
The width of a chain is the sum of the widths of its elements. The width of a group is the maximum of the widths of its elements.

To get a feeling what the above rules imply in practice, let us first have a look at an example where each node has logical width and height 1 (which is the default). The arcane options at the beginning of the code just setup things so that the computed width and depth of each node is displayed at the bottom of each node.



```
\usetikzlibrary {graphs}
\begin{tikzpicture} \graph [nodes={align=center, inner sep=1pt}, grow right=7mm,
typeset=\tikzgraphnodetext\[-4pt]
\tiny\mywidth\[-6pt]\tiny\mydepth,
placement/compute position/.append code=
{\pgfkeysgetvalue{/tikz/graphs/placement/width}\mywidth
\pgfkeysgetvalue{/tikz/graphs/placement/depth}\mydepth}
{
  a,
  b,
  c -> d -> e -> f -> g -> j,
  d -> {h, i} -> j,
  k -> l
};
```

In the next example the “logical” width and depth actually match the “physical” width and height. This is caused by the `grow right sep` option, which internally sets the `logical node width` key so that it returns the width of its parameter in points.



```
\usetikzlibrary {graphs}
\tikz
\graph [grow right sep, branch down sep, nodes={align=left, inner sep=1pt},
typeset={\tikzgraphnode{text}\[-4pt\] \tiny Width: \mywidth\[-6pt\] \tiny Depth: \mydepth},
placement/compute position/.append code=
{\pgfkeysgetvalue{/tikz/graphs/placement/width}{\mywidth}
\pgfkeysgetvalue{/tikz/graphs/placement/depth}{\mydepth}}
{
a,
b,
c -> d -> {
  e -> f -> g,
  h -> i
} -> j,
k -> l
};
```

Symmetrically to chains, as a group is being constructed, counters are available for the number of chains encountered so far in the current group and for the logical depth of the current group:

`/tikz/graphs/placement/chain count` (no value)

This key stores a number that tells us the sequence number of the chain in the current group.

```
a:0 → b:0 → c:0
      ↘
      d:1
      ↘
      e:2
f:1
g:2 → h:2
```

```
\usetikzlibrary {graphs}
\tikz \graph [
  grow right sep, branch down=5mm, typeset=\tikzgraphnode{text:\mynum,
  placement/compute position/.append code=
    {\pgfkeysgetvalue{/tikz/graphs/placement/chain count}{\mynum}}]
{
  a -> b -> {c,d,e},
  f,
  g -> h
};
```

`/tikz/graphs/placement/depth` (no value)

Similarly to the `width` key, this key stores the “logical depth” of the nodes parsed up to now in the current group or chain and, also similarly, this key may or may not be related to the actual depth/height of the current node. As for the `width`, the exact definition is as follows: For a single node, the depth is computed by the following key:

`/tikz/graphs/placement/logical node depth=<full node name>` (no default)

The code behind this key should return the “logical height” of the node `<full node name>` in the macro `\pgfmathresult`.

Second, the depth of a group is the sum of the depths of its elements. Third, the depth of a chain is the maximum of the depth of its elements.

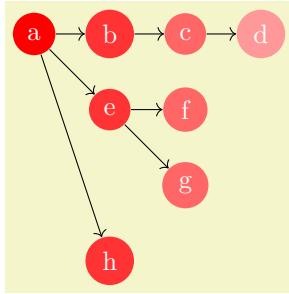
The `width`, `depth`, `element count`, and `chain count` keys get updated automatically, but do not have an effect by themselves. This is to the following two keys:

`/tikz/graphs/placement/compute position=<code>` (no default)

The `<code>` is called by the `graph` command just prior to creating a new node (the exact moment when this key is called is detailed in the description of the `place` key). When the `<code>` is called, all of the keys described above will hold numbers computed in the way described above.

The job of the `<code>` is to setup node options appropriately so that the to-be-created node will be placed correctly. Thus, the `<code>` should typically set the key `nodes={shift=<coordinate>}` where `<coordinate>` is the computed position for the node. The `<code>` could also set other options like, say, the color of a node depending on its depth.

The following example appends some code to the standard code of `compute position` so that “deeper” nodes of a tree are lighter. (Naturally, the same effect could be achieved much more easily using the `level` key.)



```

\usetikzlibrary {graphs}
\newcount\mycount
\def\lightendeepernodes{
  \pgfmathsetcount{\mycount}{%
    100-20*\pgfkeysvalueof{/tikz/graphs/placement/width}}
}
\edef\mydepth{\the\mycount}
\tikzset{nodes={fill=red!\mydepth,circle,text=white}}
}
\tikz
\graph [placement/compute position/.append code=\lightendeepernodes]
{
  a -> {
    b -> c -> d,
    e -> {
      f,
      g
    },
    h
  };
}
  
```

`/tikz/graphs/placement/place` (no value)

Executing this key has two effects: First, the key `compute position` is called to compute a good position for future nodes (usually, these “future nodes” are just a single node that is created immediately). Second, all of the above counters like `depth` or `width` are reset (but not `level`).

There are two places where this key is sensibly called: First, just prior to creating a node, which happens automatically. Second, when you change the online strategy. In this case, the computed width and depth values from one strategy typically make no sense in the other strategy, which is why the new strategy should proceed “from a fresh start”. In this case, the implicit call of `compute position` ensures that the new strategy gets the last place the old strategy would have used as its starting point, while the computation of its positions is now relative to this new starting point.

For these reasons, when an online strategy like `Cartesian placement` is called, this key gets called implicitly. You will rarely need to call this key directly, except when you define a new online strategy.

19.10 Reference: Predefined Elements

19.10.1 Graph Macros

TikZ Library `graphs.standard`

```
\usetikzlibrary{graphs.standard} % LATEX and plain TEX
\usetikzlibrary[graphs.standard] % ConTEXt
```

This library defines a number of graph macros that are often used in the literature. When new graphs are added to this collection, they will follow the definitions in the Mathematica program, see mathworld.wolfram.com/topics/SimpleGraphs.html.

Graph `subgraph I_n`

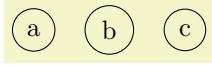
This graph consists just of n unconnected vertices. The following key is used to specify the set of these vertices:

`/tikz/graphs/V={⟨list of vertices⟩}` (no default)

Sets a list of vertex names for use with graphs like `subgraph I_n` and also other graphs. This list is available in the macro `\tikzgraphV`. The number of elements of this list is available in `\tikzgraphVnum`.

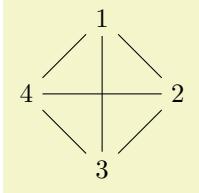
`/tikz/graphs/n=⟨number⟩` (no default)

This is an abbreviation for `V={1, …, ⟨number⟩}`, `name shore V/.style={name=V}`.



```
\usetikzlibrary {graphs.standard}
\tikz \graph [branch right, nodes={draw, circle}]
{ subgraph I_n [V={a,b,c}] };
```

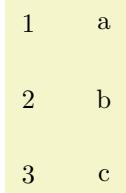
This graph is not particularly exciting by itself. However, it is often used to introduce nodes into a graph that are then connected as in the following example:



```
\usetikzlibrary {graphs.standard}
\tikz \graph [clockwise, clique] { subgraph I_n [n=4] };
```

Graph `subgraph I_nm`

This graph consists of two sets of once n unconnected vertices and then m unconnected vertices. The first set consists of the vertices set by the key `V`, the other set consists of the vertices set by the key `W`.



```
\usetikzlibrary {graphs.standard}
\tikz \graph { subgraph I_nm [V={1,2,3}, W={a,b,c}] };
```

In order to set the graph path name of the two sets, the following keys get executed:

`/tikz/graphs/name shore V` (style, initially empty)

Set this style to, say, `name=my V set` in order to set a name for the `V` set.

`/tikz/graphs/name shore W` (style, initially empty)

Same as for `name shore V`.

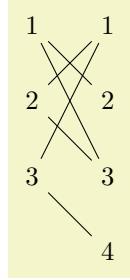
`/tikz/graphs/W={⟨list of vertices⟩}` (no default)

Sets the list of vertices for the `W` set. The elements and their number are available in the macros `\tikzgraphW` and `\tikzgraphWnum`, respectively.

`/tikz/graphs/m=⟨number⟩` (no default)

This is an abbreviation for `W={1, …, ⟨number⟩}`, `name shore W/.style={name=W}`.

The main purpose of this subgraph is to setup the nodes in a bipartite graph:

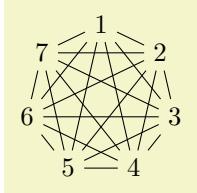


```
\usetikzlibrary {graphs.standard}
\begin{tikzpicture}
\graph [clockwise] {
    subgraph I_nm [n=3, m=4];
        V 1 -- { W 2, W 3 };
        V 2 -- { W 1, W 3 };
        V 3 -- { W 1, W 4 };
    }

```

Graph `subgraph K_n`

This graph is the complete clique on the vertices from the V key.

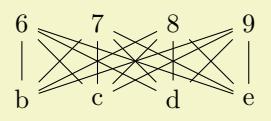


```
\usetikzlibrary {graphs.standard}
\begin{tikzpicture}
\graph [clockwise] {
    subgraph K_n [n=7];
}

```

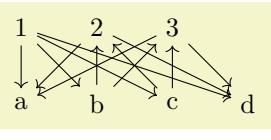
Graph `subgraph K_nm`

This graph is the complete bipartite graph with the two shores V and W as in `subgraph I_nm`.



```
\usetikzlibrary {graphs.standard}
\begin{tikzpicture}
\graph [branch right, grow down] {
    subgraph K_nm [V={6,...,9}, W={b,...,e}];
}

```

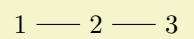


```
\usetikzlibrary {graphs.standard}
\begin{tikzpicture}
\graph [simple, branch right, grow down] {
    subgraph K_nm [V={1,2,3}, W={a,b,c,d}, ->];
    subgraph K_nm [V={2,3}, W={b,c}, <-];
}

```

Graph `subgraph P_n`

This graph is the path on the vertices in V.

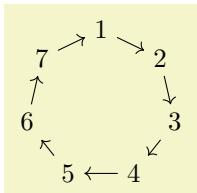


```
\usetikzlibrary {graphs.standard}
\begin{tikzpicture}
\graph [branch right] {
    subgraph P_n [n=3];
}

```

Graph `subgraph C_n`

This graph is the cycle on the vertices in V.



```
\usetikzlibrary {graphs.standard}
\begin{tikzpicture}
\graph [clockwise] {
    subgraph C_n [n=7, ->];
}

```

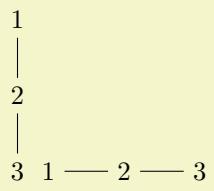
Graph `subgraph Grid_n`

This graph is a grid of the vertices in V.

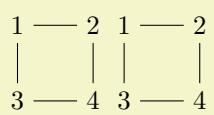
`/tikz/graphs/wrap after=<number>`

(no default)

Defines the number of nodes placed in a single row of the grid. This value implicitly defines the number of grid columns as well. In the following example a `grid placement` is used to visualize the edges created between the nodes of a `Grid_n` subgraph using different values for `wrap after`.



```
\usetikzlibrary {graphs.standard}
\begin{tikzpicture}
\graph [grid placement] {
    subgraph Grid_n [n=3, wrap after=1];
    subgraph Grid_n [n=3, wrap after=3];
}
\end{tikzpicture}
```



```
\usetikzlibrary {graphs.standard}
\begin{tikzpicture}
\graph [grid placement] {
    subgraph Grid_n [n=4, wrap after=2];
    subgraph Grid_n [n=4];
}
\end{tikzpicture}
```

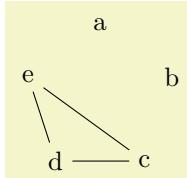
19.10.2 Group Operators

The following keys use the `operator` key to setup operators that connect the vertices of the current group having a certain color in a specific way.

`/tikz/graphs/clique=<color>`

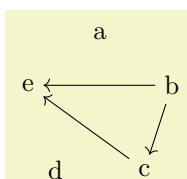
(default `all`)

Adds an edge between all vertices of the current group having the (logical) color `<color>`. Since, by default, this color is set to `all`, which is a color that all nodes get by default, when you do not specify anything, all nodes will be connected.



```
\usetikzlibrary {graphs}
\begin{tikzpicture}
\graph [clockwise, n=5] {
    a,
    b,
    {
        [clique]
        c, d, e
    }
};

```



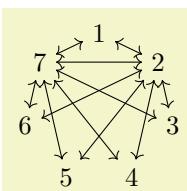
```
\usetikzlibrary {graphs}
\begin{tikzpicture}
\graph [color class=red, clockwise, n=5] {
    [clique=red, ->]
    a, b[red], c[red], d, e[red]
};

```

`/tikz/graphs/induced independent set=<color>`

(default `all`)

This key is the “opposite” of a `clique`: It removes all edges in the current group having belonging to color class `<color>`. More precisely, an edge of kind `-!-` is added for each pair of vertices. This means that edge only get removed if you specify the `simple` option.



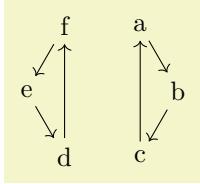
```
\usetikzlibrary {graphs.standard}
\begin{tikzpicture}
\graph [simple] {
    subgraph K_n [<->, n=7, clockwise]; % create lots of edges
    {
        [induced independent set] 1, 3, 4, 5, 6
    }
};

```

`/tikz/graphs/cycle=<color>`

(default all)

Connects the nodes colored `<color>` in a cyclic fashion. The ordering is the ordering in which they appear in the whole graph specification.



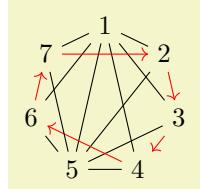
```
\usetikzlibrary {graphs}
\begin{tikzpicture}
\graph [clockwise, n=6, phase=60] {
  \cyclicity[>]{a, b, c},
  \cyclicity[<]{d, e, f}
};

```

`/tikz/graphs/induced cycle=<color>`

(default all)

While the `cycle` command will only add edges, this key will also remove all other edges between the nodes of the cycle, provided we are constructing a `simple` graph.



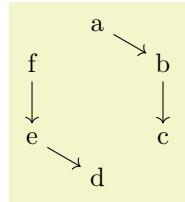
```
\usetikzlibrary {graphs.standard}
\begin{tikzpicture}
\graph [simple] {
  subgraph K_n [n=7, clockwise]; % create lots of edges
  \inducedcycle[>, edge=red]{2, 3, 4, 6, 7},
};

```

`/tikz/graphs/path=<color>`

(default all)

Works like `cycle`, only there is no edge from the last to the first vertex.



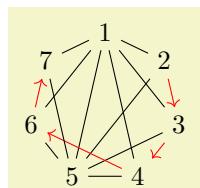
```
\usetikzlibrary {graphs}
\begin{tikzpicture}
\graph [clockwise, n=6] {
  \path[>]{a, b, c},
  \path[<]{d, e, f}
};

```

`/tikz/graphs/induced path=<color>`

(default all)

Works like `induced cycle`, only there is no edge from the last to the first vertex.



```
\usetikzlibrary {graphs.standard}
\begin{tikzpicture}
\graph [simple] {
  subgraph K_n [n=7, clockwise]; % create lots of edges
  \inducedpath[>, edges=red]{2, 3, 4, 6, 7},
};

```

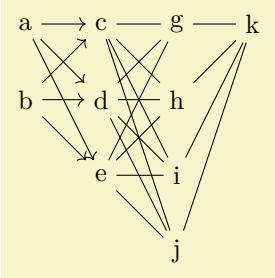
19.10.3 Joining Operators

The following keys are typically used as options of an `<edge specification>`, but can also be called in a group specification (however, then, the colors need to be set explicitly).

`/tikz/graphs/complete bipartite=<from color><to color>`

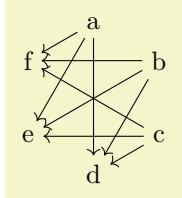
(default {source'}{target'})

Adds all possible edges from every node having color `<from color>` to every node having color `<to color>`:



```
\usetikzlibrary {graphs}
\begin{tikzpicture}
\graph [complete bipartite]
    {a, b}      -> [complete bipartite]
    {c, d, e}   -- [complete bipartite]
    {g, h, i, j} -- [complete bipartite]
    k ;

```



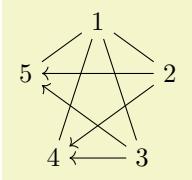
```
\usetikzlibrary {graphs}
\begin{tikzpicture}
\graph [color class=red, color class=green, clockwise, n=6] {
    [complete bipartite={red}{green}, ->]
    a [red], b[red], c[red], d[green], e[green], f[green]
};

```

/tikz/graphs/induced complete bipartite

(no value)

Works like the `complete bipartite` operator, but in a `simple` graph any edges between the vertices in either shore are removed (more precisely, they get replaced by `-!-` edges).



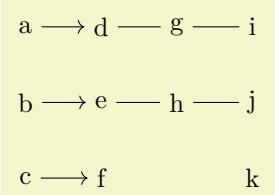
```
\usetikzlibrary {graphs.standard}
\begin{tikzpicture}
\graph [simple] {
    subgraph K_n [n=5, clockwise]; % Lots of edges
    {2, 3} -> [induced complete bipartite] {4, 5}
};

```

/tikz/graphs/matching=<from color><to color>

(default `{source'}``{target'}`)

This joining operator forms a maximum *matching* between the nodes of the two sets of nodes having colors `<from color>` and `<to color>`, respectively. The first node of the from set is connected to the first node of the to set, the second node of the from set is connected to the second node of the to set, and so on. If the sets have the same size, what results is what graph theoreticians call a *perfect matching*, otherwise only a maximum, but not perfect matching results.



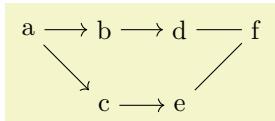
```
\usetikzlibrary {graphs}
\begin{tikzpicture}
\graph {
    {a, b, c} -> [matching]
    {d, e, f} -- [matching]
    {g, h}     -- [matching]
    {i, j, k}
};

```

/tikz/graphs/matching and star=<from color><to color>

(default `{source'}``{target'}`)

The `matching` and `star` connector works like the `matching` connector, only it behaves differently when the two to-be-connected sets have different size. In this case, all the surplus nodes get connected to the last node of the other set, resulting in what is known as a *star* in graph theory. This simple rule allows for some powerful effects (since this connector is the one initially set, there is no need to add it here):



```
\usetikzlibrary {graphs}
\begin{tikzpicture}
\graph {
    a -> {b, c} -> {d, e} -- f;
};

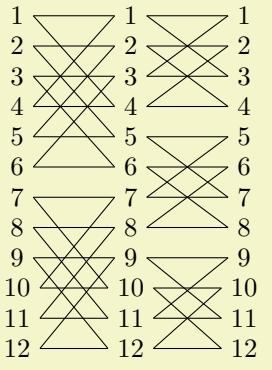
```

The `matching` and `star` connector also makes it easy to create trees and series-parallel graphs.

`/tikz/graphs/butterfly=<options>`

(no default)

The `butterfly` connector is used to create the kind of connections present between layers of a so-called *butterfly network*. As for other connectors, two sets of nodes are connected, which are the nodes having color `target'` and `source'` by default. In a *level l* connection, the first l nodes of the first set are connected to the second l nodes of the second set, while the second l nodes of the first set get connected to the first l nodes of the second set. Then, for next $2l$ nodes of both sets a similar kind of connection is installed. Additionally, each node gets connected to the corresponding node in the other set with the same index (as in a `matching`):



```
\usetikzlibrary {graphs.standard}
\tikz \graph [left anchor=east, right anchor=west,
branch down=4mm, grow right=15mm] {
subgraph I_n [n=12, name=A] --[butterfly={level=3}]
subgraph I_n [n=12, name=B] --[butterfly={level=2}]
subgraph I_n [n=12, name=C]
};
```

Unlike most joining operators, the colors of the nodes in the first and the second set are not passed as parameters to the `butterfly` key. Rather, they can be set using the `<options>`, which are executed with the path prefix `/tikz/graphs/butterfly`.

`/tikz/graphs/butterfly/level=<level>`

(no default, initially 1)

Sets the level l for the connections.

`/tikz/graphs/butterfly/from=<color>`

(no default, initially `target'`)

Sets the color class of the from nodes.

`/tikz/graphs/butterfly/to=<color>`

(no default, initially `source'`)

Sets the color class of the to nodes.

20 Matrices and Alignment

20.1 Overview

When creating pictures, one often faces the problem of correctly aligning parts of the picture. For example, you might wish that the `baselines` of certain nodes should be on the same line and some further nodes should be below these nodes with, say, their centers on a vertical lines. There are different ways of solving such problems. For example, by making clever use of anchors, nearly all such alignment problems can be solved. However, this often leads to complicated code. An often simpler way is to use *matrices*, the use of which is explained in the current section.

A TikZ matrix is similar to L^AT_EX's `{tabular}` or `{array}` environment, only instead of text each cell contains a little picture or a node. The sizes of the cells are automatically adjusted such that they are large enough to contain all the cell contents.

Matrices are a powerful tool and they need to be handled with some care. For impatient readers who skip the rest of this section: you *must* end *every* row with `\\"`. In particular, the last row *must* be ended with `\\"`.

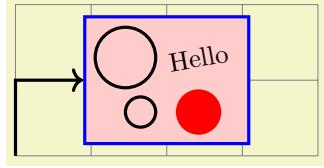
Many of the ideas implemented in TikZ's matrix support are due to Mark Wibrow – many thanks to Mark at this point!

20.2 Matrices are Nodes

Matrices are special in many ways, but for most purposes matrices are treated like nodes. This means, that you use the `node` path command to create a matrix and you only use a special option, namely the `matrix` option, to signal that the node will contain a matrix. Instead of the usual T_EX-box that makes up the `text` part of the node's shape, the matrix is used. Thus, in particular, a matrix can have a shape, this shape can be drawn or filled, it can be used in a tree, and so on. Also, you can refer to the different anchors of a matrix.

`/tikz/matrix=⟨true or false⟩` (default `true`)

This option can be passed to a `node` path command. It signals that the node will contain a matrix.



```
\begin{tikzpicture}
  \draw[help lines] (0,0) grid (4,2);
  \node [matrix,fill=red!20,draw=blue,very thick] (my matrix) at (2,1)
  {
    \draw (0,0) circle (4mm); & \node[rotate=10] {Hello}; \\
    \draw (0.2,0) circle (2mm); & \fill[red] (0,0) circle (3mm); \\
  };

  \draw [very thick,->] (0,0) |- (my matrix.west);
\end{tikzpicture}
```

The exact syntax of the matrix is explained in the course of this section.

`/tikz/every matrix` (style, initially empty)

This style is used in every matrix.

`/tikz/every outer matrix` (style, initially empty)

While the `every matrix` key also applies to the matrix contents, this only applies to the outer node which holds the matrix.

Even more so than nodes, matrices will often be the only object on a path. Because of this, there is a special abbreviation for creating matrices:

`\matrix`

Inside `{tikzpicture}` this is an abbreviation for `\path node[matrix]`.

Even though matrices are nodes, some options do not have the same effect as for normal nodes:

1. Rotations and scaling have no effect on a matrix as a whole (however, you can still transform the contents of the cells normally). Before the matrix is typeset, the rotational and scaling part of the transformation matrix is reset.

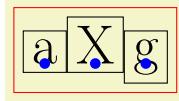
2. For multi-part shapes you can only set the `text` part of the node.
3. All options starting with `text` such as `text width` have no effect.
4. If you place a matrix on a path, the matrix contents will be collected into a macro, which tokenizes them. This means that `&` will lose its meaning as an alignment character, resulting in an error. If you need to place a matrix on a path, use `ampersand replacement` to work around that problem.

20.3 Cell Pictures

A matrix consists of rows of *cells*. Each row (including the last one!) is ended by the command `\\"`. The character `&` is used to separate cells. Inside each cell, you must place commands for drawing a picture, called the *cell picture* in the following. (However, cell pictures are not enclosed in a complete `{pgfpicture}` environment, they are a bit more light-weight. The main difference is that cell pictures cannot have layers.) It is not necessary to specify beforehand how many rows or columns there are going to be and if a row contains less cell pictures than another line, empty cells are automatically added as needed.

20.3.1 Alignment of Cell Pictures

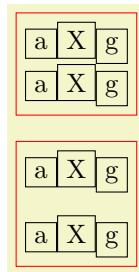
For each cell picture a bounding box is computed. These bounding boxes and the origins of the cell pictures determine how the cells are aligned. Let us start with the rows: Consider the cell pictures on the first row. Each has a bounding box and somewhere inside this bounding box the origin of the cell picture can be found (the origin might even lie outside the bounding box, but let us ignore this problem for the moment). The cell pictures are then shifted around such that all origins lie on the same horizontal line. This may make it necessary to shift some cell pictures upwards and others downwards, but it can be done and this yields the vertical alignment of the cell pictures this row. The top of the row is then given by the top of the “highest” cell picture in the row, the bottom of the row is given by the bottom of the lowest cell picture. (To be more precise, the height of the row is the maximum *y*-value of any of the bounding boxes and the depth of the row is the negated minimum *y*-value of the bounding boxes).



```
\begin{tikzpicture}
[every node/.style={draw=black, anchor=base, font=\huge}]

\matrix [draw=red]
{
\node {a}; \fill[blue] (0,0) circle (2pt); &
\node {X}; \fill[blue] (0,0) circle (2pt); &
\node {g}; \fill[blue] (0,0) circle (2pt); \\
};
\end{tikzpicture}
```

Each row is aligned in this fashion: For each row the cell pictures are vertically aligned such that the origins lie on the same line. Then the second row is placed below the first row such that the bottom of the first row touches the top of the second row (unless a `row sep` is used to add a bit of space). Then the bottom of the second row touches the top of the third row, and so on. Typically, each row will have an individual height and depth.



```
\begin{tikzpicture}
[every node/.style={draw=black, anchor=base}]

\matrix [draw=red]
{
\node {a}; & \node {X}; & \node {g}; \\
\node {a}; & \node {X}; & \node {g}; \\
};

\matrix [row sep=3mm, draw=red] at (0,-2)
{
\node {a}; & \node {X}; & \node {g}; \\
\node {a}; & \node {X}; & \node {g}; \\
};
\end{tikzpicture}
```

Let us now have a look at the columns. The rules for how the pictures on any given column are aligned are very similar to the row alignment: Consider all cell pictures in the first column. Each is shifted horizontally such that the origins lie on the same vertical line. Then, the left end of the column is at the

left end of the bounding box that protrudes furthest to the left. The right end of the column is at the right end of the bounding box that protrudes furthest to the right. This fixes the horizontal alignment of the cell pictures in the first column and the same happens the cell pictures in the other columns. Then, the right end of the first column touches the left end of the second column (unless `column sep` is used). The right end of the second column touches the left end of the third column, and so on. (Internally, two columns are actually used to achieve the desired horizontal alignment, but that is only an implementation detail.)

```
\begin{tikzpicture}[every node/.style={draw}]
\matrix [draw=red]
{
\node[left] {Hallo}; \fill[blue] (0,0) circle (2pt); \\
\node {} {X}; \fill[blue] (0,0) circle (2pt); \\
\node[right] {g}; \fill[blue] (0,0) circle (2pt); \\
};
\end{tikzpicture}
```



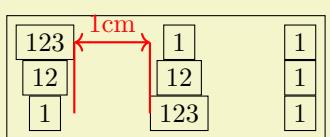
```
\begin{tikzpicture}[every node/.style={draw}]
\matrix [draw=red,column sep=1cm]
{
\node {8}; & \node {1}; & \node {6}; \\
\node {3}; & \node {5}; & \node {7}; \\
\node {4}; & \node {9}; & \node {2}; \\
};
\end{tikzpicture}
```

20.3.2 Setting and Adjusting Column and Row Spacing

There are different ways of setting and adjusting the spacing between columns and rows. First, you can use the options `column sep` and `row sep` to set a default spacing for all rows and all columns. Second, you can add options to the `&` character and the `\>` command to adjust the spacing between two specific columns or rows. Additionally, you can specify whether the space between two columns or rows should be considered between the origins of cells in the column or row or between their borders.

`/tikz/column sep=<spacing list>` (no default)

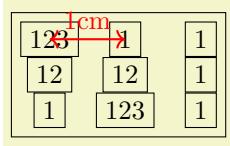
This option sets a default space that is added between every two columns. This space can be positive or negative and is zero by default. The `<spacing list>` normally contains a single dimension like `2pt`.



```
\begin{tikzpicture}
\matrix [draw,column sep=1cm,nodes=draw]
{
\node(a) {123}; & \node(b) {1}; & \node(c) {1}; \\
\node(d) {12}; & \node(e) {12}; & \node(f) {1}; \\
\node(g) {1}; & \node(h) {123}; & \node(i) {1}; \\
};
\draw [red,thick] (a.east) -- (a.east |- c)
                  (d.west) -- (d.west |- b);
\draw [<->,red,thick] (a.east) -- (d.west |- b)
  node [above,midway] {1cm};
\end{tikzpicture}
```

More generally, the `<spacing list>` may contain a whole list of numbers, separated by commas, and occurrences of the two key words `between origins` and `between borders`. The effect of specifying such a list is the following: First, all numbers occurring in the list are simply added to compute the final spacing. Second, concerning the two keywords, the last occurrence of one of the keywords is important. If the last occurrence is `between borders` or if neither occurs, then the space is inserted between the two columns normally. However, if the last occurs is `between origins`, then the following happens: The distance between the columns is adjusted such that the difference between the origins of all the cells in the first column (remember that they all lie on straight line) and the origins of all the cells in the second column is exactly the given distance.

The `between origins` option can only be used for columns mentioned in the first row, that is, you cannot specify this option for columns introduced only in later rows.

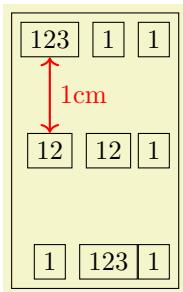


```
\begin{tikzpicture}
\matrix [draw,column sep={1cm,between origins},nodes=draw]
{
\node(a) {123}; & \node(b) {1}; & \node(c) {1}; \\
\node(d) {12}; & \node(e) {12}; & \node(f) {1}; \\
\node(g) {1}; & \node(h) {123}; & \node(i) {1}; \\
};
\draw [<->,red,thick] (a.center) -- (b.center) node [above,midway] {1cm};
\end{tikzpicture}
```

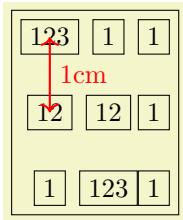
`/tikz/row sep=<spacing list>`

(no default)

This option works like `column sep`, only for rows. Here, too, you can specify whether the space is added between the lower end of the first row and the upper end of the second row, or whether the space is computed between the origins of the two rows.

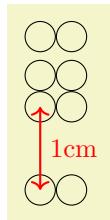


```
\begin{tikzpicture}
\matrix [draw,row sep=1cm,nodes=draw]
{
\node(a) {123}; & \node(b) {1}; & \node(c) {1}; \\
\node(d) {12}; & \node(e) {12}; & \node(f) {1}; \\
\node(g) {1}; & \node(h) {123}; & \node(i) {1}; \\
};
\draw [<->,red,thick] (a.south) -- (b.north) node [right,midway] {1cm};
\end{tikzpicture}
```



```
\begin{tikzpicture}
\matrix [draw,row sep={1cm,between origins},nodes=draw]
{
\node(a) {123}; & \node(b) {1}; & \node(c) {1}; \\
\node(d) {12}; & \node(e) {12}; & \node(f) {1}; \\
\node(g) {1}; & \node(h) {123}; & \node(i) {1}; \\
};
\draw [<->,red,thick] (a.center) -- (b.center) node [right,midway] {1cm};
\end{tikzpicture}
```

The row-end command `\\\` allows you to provide an optional argument, which must be a dimension. This dimension will be added to the list in `row sep`. This means that, firstly, any numbers you list in this argument will be added as an extra row separation between the line being ended and the next line and, secondly, you can use the keywords `between origins` and `between borders` to locally overrule the standard setting for this line pair.



```
\begin{tikzpicture}
\matrix [row sep=1mm]
{
\draw (0,0) circle (2mm); & \draw (0,0) circle (2mm); \\
\draw (0,0) circle (2mm); & \draw (0,0) circle (2mm); & \draw (0,0) circle (2mm); \\
\draw (0,0) coordinate (a) circle (2mm); & \\
\draw (0,0) circle (2mm); \\
\draw (0,0) coordinate (b) circle (2mm); & \\
\draw (0,0) circle (2mm); \\
};
\draw [<->,red,thick] (a.center) -- (b.center) node [right,midway] {1cm};
\end{tikzpicture}
```

The cell separation character `&` also takes an optional argument, which must also be a spacing list. This spacing list is added to the `column sep` having a similar effect as the option for the `\\\` command for rows.

This optional spacing list can only be given the first time a new column is started (usually in the first row), subsequent usages of this option in later rows have no effect.

8	1	6
3	5	7
4	9	2

```
\begin{tikzpicture}
\matrix [draw,nodes=draw,column sep=1mm]
{
\node {8}; &[2mm] \node{1}; &[-1mm] \node {6}; \\
\node {3}; & \node{5}; & \node {7}; \\
\node {4}; & \node{9}; & \node {2}; \\
};
\end{tikzpicture}
```

8	1	6
3	5	7
4	9	2

```
\begin{tikzpicture}
\matrix [draw,nodes=draw,column sep=1mm]
{
\node {8}; &[2mm] \node(a){1}; &[1cm,between origins] \node(b){6}; \\
\node {3}; & \node {5}; & \node {7}; \\
\node {4}; & \node {9}; & \node {2}; \\
};
\draw [->,red,thick] (a.center) -- (b.center) node [above,midway] {11mm};
\end{tikzpicture}
```

8	1	6
3	5	7
4	9	2

```
\begin{tikzpicture}
\matrix [draw,nodes=draw,column sep={1cm,between origins}]
{
\node (a) {8}; & \node (b) {1}; &[between borders] \node (c) {6}; \\
\node {3}; & \node {5}; & \node {7}; \\
\node {4}; & \node {9}; & \node {2}; \\
};
\draw [->,red,thick] (a.center) -- (b.center) node [above,midway] {10mm};
\draw [->,red,thick] (b.east) -- (c.west) node [above,midway] {10mm};
\end{tikzpicture}
```

20.3.3 Cell Styles and Options

The following styles and options are useful for changing the appearance of all cell pictures:

`/tikz/every cell={⟨row⟩}{⟨column⟩}`

(style, no default, initially empty)

This style is installed at the beginning of each cell picture with the two parameters being the current `⟨row⟩` and `⟨column⟩` of the cell. Note that setting this style to `draw` will *not* cause all nodes to be drawn since the `draw` option has to be passed to each node individually.

Inside this style (and inside all cells), the current `⟨row⟩` and `⟨column⟩` number are also accessible via the counters `\pgfmatrixcurrentrow` and `\pgfmatrixcurrentcolumn`.

`/tikz/cells=⟨options⟩`

(no default)

This key adds the `⟨options⟩` to the style `every cell`. It is mainly just a shorthand for the code `every cell/.append style=⟨options⟩`.

`/tikz/nodes=⟨options⟩`

(no default)

This key adds the `⟨options⟩` to the style `every node`. It is mainly just a shorthand for the code `every node/.append style=⟨options⟩`.

The main use of this option is the install some options for the nodes *inside* the matrix that should not apply to the matrix *itself*.

8	1	6
3	5	7
4	9	2

```
\begin{tikzpicture}
\matrix [nodes={fill=blue!20,minimum size=5mm}]
{
\node {8}; & \node{1}; & \node {6}; \\
\node {3}; & \node{5}; & \node {7}; \\
\node {4}; & \node{9}; & \node {2}; \\
};
\end{tikzpicture}
```

The next set of styles can be used to change the appearance of certain rows, columns, or cells. If more than one of these styles is defined, they are executed in the below order (the `every cell` style is executed before all of the below).

<code>/tikz/column <number></code>	(style, no value)
This style is used for every cell in column <i><number></i> .	
<code>/tikz/every odd column</code>	(style, no value)
This style is used for every cell in an odd column.	
<code>/tikz/every even column</code>	(style, no value)
This style is used for every cell in an even column.	
<code>/tikz/row <number></code>	(style, no value)
This style is used for every cell in row <i><number></i> .	
<code>/tikz/every odd row</code>	(style, no value)
This style is used for every cell in an odd row.	
<code>/tikz/every even row</code>	(style, no value)
This style is used for every cell in an even row.	
<code>/tikz/row <row number> column <column number></code>	(style, no value)
This style is used for the cell in row <i><row number></i> and column <i><column number></i> .	

8 1 6
3 5 7
4 9 2

```
\begin{tikzpicture}
  [row 1/.style={red},
   column 2/.style={green!50!black},
   row 3 column 3/.style={blue}]

  \matrix
  {
    \node {8}; & \node{1}; & \node {6}; \\
    \node {3}; & \node{5}; & \node {7}; \\
    \node {4}; & \node{9}; & \node {2}; \\
  };
\end{tikzpicture}
```

You can use the `column <number>` option to change the alignment for different columns.

123 456 789
12 45 78
1 4 7

```
\begin{tikzpicture}
  [column 1/.style={anchor=base west},
   column 2/.style={anchor=base east},
   column 3/.style={anchor=base}]
  \matrix
  {
    \node {123}; & \node{456}; & \node {789}; \\
    \node {12}; & \node{45}; & \node {78}; \\
    \node {1}; & \node{4}; & \node {7}; \\
  };
\end{tikzpicture}
```

In many matrices all cell pictures have nearly the same code. For example, cells typically start with `\node{` and end `};`. The following options allow you to execute such code in all cells:

<code>/tikz/execute at begin cell=<code></code>	(no default)
The code will be executed at the beginning of each nonempty cell.	
<code>/tikz/execute at end cell=<code></code>	(no default)
The code will be executed at the end of each nonempty cell.	
<code>/tikz/execute at empty cell=<code></code>	(no default)
The code will be executed inside each empty cell.	

8	1	6
3	5	7
4	9	2

```
\begin{tikzpicture}
  [matrix of nodes/.style={%
    execute at begin cell=\node\bgroup,
    execute at end cell=\egroup;%
  }]
  \matrix [matrix of nodes]
  {
    8 & 1 & 6 \\
    3 & 5 & 7 \\
    4 & 9 & 2 \\
  };
\end{tikzpicture}
```

8	1	-
3	-	7
-	-	2

```
\begin{tikzpicture}
  [matrix of nodes/.style={%
    execute at begin cell=\node\bgroup,
    execute at end cell=\egroup;,
    execute at empty cell=\node{--};%
  }]
  \matrix [matrix of nodes]
  {
    8 & 1 & \\
    3 & & 7 \\
    & & 2 \\
  };
\end{tikzpicture}
```

The `matrix` library defines a number of styles that make use of the above options.

20.4 Anchoring a Matrix

Since matrices are nodes, they can be anchored in the usual fashion using the `anchor` option. However, there are two ways to influence this placement further. First, the following option is often useful:

`/tikz/matrix anchor=<anchor>` (no default)

This option has the same effect as `anchor`, but the option applies only to the matrix itself, not to the cells inside. If you just say `anchor=north` as an option to the matrix node, all nodes inside matrix will also have this anchor, unless it is explicitly set differently for each node. By comparison, `matrix anchor` sets the anchor for the matrix, but for the nodes inside the value of `anchor` remain unchanged.

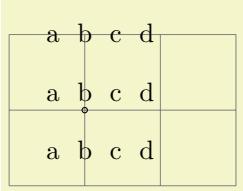
123
12
1
123
12
1

```
\begin{tikzpicture}
  \matrix [matrix anchor=west] at (0,0)
  {
    \node {123}; \\
    \node {12}; \\
    \node {1}; \\
  };
  \matrix [anchor=west] at (0,-2)
  {
    \node {123}; \\
    \node {12}; \\
    \node {1}; \\
  };
\end{tikzpicture}
```

The second way to anchor a matrix is to use *an anchor of a node inside the matrix*. For this, the `anchor` option has a special effect when given as an argument to a matrix:

`/tikz/anchor=<anchor or node.anchor>` (no default)

Normally, the argument of this option refers to anchor of the matrix node, which is the node that includes all of the stuff of the matrix. However, you can also provide an argument of the form `<node>.⟨anchor⟩` where `<node>` must be node defined inside the matrix and `⟨anchor⟩` is an anchor of this node. In this case, the whole matrix is shifted around in such a way that this particular anchor of this particular node lies at the `at` position of the matrix. The same is true for `matrix anchor`.



```
\begin{tikzpicture}
\draw[help lines] (0,0) grid (3,2);
\matrix[matrix anchor=inner node.south, anchor=base, row sep=3mm] at (1,1)
{
\node {a}; & \node {b}; & \node {c}; & \node {d}; \\
\node {a}; & \node[inner node]{b}; & \node {c}; & \node {d}; \\
\node {a}; & \node {b}; & \node {c}; & \node {d}; \\
};
\draw (inner node.south) circle (1pt);
\end{tikzpicture}
```

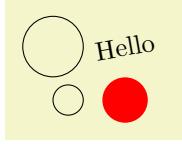
20.5 Considerations Concerning Active Characters

Even though TikZ seems to use `&` to separate cells, PGF actually uses a different command to separate cells, namely the command `\pgfmatrixnextcell` and using a normal `&` character will normally fail. What happens is that, TikZ makes `&` an active character and then defines this character to be equal to `\pgfmatrixnextcell`. In most situations this will work nicely, but sometimes `&` cannot be made active; for instance because the matrix is used in an argument of some macro or the matrix contains nodes that contain normal `{tabular}` environments. In this case you can use the following option to avoid having to type `\pgfmatrixnextcell` each time:

`/tikz/ampersand replacement=<macro name or empty>`

(no default)

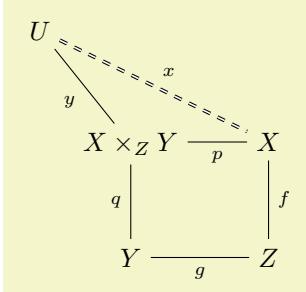
If a macro name is provided, this macro will be defined to be equal to `\pgfmatrixnextcell` inside matrices and `&` will not be made active. For instance, you could say `ampersand replacement=\&` and then use `\&` to separate columns as in the following example:



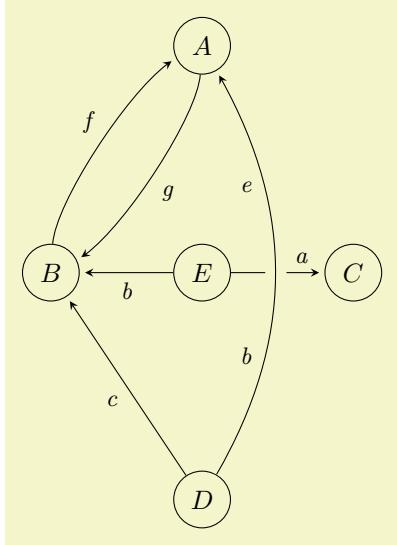
```
\tikz
\matrix [ampersand replacement=\&]
{
\draw (0,0) circle (4mm); \& \node[rotate=10] {Hello}; \\
\draw (0.2,0) circle (2mm); \& \fill[red] (0,0) circle (3mm); \\
};
```

20.6 Examples

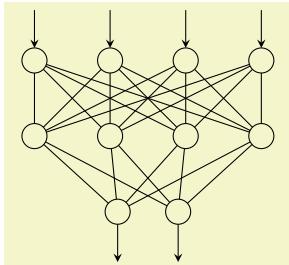
The following examples are adapted from code by Mark Wibrow. The first two redraw pictures from Timothy van Zandt's PStricks documentation:



```
\usetikzlibrary {matrix}
\begin{tikzpicture}
\matrix [matrix of math nodes, row sep=1cm]
{
| (U) | & U & | (X) | & X & | (Y) | & Y & | (Z) | & Z \\
& \& \& \& \& \& \& \\
};
\begin{scope}[every node/.style={midway, auto, font=\scriptsize}]
\draw [double, dashed] (U) -- node {$x$} (X);
\draw (X) -- node {$p$} (Y);
\draw (X) -- node {$q$} (Z);
\draw (Y) -- node {$g$} (Z);
\draw (U) -- node {$y$} (Y);
\draw (U) -- node {$f$} (Z);
\end{scope}
\end{tikzpicture}
```



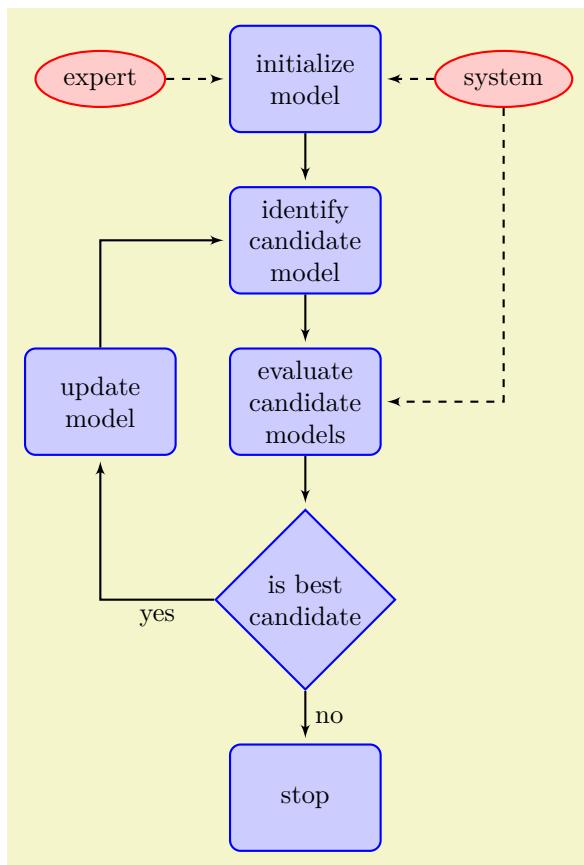
```
\usetikzlibrary {matrix}
\begin{tikzpicture} [->=stealth,->,shorten >=2pt,looseness=.5,auto]
 \matrix [matrix of math nodes,
   column sep={2cm,between origins},
   row sep={3cm,between origins},
   nodes={circle, draw, minimum size=7.5mm}]
 {
   & |(A)| A & \\
 |(B)| B & |(E)| E & |(C)| C \\
   & |(D)| D & \\
 };
 \begin{scope}[every node/.style={font=\small\itshape}]
 \draw (A) to [bend left] node [midway] {g} (B);
 \draw (B) to [bend left] node [midway] {f} (A);
 \draw (D) -- node [midway] {c} (B);
 \draw (E) -- node [midway] {b} (B);
 \draw (E) -- node [near end] {a} (C);
 \draw [-,line width=8pt,draw=graphicbackground]
 (D) to [bend right, looseness=1] (A);
 \draw (D) to [bend right, looseness=1]
 node [near start] {b} node [near end] {e} (A);
 \end{scope}
\end{tikzpicture}
```



```
\usetikzlibrary {matrix}
\begin{tikzpicture}
 \matrix (network)
 [matrix of nodes,% 
  nodes in empty cells,
  nodes={outer sep=0pt,circle,minimum size=4pt,draw},
  column sep={1cm,between origins},
  row sep={1cm,between origins}]
 {
   & & & \\
   & & & \\
 |[draw=none]| & |[xshift=1mm]| & |[xshift=-1mm]| & \\
 };
 \foreach \a in {1,...,4}{%
 \draw (network-3-2) -- (network-2-\a);
 \draw (network-3-3) -- (network-2-\a);
 \draw [-stealth] ([yshift=5mm]network-1-\a.north) -- (network-1-\a);
 \foreach \b in {1,...,4}
 \draw (network-1-\a) -- (network-2-\b);
 }
 \draw [stealth-] ([yshift=-5mm]network-3-2.south) -- (network-3-2);
 \draw [stealth-] ([yshift=-5mm]network-3-3.south) -- (network-3-3);
\end{tikzpicture}
```

The following example is adapted from code written by Kjell Magne Fauske, which is based on the following paper: K. Bossley, M. Brown, and C. Harris, Neurofuzzy identification of an autonomous underwater

vehicle, *International Journal of Systems Science*, 1999, 30, 901–913.



```

\usetikzlibrary {arrows,shapes.geometric}
\begin{tikzpicture}
[auto,
 decision/.style={diamond, draw=blue, thick, fill=blue!20,
   text width=4.5em,align=flush center,
   inner sep=1pt},
 block/.style ={rectangle, draw=blue, thick, fill=blue!20,
   text width=5em,align=center, rounded corners,
   minimum height=4em},
 line/.style ={draw, thick, -latex',shorten >=2pt},
 cloud/.style ={draw=red, thick, ellipse,fill=red!20,
   minimum height=2em}]

\matrix [column sep=5mm,row sep=7mm]
{
% row 1
\node [cloud] (expert)  {expert}; &
\node [block] (init)    {initialize model}; &
\node [cloud] (system)  {system}; \\
% row 2
& \node [block] (identify) {identify candidate model}; & \\
% row 3
& \node [block] (update)   {update model}; &
\node [block] (evaluate) {evaluate candidate models}; & \\
% row 4
& \node [decision] (decide) {is best candidate}; & \\
% row 5
& \node [block] (stop)     {stop}; & \\
};

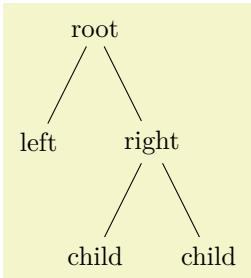
\begin{scope}[every path/.style=line]
\path      (init)    -- (identify);
\path      (identify) -- (evaluate);
\path      (evaluate) -- (decide);
\path      (update)   |- (identify);
\path      (decide)   -| node [near start] {yes} (update);
\path      (decide)   -- node [midway] {no} (stop);
\path [dashed] (expert)  -- (init);
\path [dashed] (system)  -- (init);
\path [dashed] (system)  |- (evaluate);
\end{scope}
\end{tikzpicture}

```

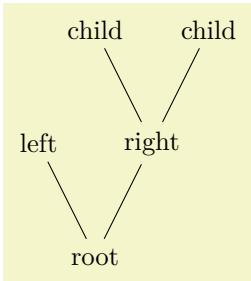
21 Making Trees Grow

21.1 Introduction to the Child Operation

Trees are a common way of visualizing hierarchical structures. A simple tree looks like this:



Admittedly, in reality trees are more likely to grow *upward* and not downward as above. You can tell whether the author of a paper is a mathematician or a computer scientist by looking at the direction their trees grow. A computer scientist's trees will grow downward while a mathematician's tree will grow upward. Naturally, the *correct* way is the mathematician's way, which can be specified as follows:



In TikZ, there are two ways of specifying trees: Using either the `graph` path operation, which is covered in Section 19, or using the `child` path operation, which is covered in the present section. Both methods have their advantages.

In TikZ, trees are specified by adding *children* to a node on a path using the `child` operation:

```
\path ... \node [options] \node [options] \node [options] ...;
```

This operation should directly follow a completed `node` operation or another `child` operation, although it is permissible that the first `child` operation is preceded by options (we will come to that).

When a `node` operation like `node {X}` is followed by `child`, TikZ starts counting the number of child nodes that follow the original `node {X}`. For this, it scans the input and stores away each `child` and its arguments until it reaches a path operation that is not a `child`. Note that this will fix the character codes of all text inside the `child` arguments, which means, in essence, that you cannot use verbatim text inside the nodes inside a `child`. Sorry.

Once the children have been collected and counted, TikZ starts generating the child nodes. For each child of a parent node TikZ computes an appropriate position where the child is placed. For each child, the coordinate system is transformed so that the origin is at this position. Then the `<child path>` is drawn. Typically, the child path just consists of a `node` specification, which results in a node being drawn at the child's position. Finally, an edge is drawn from the first node in the `<child path>` to the parent node.

The optional `foreach` part (note that there is no backslash before `foreach`) allows you to specify multiple children in a single `child` command. The idea is the following: A `\foreach` statement is (internally) used to iterate over the list of `<values>`. For each value in this list, a new `child` is added to the node. The syntax for `<variables>` and for `<values>` is the same as for the `\foreach` statement, see Section ???. For example, when you say

```
node {root} child [red] foreach \name in {1,2} {node {\name}}
```

the effect will be the same as if you had said

```
node {root} child[red] {node {1}} child[red] {node {2}}
```

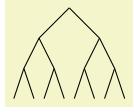
When you write

```
node {root} child[\pos] foreach \name/\pos in {1/left,2/right} {node[\pos] {\name}}
```

the effect will be the same as for

```
node {root} child[left] {node[left] {1}} child[right] {node[right] {2}}
```

You can nest things as in the following example:



```
\begin{tikzpicture}
  [level distance=4mm, level/.style={sibling distance=8mm/#1}]
  \coordinate
    child foreach \x in {0,1}
      {child foreach \y in {0,1}
        {child foreach \z in {0,1}}};
\end{tikzpicture}
```

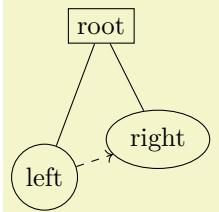
The details and options for this operation are described in the rest of this present section.

21.2 Child Paths and Child Nodes

For each `child` of a root node, its `<child path>` is inserted at a specific location in the picture (the placement rules are discussed in Section 21.5). The first node in the `<child path>`, if it exists, is special and called the *child node*. If there is no first node in the `<child path>`, that is, if the `<child path>` is missing (including the curly braces) or if it does not start with `node` or with `coordinate`, then an empty child node of shape `coordinate` is automatically added.

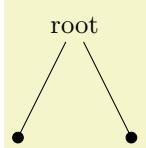
Consider the example `\node {x} child {node {y}} child;`. For the first child, the `<child path>` has the child node `node {y}`. For the second child, no child node is specified and, thus, it is just `coordinate`.

As for any normal node, you can give the child node a name, shift it around, or use options to influence how it is rendered.



```
\usetikzlibrary {shapes.geometric}
\begin{tikzpicture}[sibling distance=15mm]
  \node[rectangle,draw] {root}
    child {node [rectangle,draw] (left node) {left}}
    child {node [ellipse,draw] (right node) {right}};
  \draw[dashed,->] (left node) -- (right node);
\end{tikzpicture}
```

In many cases, the `<child path>` will just consist of a specification of a child node and, possibly, children of this child node. However, the node specification may be followed by arbitrary other material that will be added to the picture, transformed to the child's coordinate system. For your convenience, a move-to `(0,0)` operation is inserted automatically at the beginning of the path. Here is an example:



```
\begin{tikzpicture}
  \node {root}
    child {[fill] circle (2pt)}
    child {[fill] circle (2pt)};
\end{tikzpicture}
```

At the end of the `<child path>` you may add a special path operation called `edge from parent`. If this operation is not given by yourself somewhere on the path, it will be automatically added at the end. This option causes a connecting edge from the parent node to the child node to be added to the path. By giving options to this operation you can influence how the edge is rendered. Also, nodes following the `edge from parent` operation will be placed on this edge, see Section 21.6 for details.

To sum up:

1. The child path starts with a node specification. If it is not there, it is added automatically.
2. The child path ends with a `edge from parent` operation, possibly followed by nodes to be put on this edge. If the operation is not given at the end, it is added automatically.

21.3 Naming Child Nodes

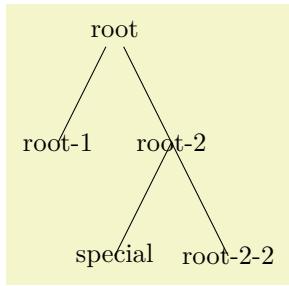
Child nodes can be named like any other node using either the `name` option or the special syntax in which the name of the node is placed in round parentheses between the `node` operation and the node's text.

If you do not assign a name to a child node, TikZ will automatically assign a name as follows: Assume that the name of the parent node is, say, `parent`. (If you did not assign a name to the parent, TikZ will do so itself, but that name will not be user-accessible.) The first child of `parent` will be named `parent-1`, the second child is named `parent-2`, and so on.

This naming convention works recursively. If the second child `parent-2` has children, then the first of these children will be called `parent-2-1` and the second `parent-2-2` and so on.

If you assign a name to a child node yourself, no name is generated automatically (the node does not have two names). However, “counting continues”, which means that the third child of `parent` is called `parent-3` independently of whether you have assigned names to the first and/or second child of `parent`.

Here is an example:

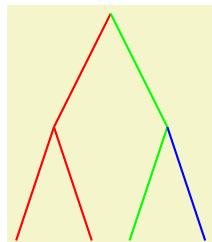


```

\begin{tikzpicture}[sibling distance=15mm]
\node (root) {root}
    child
    child {
        child {coordinate (special)}
        child
    };
\node at (root-1) {root-1};
\node at (root-2) {root-2};
\node at (special) {special};
\node at (root-2-2) {root-2-2};
\end{tikzpicture}
  
```

21.4 Specifying Options for Trees and Children

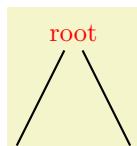
Each `child` may have its own *<options>*, which apply to “the whole child”, including all of its grandchildren. Here is an example:



```

\begin{tikzpicture}
[thick, level 1/.style={sibling distance=15mm},
 level 2/.style={sibling distance=10mm}]
\coordinate
    child[red] {child child}
    child[green] {child child[blue]};
\end{tikzpicture}
  
```

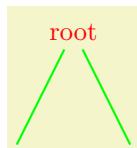
The options of the root node have no effect on the children since the options of a node are always “local” to that node. Because of this, the edges in the following tree are black, not red.



```

\begin{tikzpicture}[thick]
\node [red] {root}
    child
    child;
\end{tikzpicture}
  
```

This raises the problem of how to set options for *all* children. Naturally, you could always set options for the whole path as in `\path [red] node {root} child child;` but this is bothersome in some situations. Instead, it is easier to give the options *before the first child* as follows:



```

\begin{tikzpicture}[thick]
\node [red] {root}
    [green] % option applies to all children
    child
    child;
\end{tikzpicture}
  
```

Here is the set of rules:

1. Options for the whole tree are given before the root node.
2. Options for the root node are given directly to the `node` operation of the root.
3. Options for all children can be given between the root node and the first child.
4. Options applying to a specific child path are given as options to the `child` operation.
5. Options applying to the node of a child, but not to the whole child path, are given as options to the `node` command inside the `<child path>`.

```
\begin{tikzpicture}
\scop
[...] % Options apply to the whole tree
\node[...] {root} % Options apply to the root node only
[...] % Options apply to all children
child[...] % Options apply to this child and all its children
{
  node[...] {} % Options apply to the child node only
  ...
}
child[...] % Options apply to this child and all its children
;
\end{tikzpicture}
```

There are additional styles that influence how children are rendered:

`/tikz/every child` (style, initially empty)

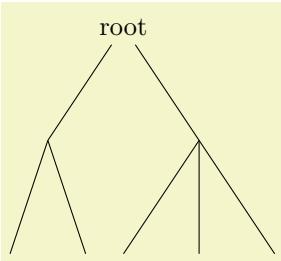
This style is used at the beginning of each child, as if you had given the style's contents as options to the `child` operation.

`/tikz/every child node` (style, initially empty)

This style is used at the beginning of each child node in addition to the `every node` style.

`/tikz/level=<number>` (style, no default, initially empty)

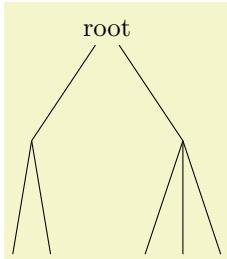
This style is executed at the beginning of each set of children, where `<number>` is the current level in the current tree. For example, when you say `\node {x} child child;`, then `level=1` is used before the first `child`. The style or code of this key will be passed `<number>` as its first parameter. If this first `child` has children itself, then `level=2` would be used for them.



```
\begin{tikzpicture}[level/.style={sibling distance=20mm/#1}]
\node {root}
  child { child child }
  child { child child child };
\end{tikzpicture}
```

`/tikz/level <number>` (style, initially empty)

This style is used in addition to the `level` style. So, when you say `\node {x} child child;`, then the following key list is executed: `level=1,level 1`.



```
\begin{tikzpicture}
[level 1/.style={sibling distance=20mm},
level 2/.style={sibling distance=5mm}]
\node {root}
  child { child child }
  child { child child child };
\end{tikzpicture}
```

21.5 Placing Child Nodes

21.5.1 Basic Idea

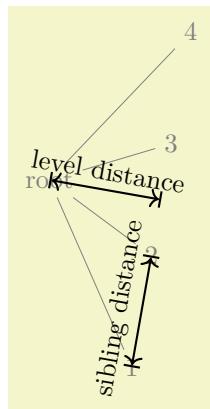
Perhaps the most difficult part in drawing a tree is the correct layout of the children. Typically, the children have different sizes and it is not easy to arrange them in such a manner that not too much space is wasted, the children do not overlap, and they are either evenly spaced or their centers are evenly distributed. Calculating good positions is especially difficult since a good position for the first child may depend on the size of the last child.

In basic TikZ, when you do not make use of the graph drawing facilities explained in Part IV, a comparatively simple approach is taken to placing the children. In order to compute a child's position, all that is taken into account is the number of the current child in the list of children and the number of children in this list. Thus, if a node has five children, then there is a fixed position for the first child, a position for the second child, and so on. These positions *do not depend on the size of the children* and, hence, children can easily overlap. However, since you can use options to shift individual children a bit, this is not as great a problem as it may seem.

Although the placement of the children only depends on their number in the list of children and the total number of children, everything else about the placement is highly configurable. You can change the distance between children (appropriately called the *sibling distance*) and the distance between levels of the tree. These distances may change from level to level. The direction in which the tree grows can be changed globally and for parts of the tree. You can even specify your own “growth function” to arrange children on a circle or along special lines or curves.

21.5.2 Default Growth Function

The default growth function works as follows: Assume that we are given a node and five children. These children will be placed on a line with their centers (or, more generally, with their anchors) spaced apart by the current *sibling distance*. The line is orthogonal to the current *direction of growth*, which is set with the *grow* and *grow'* option (the latter option reverses the ordering of the children). The distance from the line to the parent node is given by the *level distance*.

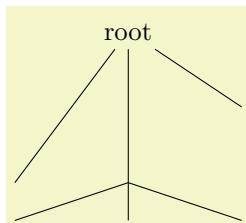


```
\begin{tikzpicture}[sibling distance=15mm, level distance=15mm]
  \path [help lines]
    node (root) {root}
    [grow=-10]
    child {node {1}}
    child {node {2}}
    child {node {3}}
    child {node {4}};
  \draw[|<->|,thick] (root-1.center)
    -- node[above,sloped] {sibling distance} (root-2.center);
  \draw[|<->|,thick] (root.center)
    -- node[above,sloped] {level distance} +(-10:\tikzleveldistance);
\end{tikzpicture}
```

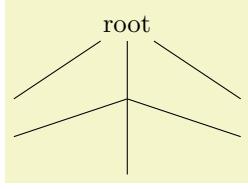
`/tikz/level distance=(distance)`

(no default, initially 15mm)

This key determines the distance between different levels of the tree, more precisely, between the parent and the line on which its children are arranged. When given to a single child, this will set the distance for this child only.



```
\begin{tikzpicture}
  \node (root)
    [level distance=20mm]
    child
    child {
      [level distance=5mm]
      child
      child
      child
    }
    child[level distance=10mm];
\end{tikzpicture}
```

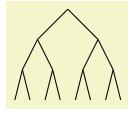


```
\begin{tikzpicture}
[level 1/.style={level distance=10mm},
 level 2/.style={level distance=5mm}]
\node {root}
  child
  child {
    child
    child[level distance=10mm]
    child
  }
  child;
\end{tikzpicture}
```

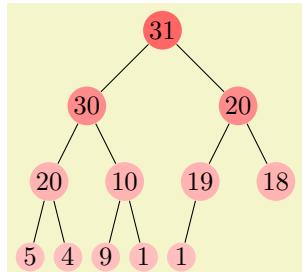
/tikz/sibling distance=(distance)

(no default, initially 15mm)

This key specifies the distance between the anchors of the children of a parent node.



```
\begin{tikzpicture}
[level distance=4mm,
 level 1/.style={sibling distance=8mm},
 level 2/.style={sibling distance=4mm},
 level 3/.style={sibling distance=2mm}]
\coordinate
  child {
    child {child child}
    child {child child}
  }
  child {
    child {child child}
    child {child child}
  };
\end{tikzpicture}
```



```
\begin{tikzpicture}
[level distance=10mm,
 every node/.style={fill=red!60,circle,inner sep=1pt},
 level 1/.style={sibling distance=20mm,nodes={fill=red!45}},
 level 2/.style={sibling distance=10mm,nodes={fill=red!30}},
 level 3/.style={sibling distance=5mm,nodes={fill=red!25}}]
\node {31}
  child {node {30}
    child {node {20}
      child {node {20}}
      child {node {10}}
    }
    child {node {20}
      child {node {19}}
      child {node {18}}
    }
  }
  child {node {20}
    child {node {10}
      child {node {9}}
      child {node {1}}
    }
    child {node {20}
      child {node {19}}
      child {node {missing}}
    }
    child {node {18}}
  };
\end{tikzpicture}
```

/tikz/grow=(direction)

(no default)

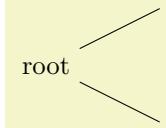
This key is used to define the *(direction)* in which the tree will grow. The *(direction)* can either be an angle in degrees or one of the following special text strings: **down**, **up**, **left**, **right**, **north**, **south**, **east**, **west**, **north east**, **north west**, **south east**, and **south west**. All of these have “their obvious meaning”, so, say, **south west** is the same as the angle -135° .

As a side effect, this option installs the default growth function.

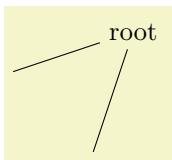
In addition to setting the direction, this option also has a seemingly strange effect: It sets the sibling distance for the current level to 0pt, but leaves the sibling distance for later levels unchanged.

This somewhat strange behaviour has a highly desirable effect: If you give this option before the list of children of a node starts, the “current level” is still the parent level. Each child will be on a later level and, hence, the sibling distance will be as specified originally. This will cause the children to be neatly aligned in a line orthogonal to the given *(direction)*. However, if you give this option locally to a single child, then “current level” will be the same as the child’s level. The zero sibling distance will then cause the child to be placed exactly at a point at distance `level distance` in the direction *(direction)*. However, the children of the child will be placed “normally” on a line orthogonal to the *(direction)*.

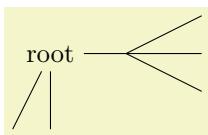
These placement effects are best demonstrated by some examples:



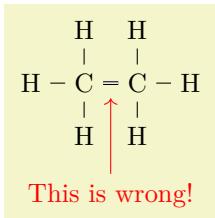
```
\tikz \node {root} [grow=right] child child;
```



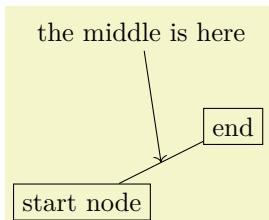
```
\tikz \node {root} [grow=south west] child child;
```



```
\begin{tikzpicture}[level distance=10mm,sibling distance=5mm]
\node {root}
  [grow=down]
  child
  child
  child[grow=right] {
    child child child
  };
\end{tikzpicture}
```



```
\begin{tikzpicture}[level distance=2em]
\node {C}
  child[grow=up]    {node {H}}
  child[grow=left]  {node {H}}
  child[grow=down]  {node {H}}
  child[grow=right] {node {C}}
    child[grow=up]   {node {H}}
    child[grow=right] {node {H}}
    child[grow=down] {node {H}}
    edge from parent[double]
      coordinate (wrong)
  ;
\draw[<,red] ([yshift=-2mm]wrong) -- +(0,-1)
  node[below]{This is wrong!};
\end{tikzpicture}
```



```
\begin{tikzpicture}
\node[rectangle,draw] (a) at (0,0) {start node};
\node[rectangle,draw] (b) at (2,1) {end};

\draw (a) -- (b)
  node[coordinate,midway] {}
  child[grow=100,<-] {node[above]{the middle is here}};
\end{tikzpicture}
```

`/tikz/grow’=(direction)`

(no default)

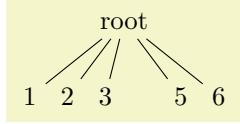
This key has the same effect as `grow`, only the children are arranged in the opposite order.

21.5.3 Missing Children

Sometimes one or more of the children of a node are “missing”. Such a missing child will count as a child with respect to the total number of children and also with respect to the current child count, but it will not be rendered.

`/tikz/missing=<true or false>` (default `true`)

If this option is given to a child, the current child counter is increased, but the child is otherwise ignored. In particular, the normal contents of the child is completely ignored.



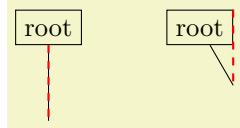
```
\begin{tikzpicture}[level distance=10mm,sibling distance=5mm]
\node [root] [grow=down]
    child { node {1} }
    child { node {2} }
    child { node {3} }
    child[missing] { node {4} }
    child { node {5} }
    child { node {6} };
\end{tikzpicture}
```

21.5.4 Custom Growth Functions

`/tikz/growth parent anchor=<anchor>` (no default, initially `center`)

This key allows you to specify which anchor of the parent node is to be used for computing the children’s position. For example, when there is only one child and the `level distance` is 2cm, then the child node will be placed two centimeters below the `<anchor>` of the parent node. “Being placed” means that the child node’s anchor (which is the anchor specified using the `anchor=` option in the `node` command of the child) is two centimeters below the parent node’s `<anchor>`.

In the following example, the two red lines both have length 1cm.

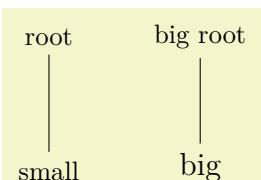


```
\begin{tikzpicture}[level distance=1cm]
\node [rectangle,draw] (a) at (0,0) {root}
[growth parent anchor=south] child;

\node [rectangle,draw] (b) at (2,0) {root}
[growth parent anchor=north east] child;

\draw [red,thick,dashed] (a.south) -- (a-1);
\draw [red,thick,dashed] (b.north east) -- (b-1);
\end{tikzpicture}
```

In the next example, the top and bottom nodes are aligned at the top and the bottom, respectively.



```
\begin{tikzpicture}
[level distance=2cm,growth parent anchor=north,
every node/.style={anchor=north,rectangle,draw},
every child node/.style={anchor=south}]
\node at (0,0) {root} child {node {small}};
\node at (2,0) {big root} child {node {\large big}};
\end{tikzpicture}
```

`/tikz/growth function=<macro name>` (no default, initially an internal function)

This rather low-level option allows you to set a new growth function. The `<macro name>` must be the name of a macro without parameters. This macro will be called for each child of a node. The initial function is an internal function that corresponds to downward growth.

The effect of executing the macro should be the following: It should transform the coordinate system in such a way that the origin becomes the place where the current child should be anchored. When the macro is called, the current coordinate system will be set up such that the anchor of the parent node is in the origin. Thus, in each call, the `<macro name>` must essentially do a shift to the child’s origin. When the macro is called, the T_EX counter `\tikznumberofchildren` will be set to the total number of children of the parent node and the counter `\tikznumberofcurrentchild` will be set to the number of the current child.

The macro may, in addition to shifting the coordinate system, also transform the coordinate system further. For example, it could be rotated or scaled.

Additional growth functions are defined in the library, see Section 77.

21.6 Edges From the Parent Node

Every child node is connected to its parent node via a special kind of edge called the `edge from parent`. This edge is added to the `<child path>` when the following path operation is encountered:

```
\path ... edge from parent[<options>] ... ;
```

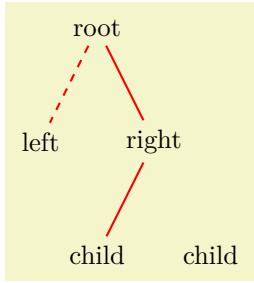
This path operation can only be used inside `<child paths>` and should be given at the end, possibly followed by `<node specifications>` like `node {a}`. If a `<child path>` does not contain this operation, it will be added at the end of the `<child path>` automatically.

By default, this operation does the following:

1. The following style is executed:

```
/tikz/edge from parent (style, initially draw)
```

This style is inserted right before the `edge from parent` path and before the `<options>` are inserted.



```
\begin{tikzpicture}
  [edge from parent/.style={draw,red,thick}]
  \node {root}
    child {node {left}}
    child {node {right}}
      child {node {child}}
      child {node {child} edge from parent[draw=none]}
  ;
\end{tikzpicture}
```

2. Next, the `<options>` are executed.

3. Next, the text stored in the following key is inserted:

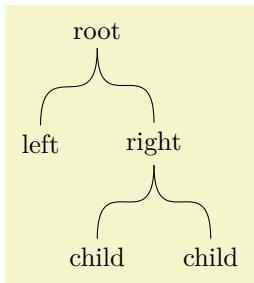
```
/tikz/edge from parent path=<path> (no default, initially code shown below)
```

This option allows you to set the `edge from parent` path to a new path. Initially, this path is the following:

```
(\tikzparentnode\tikzparentanchor) -- (\tikzchildnode\tikzchildanchor)
```

The `\tikzparentnode` is a macro that will expand to the name of the parent node. This works even when you have not assigned a name to the parent node, in this case an internal name is automatically generated. The `\tikzchildnode` is a macro that expands to the name of the child node. The two `...anchor` macros are empty by default. So, what is essentially inserted is just the path segment `(\tikzparentnode) --(\tikzchildnode)`; which is exactly an edge from the parent to the child.

You can modify this edge from parent path to achieve all sorts of effects. For example, we could replace the straight line by a curve as follows:



```
\begin{tikzpicture}[level distance=15mm, sibling distance=15mm,
  edge from parent path=
  {(\tikzparentnode.south) .. controls +(0,-1) and +(0,1)
   .. (\tikzchildnode.north)}]
  \node {root}
    child {node {left}}
    child {node {right}}
      child {node {child}}
      child {node {child}}
  ;
\end{tikzpicture}
```

Further useful `edge from parent` paths are defined in the tree library, see Section 77.

The nodes in a `<node specification>` following the `edge from parent` path command get executed as if the `pos` option had been added to all these nodes, see also Section 17.8.

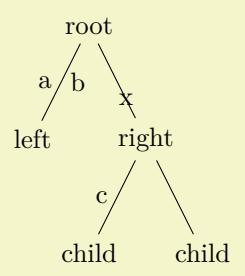
As an example, consider the following code:

```
\node (root) {} child {node (child) {} edge to parent node {label}};
```

The `edge to parent` operation and the following `node` operation will, together, have the same effect as if we had said:

```
(root) -- (child) node [pos=0.5] {label}
```

Here is a more complicated example:



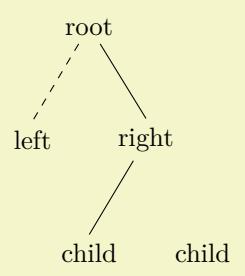
```
\begin{tikzpicture}
\node {root}
child {
    node {left}
    edge from parent
    node[left] {a}
    node[right] {b}
}
child {
    node {right}
    child {
        node {child}
        edge from parent
        node[left] {c}
    }
    child {node {child}}
    edge from parent
    node[near end] {x}
};
\end{tikzpicture}
```

As said before, the anchors in the default `edge from parent` path are empty. However, you can set them using the following keys:

`/tikz/child anchor=<anchor>` (no default, initially `border`)

Specifies the anchor where the edge from parent meets the child node by setting the macro `\tikzchildanchor` to `.<anchor>`.

If you specify `border` as the `<anchor>`, then the macro `\tikzchildanchor` is set to the empty string. The effect of this is that the edge from the parent will meet the child on the border at an automatically calculated position.



```
\begin{tikzpicture}
\node {root}
[child anchor=north]
child {node {left} edge from parent[dashed]}
child {node {right}
    child {node {child}}
    child {node {child} edge from parent[draw=None]}
};
\end{tikzpicture}
```

`/tikz/parent anchor=<anchor>` (no default, initially `border`)

This option works the same way as the `child anchor`, only for the parent.

All of the above describes the standard functioning of the `edge from parent` command. You may, however, sometimes need even more fine-grained control (the graph drawing engine needs it, for instance). In such cases the following key gives you complete control:

`/tikz/edge from parent macro=<macro>` (no default)

The $\langle macro \rangle$ gets expanded each time the `edge from parent` path operation is used. This $\langle macro \rangle$ must take two parameters and must expand to some text that is subsequently parsed by the parser. The first parameter will be the set of $\langle options \rangle$ that where passed to the `edge from parent` command, the second parameter will be the $\langle node specifications \rangle$ that following the command.

The standard behaviour of drawing a straight line from the parent node to the child node could be achieved by setting the $\langle macro \rangle$ to the following:

```
\def\mymacro#1#2{  
    [style=edge from parent, #1]  
    (\tikzparentnode\tikzparentanchor) -- #2 (\tikzchildnode\tikzchildanchor)  
}
```

Note that `#2` is placed between `--` and the node to ensure that nodes are put “on top” of the line.

22 Plots of Functions

A warning before we get started: *If you are looking for an easy way to create a normal plot of a function with scientific axes, ignore this section and instead look at the `pgfplots` package or at the `datavisualization` command from Part ??.*

22.1 Overview

TikZ can be used to create plots of functions, a job that is normally handled by powerful programs like GNUPLOT or MATHEMATICA. These programs can produce two different kinds of output: First, they can output a complete plot picture in a certain format (like PDF) that includes all low-level commands necessary for drawing the complete plot (including axes and labels). Second, they can usually also produce “just plain data” in the form of a long list of coordinates. Most of the powerful programs consider it a to be “a bit boring” to just output tabled data and very much prefer to produce fancy pictures. Nevertheless, when coaxed, they can also provide the plain data.

The advantage of creating plots directly using TikZ is *consistency*: Plots created using TikZ will automatically have the same styling and fonts as those used in the rest of a document – something that is hard to do right when an external program gets involved. Other problems people encounter with external programs include: Formulas will look different, if they can be rendered at all; line widths will usually be too thick or too thin; scaling effects upon inclusion can create a mismatch between sizes in the plot and sizes in the text; the automatic grid generated by most programs is mostly distracting; the automatic ticks generated by most programs are cryptic numerics (try adding a tick reading “ π ” at the right point); most programs make it very easy to create “chart junk” in a most convenient fashion; arrows and plot marks will almost never match the arrows used in the rest of the document. This list is not exhaustive, unfortunately.

There are basically three ways of creating plots using TikZ:

1. Use the `plot` path operation. How this works is explained in the present section. This is the most “basic” of the three options and forces you to do a lot of things “by hand” like adding axes or ticks.
2. Use the `datavisualization` path command, which is documented in Part ???. This command is much more powerful than the `plot` path operation and produces complete plots including axes and ticks. The downside is that you cannot use it to “just” quickly plot a simple curve (or, more precisely, it is hard to use it in this way).
3. Use the `pgfplots` package, which is basically an alternative to the `datavisualization` command. While the underlying philosophy of this package is not as “ambitious” as that of the command `datavisualization`, it is somewhat more mature, has a simpler design, and wider support base.

22.2 The Plot Path Operation

The `plot` path operation can be used to append a line or curve to the path that goes through a large number of coordinates. These coordinates are either given in a simple list of coordinates, read from some file, or they are computed on the fly.

The syntax of the `plot` comes in different versions.

```
\path ... --plot<further arguments> ... ;
```

This operation plots the curve through the coordinates specified in the `<further arguments>`. The current (sub)path is simply continued, that is, a line-to operation to the first point of the curve is implicitly added. The details of the `<further arguments>` will be explained in a moment.

```
\path ... plot<further arguments> ... ;
```

This operation plots the curve through the coordinates specified in the `<further arguments>` by first “moving” to the first coordinate of the curve.

The `<further arguments>` are used in different ways to specifying the coordinates of the points to be plotted:

1. `--plot[<local options>]coordinates{<coordinate 1> <coordinate 2> ... <coordinate n>}`
2. `--plot[<local options>]file{<filename>}`

3. `--plot[<local options>]<coordinate expression>`
4. `--plot[<local options>]function{<gnuplot formula>}`

These different ways are explained in the following.

22.3 Plotting Points Given Inline

Points can be given directly in the TeX-file as in the following example:



```
\tikz \draw plot coordinates {(0,0) (1,1) (2,0) (3,1) (2,1) (10:2cm)};
```

Here is an example showing the difference between `plot` and `--plot`:



```
\begin{tikzpicture}
\draw (0,0) -- (1,1) plot coordinates {(2,0) (4,0)};
\draw[color=red,xshift=5cm]
(0,0) -- (1,1) -- plot coordinates {(2,0) (4,0)};
\end{tikzpicture}
```

22.4 Plotting Points Read From an External File

The second way of specifying points is to put them in an external file named `<filename>`. Currently, the only file format that TikZ allows is the following: Each line of the `<filename>` should contain one line starting with two numbers, separated by a space. A line may also be empty or, if it starts with `#` or `%` it is considered empty. For such lines, a “new data set” is started, typically resulting in a new subpath being started in the plot (see Section ?? on how to change this behaviour, if necessary). For lines containing two numbers, they must be separated by a space. They may be followed by arbitrary text, which is ignored, *except* if it is `o` or `u`. In the first case, the point is considered to be an *outlier* and normally also results in a new subpath being started. In the second case, the point is considered to be *undefined*, which also results in a new subpath being started. Again, see Section ?? on how to change this, if necessary. (This is exactly the format that GNUPLOT produces when you say `set terminal table`.)

```
\tikz \draw plot[mark=x,smooth] file {plots/pgfmanual-sine.table};
```

The file `plots/pgfmanual-sine.table` reads:

```
#Curve 0, 20 points
#x y type
0.00000 0.00000 i
0.52632 0.50235 i
1.05263 0.86873 i
1.57895 0.99997 i
...
9.47368 -0.04889 i
10.00000 -0.54402 i
```

It was produced from the following source, using `gnuplot`:

```
set table "../plots/pgfmanual-sine.table"
set format "%5f"
set samples 20
plot [x=0:10] sin(x)
```

The `<local options>` of the `plot` operation are local to each plot and do not affect other plots “on the same path”. For example, `plot[yshift=1cm]` will locally shift the plot 1cm upward. Remember, however, that most options can only be applied to paths as a whole. For example, `plot[red]` does not have the effect of making the plot red. After all, you are trying to “locally” make part of the path red, which is not possible.

22.5 Plotting a Function

When you plot a function, the coordinates of the plot data can be computed by evaluating a mathematical expression. Since PGF comes with a mathematical engine, you can specify this expression and then have TikZ produce the desired coordinates for you, automatically.

Since this case is quite common when plotting a function, the syntax is easy: Following the `plot` command and its local options, you directly provide a *(coordinate expression)*. It looks like a normal coordinate, but inside you may use a special macro, which is `\x` by default, but this can be changed using the `variable` option. The *(coordinate expression)* is then evaluated for different values for `\x` and the resulting coordinates are plotted.

Note that you will often have to put the *x*- or *y*-coordinate inside braces, namely whenever you use an expression involving a parenthesis.

The following options influence how the *(coordinate expression)* is evaluated:

`/tikz/variable=<macro>` (no default, initially `x`)

Sets the macro whose value is set to the different values when *(coordinate expression)* is evaluated.

`/tikz/samples=<number>` (no default, initially 25)

Sets the number of samples used in the plot.

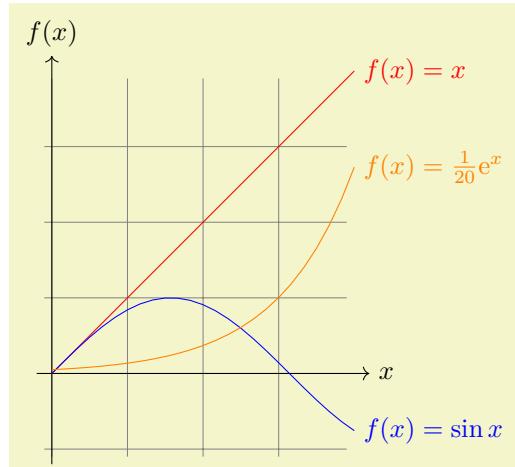
`/tikz/domain=<start>:<end>` (no default, initially -5:5)

Sets the domain from which the samples are taken.

`/tikz/samples at=<sample list>` (no default)

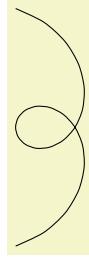
This option specifies a list of positions for which the variable should be evaluated. For instance, you can say `samples at={1,2,8,9,10}` to have the variable evaluated exactly for values 1, 2, 8, 9, and 10. You can use the `\foreach` syntax, so you can use `...` inside the *(sample list)*.

When this option is used, the `samples` and `domain` option are overruled. The other way round, setting either `samples` or `domain` will overrule this option.

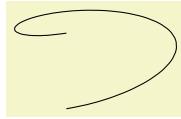


```
\begin{tikzpicture}[domain=0:4]
\draw[very thin,color=gray] (-0.1,-1.1) grid (3.9,3.9);
\draw[->] (-0.2,0) -- (4.2,0) node[right] {$f(x) = x$};
\draw[->] (0,-1.2) -- (0,4.2) node[above] {$f(x)$};

\draw[color=red] plot (\x,\x) node[right] {$f(x) = x$};
% \x r means to convert '\x' from degrees to _radians:
\draw[color=blue] plot (\x,{sin(\x r)}) node[right] {$f(x) = \sin x$};
\draw[color=orange] plot (\x,{0.05*exp(\x)}) node[right] {$f(x) = \frac{1}{20} e^x$};
\end{tikzpicture}
```



```
\tikz \draw [scale=0.5,domain=-3.141:3.141,smooth,variable=\t]
plot ({\t*sin(\t r)},{\t*cos(\t r)});
```



```
\tikz \draw [domain=0:360,smooth,variable=\t]
plot ({sin(\t)},{\t/360},{cos(\t)});
```

22.6 Plotting a Function Using Gnuplot

Often, you will want to plot points that are given via a function like $f(x) = x \sin x$. Unfortunately, \TeX does not really have enough computational power to generate the points of such a function efficiently (it is a text processing program, after all). However, if you allow it, \TeX can try to call external programs that can easily produce the necessary points. Currently, TikZ knows how to call GNUPLOT.

When TikZ encounters your operation `plot[id=<id>] function{x*sin(x)}` for the first time, it will create a file called `<prefix><id>.gnuplot`, where `<prefix>` is `\jobname`. by default, that is, the name of your main `.tex` file. If no `<id>` is given, it will be empty, which is alright, but it is better when each plot has a unique `<id>` for reasons explained in a moment. Next, TikZ writes some initialization code into this file followed by `plot x*sin(x)`. The initialization code sets up things such that the `plot` operation will write the coordinates into another file called `<prefix><id>.table`. Finally, this table file is read as if you had said `plot file{<prefix><id>.table}`.

For the plotting mechanism to work, two conditions must be met:

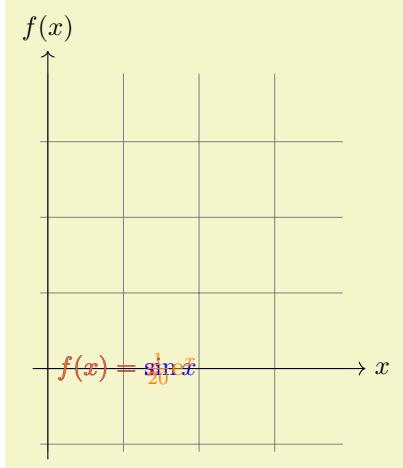
1. You must have allowed \TeX to call external programs. This is often switched off by default since this is a security risk (you might, without knowing, run a \TeX file that calls all sorts of “bad” commands). To enable this “calling external programs” a command line option must be given to the \TeX program. Usually, it is called something like `shell-escape` or `enable-write18`. For example, for my `pdflatex` the option `--shell-escape` can be given.
2. You must have installed the `gnuplot` program and \TeX must find it when compiling your file.

Unfortunately, these conditions will not always be met. Especially if you pass some source to a coauthor and the coauthor does not have GNUPLOT installed, he or she will have trouble compiling your files.

For this reason, TikZ behaves differently when you compile your graphic for the second time: If upon reaching `plot[id=<id>] function{...}` the file `<prefix><id>.table` already exists *and* if the `<prefix><id>.gnuplot` file contains what TikZ thinks that it “should” contain, the `.table` file is immediately read without trying to call a `gnuplot` program. This approach has the following advantages:

1. If you pass a bundle of your `.tex` file and all `.gnuplot` and `.table` files to someone else, that person can \TeX the `.tex` file without having to have `gnuplot` installed.
2. If the `\write18` feature is switched off for security reasons (a good idea), then, upon the first compilation of the `.tex` file, the `.gnuplot` will still be generated, but not the `.table` file. You can then simply call `gnuplot` “by hand” for each `.gnuplot` file, which will produce all necessary `.table` files.
3. If you change the function that you wish to plot or its domain, TikZ will automatically try to regenerate the `.table` file.
4. If, out of laziness, you do not provide an `id`, the same `.gnuplot` will be used for different plots, but this is not a problem since the `.table` will automatically be regenerated for each plot on-the-fly. *Note: If you intend to share your files with someone else, always use an id, so that the file can be typeset without having GNUPLOT installed.* Also, having unique ids for each plot will improve compilation speed since no external programs need to be called, unless it is really necessary.

When you use `plot function{<gnuplot formula>}`, the `<gnuplot formula>` must be given in the gnuplot syntax, whose details are beyond the scope of this manual. Here is the ultra-condensed essence: Use `x` as the variable and use the C-syntax for normal plots, use `t` as the variable for parametric plots. Here are some examples:



```
\begin{tikzpicture}[domain=0:4]
\draw[very thin,color=gray] (-0.1,-1.1) grid (3.9,3.9);
\draw[->] (-0.2,0) -- (4.2,0) node[right] {$x$};
\draw[->] (0,-1.2) -- (0,4.2) node[above] {$f(x)$};

\draw[color=red] plot[id=x] function{x} node[right] {$f(x) = x$};
\draw[color=blue] plot[id=sin] function{sin(x)} node[right] {$f(x) = \sin x$};
\draw[color=orange] plot[id=exp] function{0.05*exp(x)} node[right] {$f(x) = \frac{1}{20} e^x$};
\end{tikzpicture}
```

The plot is influenced by the following options: First, the options `samples` and `domain` explained earlier. Second, there are some more specialized options.

`/tikz/parametric=<boolean>` (default `true`)

Sets whether the plot is a parametric plot. If true, then `t` must be used instead of `x` as the parameter and two comma-separated functions must be given in the `<gnuplot formula>`. An example is the following:

```
\tikz \draw[scale=0.5,domain=-3.141:3.141,smooth]
plot[parametric,id=parametric-example] function{t*sin(t),t*cos(t)};
```

`/tikz/range=<start>:<end>` (no default)

This key sets the range of the plot. If set, all points whose `y`-coordinates lie outside this range will be considered to be outliers and will cause jumps in the plot, by default:

```
\tikz \draw[scale=0.5,domain=-3.141:3.141, samples=100, smooth, range=-3:3]
plot[id=tan-example] function{tan(x)};
```

`/tikz/yrange=<start>:<end>` (no default)

Same as `range`.

`/tikz/xrange=<start>:<end>` (no default)

Set the `x`-range. This makes sense only for parametric plots.

```
\tikz \draw[scale=0.5,domain=-3.141:3.141,smooth,xrange=0:1]
plot[parametric,id=parametric-example-cut] function{t*sin(t),t*cos(t)};
```

`/tikz/id=<id>` (no default)

Sets the identifier of the current plot. This should be a unique identifier for each plot (though things will also work if it is not, but not as well, see the explanations above). The `<id>` will be part of a filename, so it should not contain anything fancy like `*` or `$`.

`/tikz/prefix=<prefix>`

(no default)

The `<prefix>` is put before each plot file name. The default is `\jobname.`, but if you have many plots, it might be better to use, say `plots/` and have all plots placed in a directory. You have to create the directory yourself.

`/tikz/raw gnuplot`

(no value)

This key causes the `<gnuplot formula>` to be passed on to GNUPLOT without setting up the samples or the `plot` operation. Thus, you could write

```
plot[raw gnuplot,id=raw-example] function[set samples 25; plot sin(x)]
```

This can be useful for complicated things that need to be passed to GNUPLOT. However, for really complicated situations you should create a special external generating GNUPLOT file and use the `file-` syntax to include the table “by hand”.

The following styles influence the plot:

`/tikz/every plot`

(style, initially empty)

This style is installed in each plot, that is, as if you always said

```
plot[every plot,...]
```

This is most useful for globally setting a prefix for all plots by saying:

```
\tikzset{every plot/.style={prefix=plots/}}
```

22.7 Placing Marks on the Plot

As we saw already, it is possible to add *marks* to a plot using the `mark` option. When this option is used, a copy of the plot mark is placed on each point of the plot. Note that the marks are placed *after* the whole path has been drawn/filled/shaded. In this respect, they are handled like text nodes.

In detail, the following options govern how marks are drawn:

`/tikz/mark=<mark mnemonic>`

(no default)

Sets the mark to a mnemonic that has previously been defined using the `\pgfdeclareplotmark`. By default, `*`, `+`, and `x` are available, which draw a filled circle, a plus, and a cross as marks. Many more marks become available when the library `plotmarks` is loaded. Section 66.6 lists the available plot marks.

One plot mark is special: the `ball` plot mark is available only in TikZ. The `ball color` option determines the ball’s color. Do not use this option with a large number of marks since it will take very long to render in PostScript.

Option	Effect
<code>mark=ball</code>	

`/tikz/mark repeat=<r>`

(no default)

This option tells TikZ that only every *r*th mark should be drawn.

```
\tikz \draw plot[mark=x,mark repeat=3,smooth] file {plots/pgfmanual-sine.table};
```

`/tikz/mark phase=<p>`

(no default)

This option tells TikZ that the first mark to be draw should be the *p*th, followed by the *(p+r)*th, then the *(p+2r)*th, and so on.

```
\tikz \draw plot[mark=x,mark repeat=3,mark phase=6,smooth] file {plots/pgfmanual-sine.table};
```

`/tikz/mark indices=<list>` (no default)

This option allows you to specify explicitly the indices at which a mark should be placed. Counting starts with 1. You can use the `\foreach` syntax, that is, `...` can be used.

```
\tikz \draw plot[mark=x,mark indices={1,4,...,10,11,12,...,16,20},smooth]
  file {plots/pgfmanual-sine.table};
```

`/tikz/mark size=<dimension>` (no default)

Sets the size of the plot marks. For circular plot marks, `<dimension>` is the radius, for other plot marks `<dimension>` should be about half the width and height.

This option is not really necessary, since you achieve the same effect by specifying `scale=<factor>` as a local option, where `<factor>` is the quotient of the desired size and the default size. However, using `mark size` is a bit faster and more natural.

`/tikz/every mark` (style, no value)

This style is installed before drawing plot marks. For example, you can scale (or otherwise transform) the plot mark or set its color.

`/tikz/mark options=<options>` (no default)

Redefines `every mark` such that it sets `{<options>}`.

```
\tikz \fill[fill=blue!20]
  plot[mark=triangle*,mark options={color=blue,rotate=180}]
    file{plots/pgfmanual-sine.table} |- (0,0);
```

`/tikz/no marks` (style, no value)

Disables markers (the same as `mark=none`).

`/tikz/no markers` (style, no value)

Disables markers (the same as `mark=none`).

22.8 Smooth Plots, Sharp Plots, Jump Plots, Comb Plots and Bar Plots

There are different things the `plot` operation can do with the points it reads from a file or from the inlined list of points. By default, it will connect these points by straight lines. However, you can also use options to change the behavior of `plot`.

`/tikz/sharp plot` (no value)

This is the default and causes the points to be connected by straight lines. This option is included only so that you can “switch back” if you “globally” install, say, `smooth`.

`/tikz/smooth` (no value)

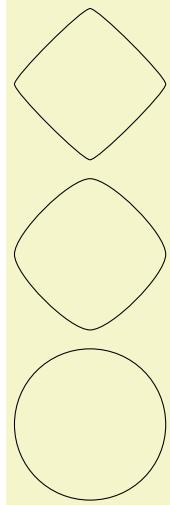
This option causes the points on the path to be connected using a smooth curve:

```
\tikz\draw plot[smooth] file{plots/pgfmanual-sine.table};
```

Note that the smoothing algorithm is not very intelligent. You will get the best results if the bending angles are small, that is, less than about 30° and, even more importantly, if the distances between points are about the same all over the plotting path.

`/tikz/tension=<value>` (no default)

This option influences how “tight” the smoothing is. A lower value will result in sharper corners, a higher value in more “round” curves. A value of 1 results in a circle if four points at quarter-positions on a circle are given. The default is 0.55. The “correct” value depends on the details of plot.



```
\begin{tikzpicture}[smooth cycle]
\draw plot [tension=0.2]
coordinates{(0,0) (1,1) (2,0) (1,-1)};
\draw[yshift=-2.25cm] plot [tension=0.5]
coordinates{(0,0) (1,1) (2,0) (1,-1)};
\draw[yshift=-4.5cm] plot [tension=1]
coordinates{(0,0) (1,1) (2,0) (1,-1)};
\end{tikzpicture}
```

/tikz/smooth cycle

(no value)

This option causes the points on the path to be connected using a closed smooth curve.



```
\tikz[scale=0.5]
\draw plot[smooth cycle] coordinates{(0,0) (1,0) (2,1) (1,2)}
plot coordinates{(0,0) (1,0) (2,1) (1,2)} -- cycle;
```

/tikz/const plot

(no value)

This option causes the points on the path to be connected using piecewise constant series of lines:

```
\tikz\draw plot[const plot] file{plots/pgfmanual-sine.table};
```

/tikz/const plot mark left

(no value)

Just an alias for /tikz/const plot.

```
\tikz\draw plot[const plot mark left,mark=*] file{plots/pgfmanual-sine.table};
```

/tikz/const plot mark right

(no value)

A variant of /tikz/const plot which places its mark on the right ends:

```
\tikz\draw plot[const plot mark right,mark=*] file{plots/pgfmanual-sine.table};
```

/tikz/const plot mark mid

(no value)

A variant of /tikz/const plot which places its mark in the middle of the horizontal lines:

```
\tikz\draw plot[const plot mark mid,mark=*] file{plots/pgfmanual-sine.table};
```

More precisely, it generates vertical lines in the middle between each pair of consecutive points. If the mesh width is constant, this leads to symmetrically placed marks (“middle”).

/tikz/jump mark left

(no value)

This option causes the points on the path to be drawn using piecewise constant, non-connected series of lines. If there are any marks, they will be placed on left open ends:

```
\tikz\draw plot[jump mark left, mark=*] file{plots/pgfmanual-sine.table};
```

/tikz/jump mark right (no value)

This option causes the points on the path to be drawn using piecewise constant, non-connected series of lines. If there are any marks, they will be placed on right open ends:

```
\tikz\draw plot[jump mark right, mark=*] file{plots/pgfmanual-sine.table};
```

/tikz/jump mark mid (no value)

This option causes the points on the path to be drawn using piecewise constant, non-connected series of lines. If there are any marks, they will be placed in the middle of the horizontal line segments:

```
\tikz\draw plot[jump mark mid, mark=*] file{plots/pgfmanual-sine.table};
```

In case of non-constant mesh widths, the same remarks as for `const plot mark mid` apply.

/tikz/ycomb (no value)

This option causes the `plot` operation to interpret the plotting points differently. Instead of connecting them, for each point of the plot a straight line is added to the path from the x -axis to the point, resulting in a sort of “comb” or “bar diagram”.

```
\tikz\draw[ultra thick] plot[ycomb,thin,mark=*] file{plots/pgfmanual-sine.table};
```



```
\begin{tikzpicture}[ycomb]
  \draw[color=red, line width=6pt]
    plot coordinates{(0,1) (.5,1.2) (1,.6) (1.5,.7) (2,.9)};
  \draw[color=red!50, line width=4pt, xshift=3pt]
    plot coordinates{(0,1.2) (.5,1.3) (1,.5) (1.5,.2) (2,.5)};
\end{tikzpicture}
```

/tikz/xcomb (no value)

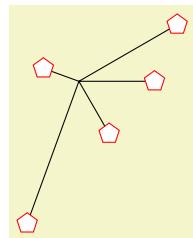
This option works like `ycomb` except that the bars are horizontal.



```
\tikz \draw plot[xcomb,mark=x] coordinates{(1,0) (0.8,0.2) (0.6,0.4) (0.2,1)};
```

/tikz/polar comb (no value)

This option causes a line from the origin to the point to be added to the path for each plot point.

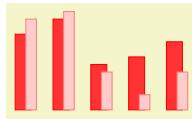


```
\tikz \draw plot[polar comb,
  mark=pentagon*,mark options={fill=white,draw=red},mark size=4pt]
  coordinates {(0:1cm) (30:1.5cm) (160:.5cm) (250:2cm) (-60:.8cm)};
```

/tikz/ybar (no value)

This option produces fillable bar plots. It is thus very similar to `ycomb`, but it employs rectangular shapes instead of line-to operations. It thus allows to use any fill or pattern style.

```
\tikz\draw[draw=blue,fill=blue!60!black] plot[ybar] file{plots/pgfmanual-sine.table};
```



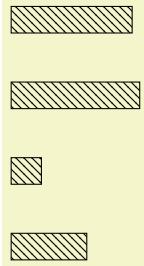
```
\begin{tikzpicture}[ybar]
  \draw[color=red,fill=red!80,bar width=6pt]
    plot coordinates{(0,1) (.5,1.2) (1,.6) (1.5,.7) (2,.9)};
  \draw[color=red!50,fill=red!20,bar width=4pt,bar shift=3pt]
    plot coordinates{(0,1.2) (.5,1.3) (1,.5) (1.5,.2) (2,.5)};
\end{tikzpicture}
```

The use of `bar width` and `bar shift` is explained in the `plot handlers` library documentation, section 66.4. Please refer to page 595.

/tikz/xbar

(no value)

This option works like `ybar` except that the bars are horizontal.

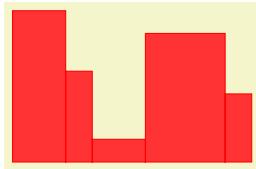


```
\usetikzlibrary {patterns}
\tikz \draw[pattern=north west lines] plot[xbar]
coordinates{(1,0) (0.4,1) (1.7,2) (1.6,3)};
```

/tikz/ybar interval

(no value)

As `/tikz/ybar`, this option produces vertical bars. However, bars are centered at coordinate *intervals* instead of interval edges, and the bar's width is also determined relatively to the interval's length:



```
\begin{tikzpicture}[ybar interval,x=10pt]
  \draw[color=red,fill=red!80]
    plot coordinates{(0,2) (2,1.2) (3,.3) (5,1.7) (8,.9) (9,.9)};
\end{tikzpicture}
```

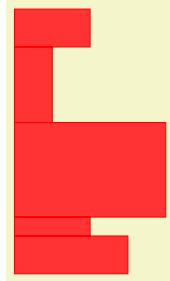
Since there are N intervals $[x_i, x_{i+1}]$ for given $N + 1$ coordinates, you will always have one coordinate more than bars. The last y value will be ignored.

You can configure relative shifts and relative bar widths, which is explained in the `plot handlers` library documentation, section 66.4. Please refer to page 596.

/tikz/xbar interval

(no value)

Works like `ybar interval`, but for horizontal bar plots.

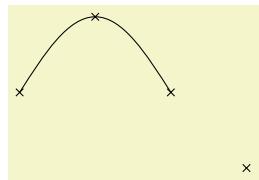


```
\begin{tikzpicture}[xbar interval,x=0.5cm,y=0.5cm]
  \draw[color=red,fill=red!80]
    plot coordinates {(3,0) (2,1) (4,1.5) (1,4) (2,6) (2,7)};
\end{tikzpicture}
```

/tikz/only marks

(no value)

This option causes only marks to be shown; no path segments are added to the actual path. This can be useful for quickly adding some marks to a path.



```
\tikz \draw (0,0) sin (1,1) cos (2,0)
  plot[only marks,mark=x] coordinates{(0,0) (1,1) (2,0) (3,-1)};
```

23 Transparency

23.1 Overview

Normally, when you paint something using any of TikZ's commands (this includes stroking, filling, shading, patterns, and images), the newly painted objects totally obscure whatever was painted earlier in the same area.

You can change this behaviour by using something that can be thought of as “(semi)transparent colors”. Such colors do not completely obscure the background, rather they blend the background with the new color. At first sight, using such semitransparent colors might seem quite straightforward, but the math going on in the background is quite involved and the correct handling of transparency fills some 64 pages in the PDF specification.

In the present section, we start with the different ways of specifying “how transparent” newly drawn objects should be. The simplest way is to just specify a percentage like “60% transparent”. A much more general way is to use something that I call a *fading*, also known as a soft mask or a mask.

At the end of the section we address the problem of creating so-called *transparency groups*. This problem arises when you paint over a position several times with a semitransparent color. Sometimes you want the effect to accumulate, sometimes you do not.

Note: Transparency is best supported by the pdftEX driver. The SVG driver also has some support. For PostScript output, opacity is rendered correctly only with the most recent versions of Ghostscript. Printers and other programs will typically ignore the opacity setting.

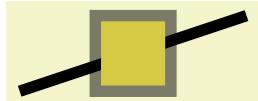
23.2 Specifying a Uniform Opacity

Specifying a stroke and/or fill opacity is quite easy using the following options.

`/tikz/draw opacity=<value>` (no default)

This option sets “how transparent” lines should be. A value of 1 means “fully opaque” or “not transparent at all”, a value of 0 means “fully transparent” or “invisible”. A value of 0.5 yields lines that are semitransparent.

Note that when you use PostScript as your output format, this option works only with recent versions of Ghostscript.



```
\begin{tikzpicture}[line width=1ex]
\draw (0,0) -- (3,1);
\filldraw [fill=yellow!80!black,draw opacity=0.5] (1,0) rectangle (2,1);
\end{tikzpicture}
```

Note that the `draw opacity` options only sets the opacity of drawn lines. The opacity of fillings is set using the option `fill opacity` (documented in Section 15.5.3). The option `opacity` sets both at the same time.

`/tikz-opacity=<value>` (no default)

Sets both the drawing and filling opacity to `<value>`.

The following predefined styles make it easier to use this option:

`/tikz/transparent` (style, no value)

Makes everything totally transparent and, hence, invisible.



```
\tikz{\fill[red] (0,0) rectangle (1,0.5);
\fill[transparent,red] (0.5,0) rectangle (1.5,0.25); }
```

`/tikz/ultra nearly transparent` (style, no value)

Makes everything, well, ultra nearly transparent.



```
\tikz{\fill[red] (0,0) rectangle (1,0.5);
\fill[ultra nearly transparent] (0.5,0) rectangle (1.5,0.25); }
```

`/tikz/very nearly transparent` (style, no value)



```
\tikz{\fill[red] (0,0) rectangle (1,0.5);
      \fill[very nearly transparent] (0.5,0) rectangle (1.5,0.25); }
```

`/tikz/nearly transparent` (style, no value)



```
\tikz{\fill[red] (0,0) rectangle (1,0.5);
      \fill[nearly transparent] (0.5,0) rectangle (1.5,0.25); }
```

`/tikz/semitransparent` (style, no value)



```
\tikz{\fill[red] (0,0) rectangle (1,0.5);
      \fill[semitransparent] (0.5,0) rectangle (1.5,0.25); }
```

`/tikz/nearly opaque` (style, no value)



```
\tikz{\fill[red] (0,0) rectangle (1,0.5);
      \fill[nearly opaque] (0.5,0) rectangle (1.5,0.25); }
```

`/tikz/very nearly opaque` (style, no value)



```
\tikz{\fill[red] (0,0) rectangle (1,0.5);
      \fill[very nearly opaque] (0.5,0) rectangle (1.5,0.25); }
```

`/tikz/ultra nearly opaque` (style, no value)



```
\tikz{\fill[red] (0,0) rectangle (1,0.5);
      \fill[ultra nearly opaque] (0.5,0) rectangle (1.5,0.25); }
```

`/tikz/opaque` (style, no value)

This yields completely opaque drawings, which is the default.

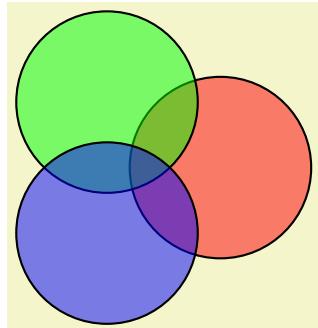


```
\tikz{\fill[red] (0,0) rectangle (1,0.5);
      \fill[opaque] (0.5,0) rectangle (1.5,0.25); }
```

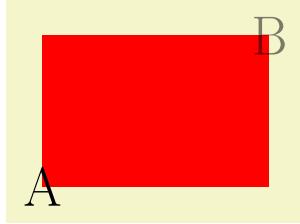
`/tikz/fill opacity=<value>` (no default)

This option sets the opacity of fillings. In addition to filling operations, this opacity also applies to text and images.

Note, again, that when you use PostScript as your output format, this option works only with recent versions of Ghostscript.



```
\begin{tikzpicture}[thick, fill opacity=0.5]
  \filldraw[fill=red] (0:1cm) circle (12mm);
  \filldraw[fill=green] (120:1cm) circle (12mm);
  \filldraw[fill=blue] (-120:1cm) circle (12mm);
\end{tikzpicture}
```

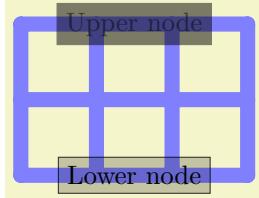


```
\begin{tikzpicture}
  \fill[red] (0,0) rectangle (3,2);
  \node [fill opacity=0.5] at (0,0) {\huge A};
  \node [fill opacity=0.5] at (3,2) {\huge B};
\end{tikzpicture}
```

`/tikz/text opacity=<value>`

(no default)

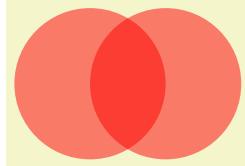
Sets the opacity of text labels, overriding the `fill opacity` setting.



```
\begin{tikzpicture}[every node/.style={fill,draw}]
  \draw[line width=2mm,blue!50,line cap=round] (0,0) grid (3,2);

  \node[opacity=0.5] at (1.5,2) {Upper node};
  \node[draw opacity=0.8,fill opacity=0.2,text opacity=1]
    at (1.5,0) {Lower node};
\end{tikzpicture}
```

Note the following effect: If you set up a certain opacity for stroking or filling and you stroke or fill the same area twice, the effect accumulates:



```
\begin{tikzpicture}[fill opacity=0.5]
  \fill[red] (0,0) circle (1);
  \fill[red] (1,0) circle (1);
\end{tikzpicture}
```

Often, this is exactly what you intend, but not always. You can use transparency groups, see the end of this section, to change this.

23.3 Blend Modes

A *blend mode* specifies how colors mix when you paint on a canvas. Normally, if you paint a red box on a green circle, the red color will completely replace the green circle. However, in some situations you might also wish the red color to somehow “mix” or “blend” with the green circle. We already saw that, using transparency, we can draw something without completely obscuring the background. *Blending* is a similar operation, only here we mix colors in more complicated ways.

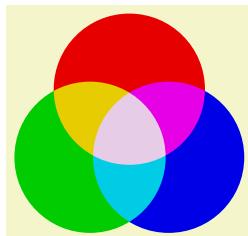
Note: Blending is a rather “advanced” feature of PDF. Most renderers, let alone printers, will have trouble rendering blending correctly.

`/tikz/blend mode=<mode>`

(no default)

Sets the current blend mode to `<mode>`. Here `<mode>` must be one of the modes listed below. More details on these modes can also be found in Section 7.2.4 of the PDF Specification, version 1.7.

In the following example, the blend mode is only used and set inside a transparency group (see also Section 23.5). This is because most renderers (viewing programs) have trouble rendering blending correctly otherwise. For instance, at the time of writing, the versions of Adobe’s Reader and Apple’s Preview render the following drawing very differently, if the transparency group is not used in the following example.



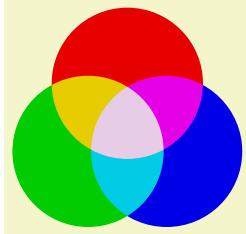
```
\tikz {
  \begin{scope} [
    transparency group
    \begin{scope} [blend mode=screen]
      \fill[red!90!black] (90:.6) circle (1);
      \fill[green!80!black] (210:.6) circle (1);
      \fill[blue!90!black] (330:.6) circle (1);
    \end{scope}
  \end{scope}
}
```

Because of the trouble with rendering blending correctly outside transparency groups, there is a special key that establishes a transparency group and sets a blend mode simultaneously:

`/tikz/blend group=<mode>`

(no default)

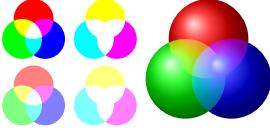
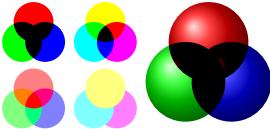
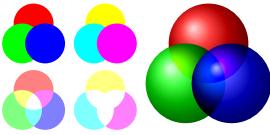
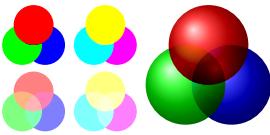
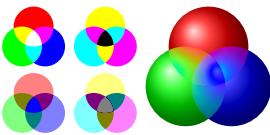
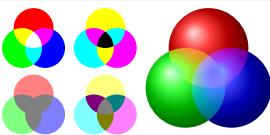
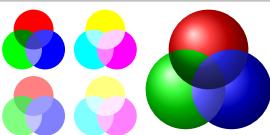
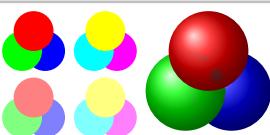
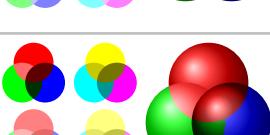
This key can only be used with a scope (like `transparency group`). It will cause the current scope to become a transparency group and, inside this group, the blend mode will be set to `<mode>`.



```
\tikz [blend group=screen] {
  \fill[red!90!black] (90:.6) circle (1);
  \fill[green!80!black] (210:.6) circle (1);
  \fill[blue!90!black] (330:.6) circle (1);
}
```

Here is an overview of the effects of the different available blend modes. In the examples, we always have three circles drawn on top of each other (as in the example code earlier): We start with a triple of pure red, green, and blue. Below it, we have a triple of light versions of these three colors (`red!50`, `green!50`, and `blue!50`). Next comes the triple yellow, cyan, and magenta; again with a triple of light versions below it. The large example consists of three balls (produced using `ball color`) having the colors red, green, and blue, are drawn on top of each other just like the circles.

Example	Mode	Explanations quoted from Table 7.2 of the PDF Specification, Version 1.7
	normal	When painting a pixel with a some color (called the “source color”), the background color (called the “backdrop”) is completely ignored.
	multiply	Multiples the backdrop and source color values. The result color is always at least as dark as either of the two constituent colors. Multiplying any color with black produces black; multiplying with white leaves the original color unchanged. Painting successive overlapping objects with a color other than black or white produces progressively darker colors.
	screen	Multiples the complements of the backdrop and source color values, then complements the result. The result color is always at least as light as either of the two constituent colors. Screening any color with white produces white; screening with black leaves the original color unchanged. The effect is similar to projecting multiple photographic slides simultaneously onto a single screen.
	overlay	Multiples or screens the colors, depending on the backdrop color value. Source colors overlay the backdrop while preserving its highlights and shadows. The backdrop color is not replaced but is mixed with the source color to reflect the lightness or darkness of the backdrop.
	darken	Selects the darker of the backdrop and source colors. The backdrop is replaced with the source where the source is darker; otherwise, it is left unchanged.

	lighten	Selects the lighter of the backdrop and source colors. The backdrop is replaced with the source where the source is lighter; otherwise, it is left unchanged.
	color dodge	Brightens the backdrop color to reflect the source color. Painting with black produces no changes.
	color burn	Darkens the backdrop color to reflect the source color. Painting with white produces no change.
	hard light	Multiplies or screens the colors, depending on the source color value. The effect is similar to shining a harsh spotlight on the backdrop.
	soft light	Darkens or lightens the colors, depending on the source color value. The effect is similar to shining a diffused spotlight on the backdrop.
	difference	Subtracts the darker of the two constituent colors from the lighter color. Painting with white inverts the backdrop color; painting with black produces no change.
	exclusion	Produces an effect similar to that of the Difference mode but lower in contrast. Painting with white inverts the backdrop color; painting with black produces no change.
	hue	Creates a color with the hue of the source color and the saturation and luminosity of the backdrop color.
	saturation	Creates a color with the saturation of the source color and the hue and luminosity of the backdrop color. Painting with this mode in an area of the backdrop that is a pure gray (no saturation) produces no change.
	color	Creates a color with the hue and saturation of the source color and the luminosity of the backdrop color. This preserves the gray levels of the backdrop and is useful for coloring monochrome images or tinting color images.
	luminosity	Creates a color with the luminosity of the source color and the hue and saturation of the backdrop color. This produces an inverse effect to that of the Color mode.

23.4 Fadings

For complicated graphics, uniform transparency settings are not always sufficient. Suppose, for instance, that while you paint a picture, you want the transparency to vary smoothly from completely opaque to completely transparent. This is a “shading-like” transparency. For such a form of transparency I will use the term *fading* (as a noun). They are also known as *soft masks*, *opacity masks*, *masks*, or *soft clips*.

23.4.1 Creating Fadings

How do we specify a fading? This is a bit of an art since the underlying mechanism is quite powerful, but a bit difficult to use.

Let us start with a bit of terminology. A *fading* specifies for each point of an area the transparency of that point. This transparency can be any number between 0 and 1. A *fading picture* is a normal graphic that, in a way to be described in a moment, determines the transparency of points inside the fading. Each fading has an underlying fading picture.

The fading picture is a normal graphic drawn using any of the normal graphic drawing commands. A fading and its fading picture are related as follows: Given any point of the fading, the transparency of this point is determined by the luminosity of the fading picture at the same position. The luminosity of a point determines “how bright” the point is. The brighter the point in the fading picture, the more opaque is the point in the fading. In particular, a white point of the fading picture is completely opaque in the fading and a black point of the fading picture is completely transparent in the fading. (The background of the fading picture is always transparent in the fading as if the background were black.)

It is rather counter-intuitive that a *white* pixel of the fading picture will be *opaque* in the fading and a *black* pixel will be *transparent*. For this reason, TikZ defines a color called `transparent` that is the same as `black`. The nice thing about this definition is that the color `transparent!<percentage>` in the fading picture yields a pixel that is `<percentage>` percent transparent in the fading.

Turning a fading picture into a normal picture is achieved using the following commands, which are *only defined in the library*, namely the library `fadings`. So, to use them, you have to say `\usetikzlibrary{fadings}` first.

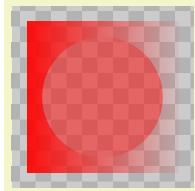
```
\begin{tikzfadingfrompicture}[\langle options\rangle]
  <environment contents>
\end{tikzfadingfrompicture}
```

This command works like a `{tikzpicture}`, only the picture is not shown, but instead a fading is defined based on this picture. To set the name of the picture, use the `name` option (which is normally used to set the name of a node).

`/tikz/name=\{\langle name\rangle\}` (no default)

Use this option with the `{tikzfadingfrompicture}` environment to set the name of the fading. You *must* provide this option.

The following shading is 2cm by 2cm and gets more and more transparent from left to right, but is 50% transparent for a large circle in the middle.

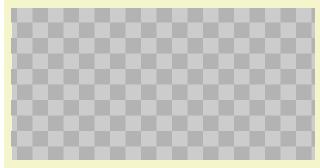


```
\use tikzlibrary {fadings,patterns}
\begin{tikzfadingfrompicture}[name=fade right]
  \shade [left color=transparent!0,
          right color=transparent!100] (0,0) rectangle (2,2);
  \fill [transparent!50] (1,1) circle (0.7);
\end{tikzfadingfrompicture}

% Now we use the fading in another picture:
\begin{tikzpicture}
  % Background
  \fill [black!20] (-1.2,-1.2) rectangle (1.2,1.2);
  \pattern [pattern=checkerboard,pattern color=black!30]
            (-1.2,-1.2) rectangle (1.2,1.2);

  \fill [path fading=fade right,red] (-1,-1) rectangle (1,1);
\end{tikzpicture}
```

In the next example we create a fading picture that contains some text. When the fading is used, we only see the shading “through it”.



```
\usetikzlibrary {fadings,patterns}
\begin{tikzfadingfrompicture}[name=tikz]
  \node [text=transparent!20]
    {\fontencoding{T1}\fontfamily{ptm}\fontsize{45}{45}\bfseries\selectfont Ti\emph{k}Z};
\end{tikzfadingfrompicture}

% Now we use the fading in another picture:
\begin{tikzpicture}
  \fill [black!20] (-2,-1) rectangle (2,1);
  \path [pattern=checkerboard,pattern color=black!30]
    (-2,-1) rectangle (2,1);

  \shade[path fading=tikz,fit fading=false,
         left color=blue,right color=black]
    (-2,-1) rectangle (2,1);
\end{tikzpicture}
```

The same effect can also be achieved using knockout groups, see Section 23.5.

```
\tikzfadingfrompicture[<options>]
  <environment contents>
\endtikzfadingfrompicture
```

The plainTeX version of the environment.

```
\starttikzfadingfrompicture[<options>]
  <environment contents>
\stoptikzfadingfrompicture
```

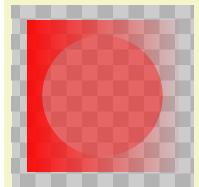
The ConTeXt version of the environment.

```
\tikzfading[<options>]
```

This command is used to define a fading similarly to the way a shading is defined. In the *<options>* you should

1. use the `name=<name>` option to set a name for the fading,
2. use the `shading` option to set the name of the shading that you wish to use,
3. extra options for setting the colors of the shading (typically you will set them to the color `transparent!<percentage>`).

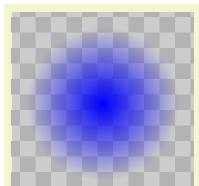
Then, a new fading named *<name>* will be created based on the shading.



```
\usetikzlibrary {fadings,patterns}
\tikzfading[name=fade right,
            left color=transparent!0,
            right color=transparent!100]

% Now we use the fading in another picture:
\begin{tikzpicture}
  % Background
  \fill [black!20] (-1.2,-1.2) rectangle (1.2,1.2);
  \path [pattern=checkerboard,pattern color=black!30]
    (-1.2,-1.2) rectangle (1.2,1.2);

  \fill [red,path fading=fade right] (-1,-1) rectangle (1,1);
\end{tikzpicture}
```



```
\usetikzlibrary {fadings,patterns}
\tikzfading[name=fade out,
            inner color=transparent!0,
            outer color=transparent!100]

% Now we use the fading in another picture:
\begin{tikzpicture}
  % Background
  \fill [black!20] (-1.2,-1.2) rectangle (1.2,1.2);
  \path [pattern=checkerboard,pattern color=black!30]
    (-1.2,-1.2) rectangle (1.2,1.2);

  \fill [blue,path fading=fade out] (-1,-1) rectangle (1,1);
\end{tikzpicture}
```

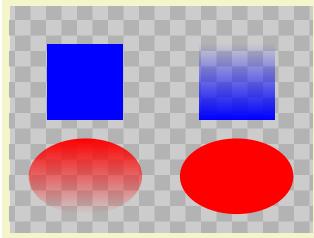
23.4.2 Fading a Path

A fading specifies for each pixel of a certain area how transparent this pixel will be. The following options are used to install such a fading for the current scope or path.

`/tikz/path fading=<name>`

(default scope's setting)

This option tells TikZ that the current path should be faded with the fading `<name>`. If no `<name>` is given, the `<name>` set for the whole scope is used. Similarly to options like `draw` or `fill`, this option is reset for each path, so you have to add it to each path that should be faded. You can also specify `none` as `<name>`, in which case fading for the path will be switched off in case it has been switched on by previous options or styles.



```
\usetikzlibrary {fadings,patterns}
\begin{tikzpicture}[path fading=south]
% Checker board
\fill [black!20] (0,0) rectangle (4,3);
\pattern [pattern=checkerboard,pattern color=black!30]
(0,0) rectangle (4,3);

\fill [color=blue] (0.5,1.5) rectangle +(1,1);
\fill [color=blue,path fading=north] (2.5,1.5) rectangle +(1,1);

\fill [color=red,path fading] (1,0.75) ellipse (.75 and .5);
\fill [color=red] (3,0.75) ellipse (.75 and .5);
\end{tikzpicture}
```

`/tikz/fit fading=<boolean>`

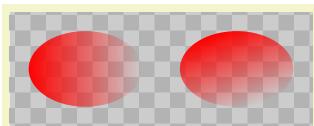
(default `true`, initially `true`)

When set to `true`, the fading is shifted and resized (in exactly the same way as a shading) so that it covers the current path. When set to `false`, the fading is only shifted so that it is centered on the path's center, but it is not resized. This can be useful for special-purpose fadings, for instance when you use a fading to "punch out" something.

`/tikz/fading transform=<transformation options>`

(no default)

The `<transformation options>` are applied to the fading before it is used. For instance, if `<transformation options>` is set to `rotate=90`, the fading is rotated by 90 degrees.



```
\usetikzlibrary {fadings,patterns}
\begin{tikzpicture}[path fading=fade down]
% Checker board
\fill [black!20] (0,0) rectangle (4,1.5);
\path [pattern=checkerboard,pattern color=black!30] (0,0) rectangle (4,1.5);

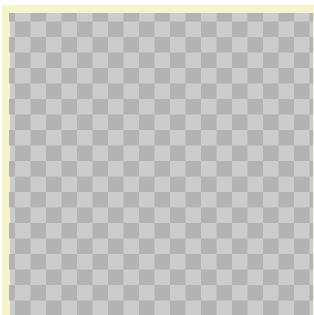
\fill [red,path fading,fading transform={rotate=90}] (1,0.75) ellipse (.75 and .5);
\fill [red,path fading,fading transform={rotate=30}] (3,0.75) ellipse (.75 and .5);
\end{tikzpicture}
```

`/tikz/fading angle=<degree>`

(no default)

A shortcut for `fading transform={rotate=<degree>}`.

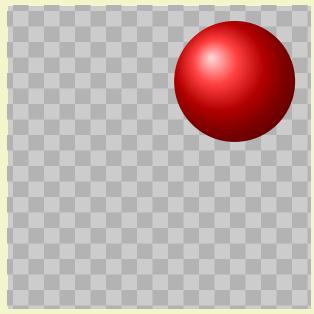
Note that you can "fade just about anything". In particular, you can fade a shading.



```
\usetikzlibrary {fadings,patterns}
\begin{tikzpicture}
% Checker board
\fill [black!20] (0,0) rectangle (4,4);
\path [pattern=checkerboard,pattern color=black!30] (0,0) rectangle (4,4);

\shade [ball color=blue,path fading=south] (2,2) circle (1.8);
\end{tikzpicture}
```

The `fade inside` of the following example is more transparent in the middle than on the outside.

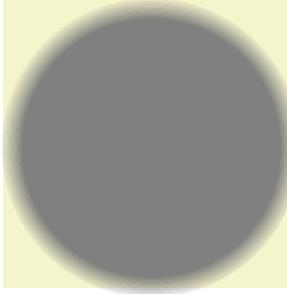


```
\usetikzlibrary {fadings,patterns}
\tikzfading[name=fade inside,
            inner color=transparent!80,
            outer color=transparent!30]
\begin{tikzpicture}
    % Checker board
    \fill [black!20] (0,0) rectangle (4,4);
    \path [pattern=checkerboard,pattern color=black!30] (0,0) rectangle (4,4);

    \shade [ball color=red] (3,3) circle (0.8);
    \shade [ball color=white,path fading=fade inside] (2,2) circle (1.8);
\end{tikzpicture}
```

Note that adding the `path fading` option to a node fades the (background) path, not the text itself. To fade the text, you need to use a scope fading (see below).

Note that using fadings in conjunction with patterns can create visually rather pleasing effects:



```
\usetikzlibrary {fadings,patterns,shadows}
\tikzfading[name=middle,
            top color=transparent!50,
            bottom color=transparent!50,
            middle color=transparent!20]
\begin{tikzpicture}
    \node [circle,circular drop shadow,
           pattern=horizontal lines dark blue,
           path fading=south,
           minimum size=3.6cm] {};
    \pattern [path fading=north,
              pattern=horizontal lines dark gray]
    (0,0) circle (1.8cm);
    \pattern [path fading=middle,
              pattern=crosshatch dots light steel blue]
    (0,0) circle (1.8cm);
\end{tikzpicture}
```

23.4.3 Fading a Scope

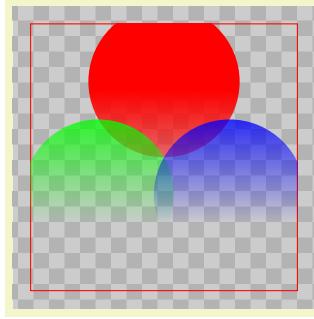
In addition to fading individual paths, you may also wish to “fade a scope”, that is, you may wish to install a fading that is used globally to specify the transparency for all objects drawn inside a scope. This effect can also be thought of as a “soft clip” and it works in a similar way: You add the `scope fading` option to a path in a scope – typically the first one – and then all subsequent drawings in the scope are faded. You will use a `transparency group` in conjunction, see the end of this section.

`/tikz/scope fading=<fading>`

(no default)

In principle, this key works in exactly the same way as the `path fading` key. The only difference is, that the effect of the fading will persist after the current path till the end of the scope. Thus, the `<fading>` is applied to all subsequent drawings in the current scope, not just to the current path. In this regard, the option works very much like the `clip` option. (Note, however, that, unlike the `clip` option, fadings do not accumulate unless a `transparency group` is used.)

The keys `fit fading` and `fading transform` have the same effect as for `path fading`. Also that, just as for `path fading`, providing the `scope fading` option with a `{scope}` only sets the name of the fading to be used. You have to explicitly provide the `scope fading` with a path to actually install a fading.



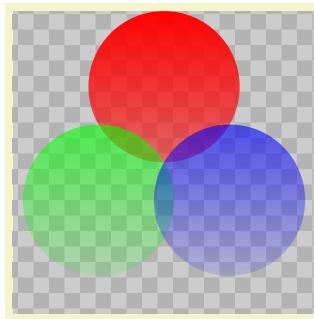
```
\usetikzlibrary {fadings,patterns}
\begin{tikzpicture}
  \fill [black!20] (-2,-2) rectangle (2,2);
  \pattern [pattern=checkerboard,pattern color=black!30]
    (-2,-2) rectangle (2,2);

% The bounding box of the shading:
\draw [red] (-50bp,-50bp) rectangle (50bp,50bp);

\path [scope fading=south,fit fading=false] (0,0);
% fading is centered at its natural size

\fill[red] ( 90:1) circle (1);
\fill[green] (210:1) circle (1);
\fill[blue] (330:1) circle (1);
\end{tikzpicture}
```

In the following example we resize the fading to the size of the whole picture:



```
\usetikzlibrary {fadings,patterns}
\begin{tikzpicture}
  \fill [black!20] (-2,-2) rectangle (2,2);
  \pattern [pattern=checkerboard,pattern color=black!30]
    (-2,-2) rectangle (2,2);

\path [scope fading=south] (-2,-2) rectangle (2,2);

\fill[red] ( 90:1) circle (1);
\fill[green] (210:1) circle (1);
\fill[blue] (330:1) circle (1);
\end{tikzpicture}
```

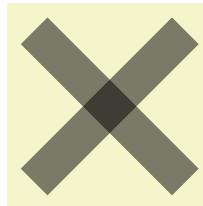
Scope fadings are also needed if you wish to fade a node.

This is some text that will fade out as we go right and down. It is pretty hard to achieve this effect in other ways.

```
\usetikzlibrary {fadings}
\tikz \node [scope fading=south,fading angle=45,text width=3.5cm]
{
  This is some text that will fade out as we go right
  and down. It is pretty hard to achieve this effect in
  other ways.
};
```

23.5 Transparency Groups

Consider the following cross and sign. They “look wrong” because we can see how they were constructed, while this is not really part of the desired effect.



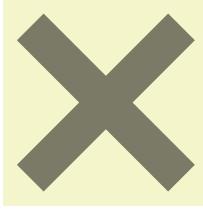
```
\begin{tikzpicture} [opacity=.5]
  \draw [line width=5mm] (0,0) -- (2,2);
  \draw [line width=5mm] (2,0) -- (0,2);
\end{tikzpicture}
```



```
\usetikzlibrary {shapes.symbols}
\begin{tikzpicture}
  \node at (0,0) [forbidden sign,line width=2ex,draw=red,fill=white] {Smoking};

  \node [opacity=.5]
    at (2,0) [forbidden sign,line width=2ex,draw=red,fill=white] {Smoking};
\end{tikzpicture}
```

Transparency groups are used to render them correctly:



```
\begin{tikzpicture} [opacity=.5]
\begin{scope} [transparency group]
\draw [line width=5mm] (0,0) -- (2,2);
\draw [line width=5mm] (2,0) -- (0,2);
\end{scope}
\end{tikzpicture}
```



```
\usetikzlibrary {shapes.symbols}
\begin{tikzpicture}
\node at (0,0) [forbidden sign,line width=2ex,draw=red,fill=white] {Smoking};

\begin{scope} [opacity=.5,transparency group]
\node at (2,0) [forbidden sign,line width=2ex,draw=red,fill=white]
{Smoking};
\end{scope}
\end{tikzpicture}
```

`/tikz/transparency group=[<options>]` (no default)

This option can be given to a `scope`. It will have the following effect: The scope's contents is stroked / filled “ignoring any outside transparency”. This means, all previous transparency settings are ignored (you can still set transparency inside the group, but never mind). For instance, in the forbidden sign example, the whole sign is first painted (conceptually) like the image on the left hand side. Note that some pixels of the sign are painted multiple times (up to three times), but only the last color “wins”.

Then, when the scope is finished, it is painted as a whole. The `fill` transparency settings are now applied to the resulting picture. For instance, the pixel that has been painted three times is just red at the end, so this red color will be blended with whatever is “behind” the group on the page.



```
\usetikzlibrary {patterns,shapes.symbols}
\begin{tikzpicture}
\pattern [pattern=checkerboard,pattern color=black!15] (-1,-1) rectangle (3,1);
\node at (0,0) [forbidden sign,line width=2ex,draw=red,fill=white] {Smoking};

\begin{scope} [transparency group,opacity=.5]
\node at (2,0) [forbidden sign,line width=2ex,draw=red,fill=white]
{Smoking};
\end{scope}
\end{tikzpicture}
```

Note that in the example, the `opacity=.5` is not active inside the transparency group: The group is only established at beginning of the scope and all options given to the `{scope}` environment are set before the group is established. To change the opacity *inside* the group, you need to open another scope inside it or use the `opacity` key with a command inside the group:



```
\usetikzlibrary {patterns,shapes.symbols}
\begin{tikzpicture}
\pattern [pattern=checkerboard,pattern color=black!15] (-1,-1) rectangle (3,1);
\node at (0,0) [forbidden sign,line width=2ex,draw=red,fill=white] {Smoking};

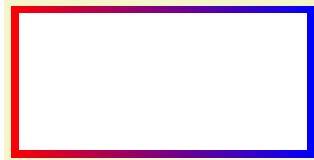
\begin{scope} [transparency group,opacity=.5]
\node (s) at (2,0) [forbidden sign,line width=2ex,draw=red,fill=white]
{Smoking};

\draw [opacity=.5, line width=2ex, blue] (1.2,0) -- (2.8,0);
\end{scope}
\end{tikzpicture}
```

The `<options>` are a list of comma-separated options:

- **knockout** When this option is given inside the `<options>`, the group becomes a so-called *knockout* group. This means, essentially, that inside the group everything is painted as if the “`opacity`” of a line or area were just another color channel. In particular, if you paint a pixel with `opacity 0` inside a knockout group, this pixel becomes perfectly transparent immediately. In contrast, painting a pixel with something of `opacity 0` normally has no effect.

Not all renderers, let alone printers, will support this. At the time of writing, Apple's Preview will not show the following correctly (you should see the text TikZ in the middle):



```
\begin{tikzpicture}
  \shade [left color=red,right color=blue] (-2,-1) rectangle (2,1);
  \begin{scope}[transparency group=knockout]
    \fill [white] (-1.9,-.9) rectangle (1.9,.9);
    \node [opacity=0,font=\fontencoding{T1}\fontfamily{ptm}\fontsize{45}{45}\bfseries]{Ti\emph{k}Z};
  \end{scope}
\end{tikzpicture}
```

In the example, we first draw a large shading and then, inside the transparency group “overwrite” most of this shading by a big white rectangle. The interesting part is the text of the node, which has opacity 0. Normally, this would mean that nothing is shown. However, in a knockout group, we “paint” the text with an “opacity zero” color. The effect is that part of the totally opaque white rectangle gets overwritten by a perfectly transparent area (namely exactly the area taken up by the pixels of the text). When this whole knockout group is then placed on top of the shading, the shading will “shine through” at the knocked-out pixels.

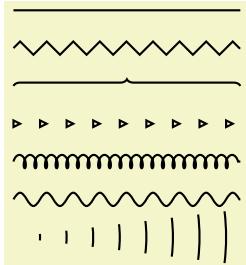
- `isolated=false` A group can be isolated or not. By default, they are isolated, since this is typically what you want. For details on what isolated groups are, exactly, see Section 7.3.4 of the PDF Specification, version 1.7.

Note that when a transparency group is created, TikZ must correctly determine the size of the material inside the group. Usually, this is no problem, but when you use things like `overlay` or `transform canvas`, trouble may result. In this case, please consult Section ?? on how to sidestep this problem in such cases.

24 Decorated Paths

24.1 Overview

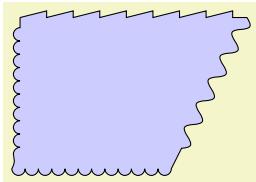
Decorations are a general concept to make (sub)paths “more interesting”. Before we have a look at the details, let us have a look at some examples:



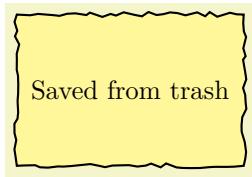
```
\usetikzlibrary {decorations.pathmorphing, decorations.pathreplacing, decorations.shapes, }
\begin{tikzpicture}[thick]
  \draw
  \draw[decorate,decoration=zigzag]
  \draw[decorate,decoration=brace]
  \draw[decorate,decoration=triangles]
  \draw[decorate,decoration={coil,segment length=4pt}]
  \draw[decorate,decoration={coil,aspect=0}]
  \draw[decorate,decoration={expanding waves,angle=7}]
\end{tikzpicture}
```



```
\usetikzlibrary {decorations.pathmorphing}
\begin{tikzpicture}
  \node [fill=red!20,draw,decorate,decoration={bumps,mirror}, minimum height=1cm]
    {Bumpy};
\end{tikzpicture}
```



```
\usetikzlibrary {decorations.pathmorphing}
\begin{tikzpicture}
  \filldraw[fill=blue!20]
  decorate [decoration=saw] { -- (3,3) }
  decorate [decoration={coil,aspect=0}] { -- (2,1) }
  decorate [decoration=bumps] { -| (0,3) };
\end{tikzpicture}
```



```
\usetikzlibrary {decorations.pathmorphing}
\begin{tikzpicture}
  \node [fill=yellow!50,draw,thick, minimum height=2cm, minimum width=3cm,
        decorate, decoration={random steps,segment length=3pt,amplitude=1pt}]
    {Saved from trash};
\end{tikzpicture}
```

The general idea of decorations is the following: First, you construct a path using the usual path construction commands. The resulting path is, in essence, a series of straight and curved lines. Instead of directly using this path for filling or drawing, you can then specify that it should form the basis for a decoration. In this case, depending on which decoration you use, a new path is constructed “along” the path you specified. For instance, with the `zigzag` decoration, the new path is a zigzagging line that goes along the old path.

Let us have a look at an example: In the first picture, we see a path that consists of a line, an arc, and a line. In the second picture, this path has been used as the basis of a decoration.

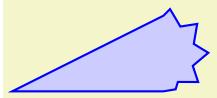


```
\usetikzlibrary {decorations.pathmorphing}
\tikz \fill
  [fill=blue!20,draw=blue,thick] (0,0) -- (2,1) arc (90:-90:.5) -- cycle;
```



```
\usetikzlibrary {decorations.pathmorphing}
\tikz \fill [decorate,decoration=zigzag]
  [fill=blue!20,draw=blue,thick] (0,0) -- (2,1) arc (90:-90:.5) -- cycle;
```

It is also possible to decorate only a subpath (the exact syntax will be explained later in this section).



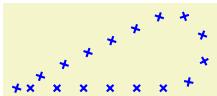
```
\usetikzlibrary {decorations.pathmorphing}
\begin{tikzpicture}
\fill [decoration=zigzag]
[fill=blue!20,draw=blue,thick] (0,0) -- (2,1)
decorate { arc (90:-90:.5) } -- cycle;
\end{tikzpicture}
```

The `zigzag` decoration will be called a *path morphing* decoration because it morphs a path into a different, but topologically equivalent path. Not all decorations are path morphing; rather there are three kinds of decorations.

1. The just-mentioned *path morphing* decorations morph the path in the sense that what used to be a straight line might afterwards be a squiggly line or might have bumps. However, a line is still a line and path deforming decorations do not change the number of subpaths.

Examples of such decorations are the `snake` or the `zigzag` decoration. Many such decorations are defined in the library `decorations.pathmorphing`.

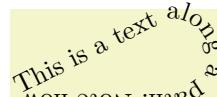
2. *Path replacing* decorations completely replace the path by a different path that is only “loosely based” on the original path. For instance, the `crosses` decoration replaces a path by a path consisting of a sequence of crosses. Note how in the following example filling the path has no effect since the path consist only of (numerous) unconnected straight line subpaths:



```
\usetikzlibrary {decorations.shapes}
\begin{tikzpicture}
\fill [decoration=crosses]
[fill=blue!20,draw=blue,thick] (0,0) -- (2,1) arc (90:-90:.5) -- cycle;
\end{tikzpicture}
```

Examples of path replacing decorations are `crosses` or `ticks` or `shape backgrounds`. Such decorations are defined in the library `decorations.pathreplacing`, but also in `decorations.shapes`.

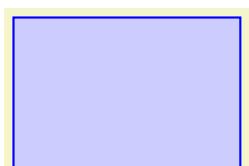
3. *Path removing* decorations completely remove the to-be-decorated path. Thus, they have no effect on the main path that is being constructed. Instead, they typically have numerous *side effects*. For instance, they might “write some text” along the (removed) path or they might place nodes along this path. Note that for such decorations the path usage command for the main path have no influence on how the decoration looks like.



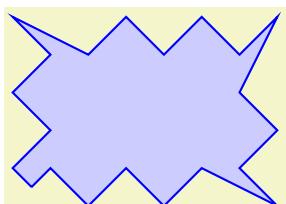
```
\usetikzlibrary {decorations.text}
\begin{tikzpicture}
\fill [decoration={text along path,
text=This is a text along a path. Note how the path is lost.}]
[fill=blue!20,draw=blue,thick] (0,0) -- (2,1) arc (90:-90:.5) -- cycle;
\end{tikzpicture}
```

Decorations are defined in different decoration libraries, see Section 51 for details. It is also possible to define your own decorations, see Section ??, but you need to use the PGF basic layer and a bit of theory is involved.

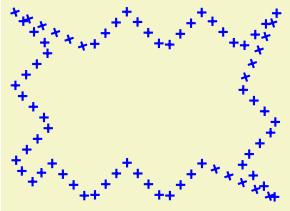
Decorations can be used to decorate already decorated paths. In the following three graphics, we start with a simple path, then decorate it once, and then decorate the decorated path once more.



```
\begin{tikzpicture}
\fill [fill=blue!20,draw=blue,thick]
(0,0) rectangle (3,2);
\end{tikzpicture}
```



```
\usetikzlibrary {decorations.pathmorphing}
\begin{tikzpicture}
\fill [fill=blue!20,draw=blue,thick]
decorate[decoration={zigzag,segment length=10mm,amplitude=2.5mm}]
{ (0,0) rectangle (3,2) };
\end{tikzpicture}
```



```
\usetikzlibrary { decorations.pathmorphing, decorations.shapes, }
\begin{tikzpicture}
\fill [blue!20, draw=blue, thick]
  decorate[decoration={crosses, segment length=2mm}] {
    decorate[decoration={zigzag, segment length=10mm, amplitude=2.5mm}] {
      (0,0) rectangle (3,2)
    }
  };
\end{tikzpicture}
```

One final word of warning: Decorations can be pretty slow to typeset and they can be inaccurate. The reason is that PGF has to do a *lot* of rather difficult computations in the background and \TeX is not very good at doing math. Decorations are fastest when applied to straight line segments, but even then they are much slower than other alternatives. For instance, the `ticks` decoration can be simulated by clever use of a dashing pattern and the dashing pattern will literally be thousands of times faster to typeset. However, for most decorations there are no real alternatives.

TikZ Library `decorations`

```
\usetikzlibrary{decorations} % \TeX{} and plain \TeX
\usetikzlibrary[decorations] % Con\TeX{}t
```

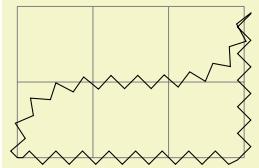
In order to use decorations, you first have to load a `decorations` library. This `decorations` library defines the basic options described in the following, but it does not define any new decorations. This is done by libraries like `decorations.text`. Since these more specialized libraries include the `decorations` library automatically, you usually do not have to bother about it.

24.2 Decorating a Subpath Using the `Decorate Path` Command

The most general way to decorate a (sub)path is the following path command.

```
\path ... decorate[<options>]{<subpath>} ...;
```

This path operation causes the `<subpath>` to be decorated using the current decoration. Depending on the decoration, this may or may not extend the current path.

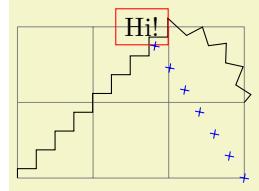


```
\usetikzlibrary {decorations.pathmorphing}
\begin{tikzpicture}
\draw [help lines] grid (3,2);
\draw decorate [decoration={name=zigzag}]
  { (0,0) .. controls (0,2) and (3,0) .. (3,2) |- (0,0) };
\end{tikzpicture}
```

The path can include straight lines, curves, rectangles, arcs, circles, ellipses, and even already decorated paths (that is, you can nest applications of the `decorate` path command, see below).

Due to the limits on the precision in \TeX , some inaccuracies in positioning when crossing input segment boundaries may occasionally be found.

You can use nodes normally inside the `<subpath>`.



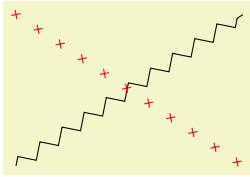
```
\usetikzlibrary {decorations.pathmorphing, decorations.shapes}
\begin{tikzpicture}
\draw [help lines] grid (3,2);
\draw decorate [decoration={name=zigzag}]
  { (0,0) -- (2,2) node (hi) [left, draw=red] {Hi!} arc(90:0:1)};
\draw [blue] decorate [decoration={crosses}] {(3,0) -- (hi)};
\end{tikzpicture}
```

The following key is used to select the decoration and also to select further “rendering options” for the decoration.

`/pgf/decoration=<decoration options>` (no default)
alias `/tikz/decoration`

This option is used to specify which decoration is used and how it will look like. Note that this key will *not* cause any decorations to be applied, immediately. It takes the `decorate` path command

or the `decorate` option to actually decorate a path. The `decoration` option is only used to specify which decoration should be used, in principle. You can also use this option at the beginning of a picture or a scope to specify the decoration to be used with each invocation of the `decorate` path command. Naturally, any local options of the `decorate` path command override these “global” options.



```
\usetikzlibrary { decorations.pathmorphing, decorations.shapes, }
\begin{tikzpicture}[decoration=zigzag]
\draw      decorate {(0,0) -- (3,2)};
\draw [red] decorate [decoration=crosses] {(0,2) -- (3,0)};
\end{tikzpicture}
```

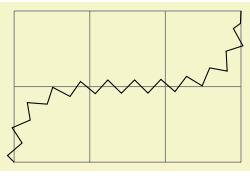
The `(decoration options)` are special options (which have the path prefix `/pgf/decoration/`) that determine the properties of the decoration. Which options are appropriate for a decoration strongly depend on the decoration, you will have to look up the appropriate options in the documentation of the decoration, see Section 51.

There is one option (available only in TikZ) that is special:

`/pgf/decoration/name=<name>` (no default, initially `none`)

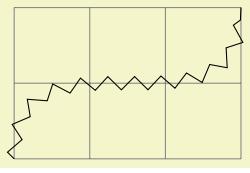
Use this key to set which decoration is to be used. The `<name>` can both be a decoration or a meta-decoration (you need to worry about the difference only if you wish to define your own decorations).

If you set `<name>` to `none`, no decorations are added.



```
\usetikzlibrary {decorations.pathmorphing}
\begin{tikzpicture}
\draw [help lines] grid (3,2);
\draw decorate [decoration={name=zigzag}]
  { (0,0) .. controls (0,2) and (3,0) .. (3,2) };
\end{tikzpicture}
```

Since this option is used so often, you can also leave out the `name=` part. Thus, the above example can be rewritten more succinctly:

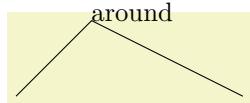


```
\usetikzlibrary {decorations.pathmorphing}
\begin{tikzpicture}
\draw [help lines] grid (3,2);
\draw decorate [decoration=zigzag]
  { (0,0) .. controls (0,2) and (3,0) .. (3,2) };
\end{tikzpicture}
```

In general, when `(decoration options)` are parsed, for each unknown key it is checked whether that key happens to be a (meta-)decoration and, if so, the `name` option is executed for this key.

Further options allow you to adjust the position of decorations relative to the to-be-decorated path. See Section 24.4 below for details.

Recall that some decorations actually completely remove the to-be-decorated path. In such cases, the construction of the main path is resumed after the `decorate` path command ends.



```
\usetikzlibrary {decorations.text}
\begin{tikzpicture}[decoration={text along path, text= around and around and around and around we go}]
\draw (0,0) -- (1,1) decorate { -- (2,1) } -- (3,0);
\end{tikzpicture}
```

It is permissible to nest `decorate` commands. In this case, the path resulting from the first decoration process is used as the to-be-decorated path for the second decoration process. This is especially useful for drawing fractals. The Koch snowflake decoration replaces a straight line like `_____` by `/___`.

Repeatedly applying this transformation to a triangle yields a fractal that looks a bit like a snowflake, hence the name.



```
\usetikzlibrary {decorations.fractals}
\begin{tikzpicture}[decoration=Koch snowflake, draw=blue, fill=blue!20, thick]
  \filldraw (0,0) -- ++(60:1) -- ++(-60:1) -- cycle ;
  \filldraw decorate{ (0,-1) -- ++(60:1) -- ++(-60:1) -- cycle };
  \filldraw decorate{ decorate{ (0,-2.5) -- ++(60:1) -- ++(-60:1) -- cycle }};
\end{tikzpicture}
```

24.3 Decorating a Complete Path

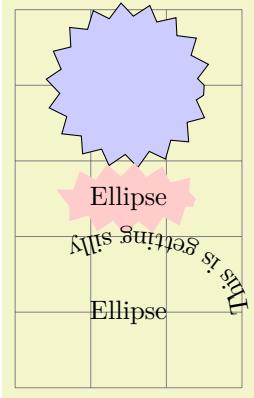
You may sometimes wish to decorate a path over whose construction you have no control. For instance, the path of the background of a node is created without having a chance to issue a `decorate` path command. In such cases you can use the following option, which allows you to decorate a path “after the fact”.

`/tikz/decorate=(boolean)` (default `true`)

When this key is set, the whole path is decorated after it has been finished. The decoration used for decorating the path is set via the `decoration` way, in exactly the same way as for the `decorate` path command. Indeed, the following two commands have the same effect:

1. `\path decorate[<options>] {<path>};`
2. `\path [decorate,<options>] <path>;`

The main use or the `decorate` option is the you can also use it with the nodes. It then causes the background path of the node to be decorated. Note that you can decorate a background path only once in this manner. That is, in contrast to the `decorate` path command you cannot apply this option twice (this would just set it to `true`, once more).



```
\usetikzlibrary { decorations.pathmorphing, decorations.text, shapes.geometric, }
\begin{tikzpicture}[decoration=zigzag]
  \draw [help lines] (0,0) grid (3,5);

  \draw [fill=blue!20,decorate] (1.5,4) circle (1cm);

  \node at (1.5,2.5) [fill=red!20,decorate,ellipse] {Ellipse};

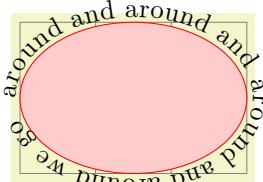
  \node at (1.5,1) [inner sep=6mm,fill=red!20,decorate,ellipse,decoration=
    {text along path, text={This is getting silly}}] {Ellipse};
\end{tikzpicture}
```

In the last example, the `text along path` decoration removes the path. In such cases it is useful to use a pre- or postaction to cause the decoration to be applied only before or after the main path has been used. Incidentally, this is another application of the `decorate` option that you cannot achieve with the `decorate` path command.



```
\usetikzlibrary { decorations.pathmorphing, decorations.text, shapes.geometric, }
\begin{tikzpicture}[decoration=zigzag]
  \node at (1.5,1) [inner sep=6mm,fill=red!20,ellipse,
    postaction={decorate,decoration=
    {text along path, text={This is getting silly}}}] {Ellipse};
\end{tikzpicture}
```

Here is more useful example, where a postaction is used to add the path after the main path has been drawn.



```
\usetikzlibrary {decorations.text}
\begin{tikzpicture}
\draw [help lines] grid (3,2);
\fill [draw=red,fill=red!20,
      postaction={decorate,decoration={raise=2pt,text along path,
      text=around and around and around and around we go}}]
(0,1) arc (180:-180:1.5cm and 1cm);
\end{tikzpicture}
```

24.4 Adjusting Decorations

24.4.1 Positioning Decorations Relative to the To-Be-Decorate Path

The following option, which are only available with TikZ, allow you to modify the positioning of decorations relative to the to-be-decorated path.

/pgf/decoration/raise=<dimension> (no default, initially 0pt)

The segments of the decoration are raised by *<dimension>* relative to the to-be-decorated path. More precisely, the segments of the path are offset by this much “to the left” of the path as we travel along the path. This raising is done after and in addition to any transformations set using the `transform` option (see below).

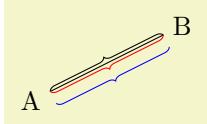
A negative *<dimension>* will offset the decoration “to the right” of the to-be-decorated path.



```
\usetikzlibrary {decorations.shapes}
\begin{tikzpicture}
\draw [help lines] (0,0) grid (3,2);
\draw (0,0) -- (1,1) arc (90:0:2 and 1);
\draw [decoration=crosses]
{ (0,0) -- (1,1) arc (90:0:2 and 1) };
\draw[red] [decoration={crosses,raise=5pt}]
{ (0,0) -- (1,1) arc (90:0:2 and 1) };
\end{tikzpicture}
```

/pgf/decoration/mirror=<boolean> (no default)

Causes the segments of the decoration to be mirrored along the to-be-decorated path. This is done after and in addition to any transformations set using the `transform` and/or `raise` options.



```
\usetikzlibrary {decorations.pathreplacing}
\begin{tikzpicture}
\node (a) {A};
\node (b) at (2,1) {B};
\draw (a) -- (b);
\draw[decoration=brace] (a) -- (b);
\draw[decoration={brace,mirror},red] (a) -- (b);
\draw[decoration={brace,mirror,raise=5pt},blue] (a) -- (b);
\end{tikzpicture}
```

/pgf/decoration/transform=<transformations> (no default)

This key allows you to specify general *<transformations>* to be applied to the segments of a decoration. These transformations are applied before and independently of `raise` and `mirror` transformations. The *<transformations>* should be normal TikZ transformations like `shift` or `rotate`.

In the following example the `shift` only transformation is used to make sure that the crosses are *not* sloped along the path.



```
\usetikzlibrary {decorations.shapes}
\begin{tikzpicture}
  \draw [help lines] (0,0) grid (3,2);
  \draw (0,0) -- (1,1) arc (90:0:2 and 1);
  \draw[red,very thick] decorate [decoration={%
    crosses,transform={shift only},shape size=1.5mm}]{%
    (0,0) -- (1,1) arc (90:0:2 and 1)};
\end{tikzpicture}
```

24.4.2 Starting and Ending Decorations Early or Late

You sometimes may wish to “end” a decoration a bit early on the path. For instance, you might wish a `snake` decoration to stop 5mm before the end of the path and to continue in a straight line. There are different ways of achieving this effect, but the easiest may be the `pre` and `post` options, which only have an effect in TikZ. Note, however, that they can only be used with decorations, not with meta-decorations.

`/pgf/decoration/pre=<decoration>`

(no default, initially `lineto`)

This key sets a decoration that should be used before the main decoration starts. The `<decoration>` will be used for a length of `pre length`, which `0pt` by default. Thus, for the `pre` option to have any effect, you also need to set the `pre length` option.



```
\usetikzlibrary {decorations.pathmorphing}
\tikz [decoration={zigzag,pre=lineto,pre length=1cm}]
  \draw [decorate] (0,0) -- (2,1) arc (90:0:1);
```

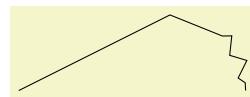


```
\usetikzlibrary {decorations.pathmorphing}
\tikz [decoration={zigzag,pre=moveto,pre length=1cm}]
  \draw [decorate] (0,0) -- (2,1) arc (90:0:1);
```

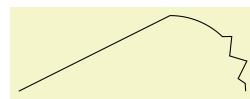


```
\usetikzlibrary {decorations.pathmorphing, decorations.shapes, }
\tikz [decoration={zigzag,pre=crosses,pre length=1cm}]
  \draw [decorate] (0,0) -- (2,1) arc (90:0:1);
```

Note that the default `pre` option is `lineto`, not `curveto`. This means that the default `pre` decoration will not follow curves (for efficiency reasons). Change the `pre` key to `curveto` if you have a curved path.



```
\usetikzlibrary {decorations.pathmorphing}
\tikz [decoration={zigzag,pre length=3cm}]
  \draw [decorate] (0,0) -- (2,1) arc (90:0:1);
```



```
\usetikzlibrary {decorations.pathmorphing}
\tikz [decoration={zigzag,pre=curveto,pre length=3cm}]
  \draw [decorate] (0,0) -- (2,1) arc (90:0:1);
```

`/pgf/decoration/pre length=<dimension>`

(no default, initially `0pt`)

This key sets the distance along which the pre-decoration should be used. If you do not need/wish a pre-decoration, set this key to `0pt` (exactly this string, not just to something that evaluates to the same things such as `0cm`).

`/pgf/decoration/post=<decoration>`

(no default, initially `lineto`)

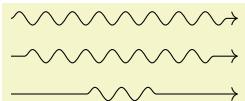
Works like `pre`, only for the end of the decoration.

`/pgf/decorations/post length=<dimension>`

(no default, initially 0pt)

Works like `pre length`, only for the end of the decoration.

Here is a typical example that shows how these keys can be used:



```
\usetikzlibrary {decorations.pathmorphing}
\begin{tikzpicture}
[decoration=snake,
 line around/.style={decoration={pre length=#1,post length=#1}}]

\draw[->,decorate] (0,0) -- ++(3,0);
\draw[->,decorate,line around=5pt] (0,-5mm) -- ++(3,0);
\draw[->,decorate,line around=1cm] (0,-1cm) -- ++(3,0);
\end{tikzpicture}
```

25 Transformations

PGF has a powerful transformation mechanism that is similar to the transformation capabilities of METAFONT. The present section explains how you can access it in TikZ.

25.1 The Different Coordinate Systems

It is a long process from a coordinate like, say, (1, 2) or (1cm, 5pt), to the position a point is finally placed on the display or paper. In order to find out where the point should go, it is constantly “transformed”, which means that it is mostly shifted around and possibly rotated, slanted, scaled, and otherwise mutilated.

In detail, (at least) the following transformations are applied to a coordinate like (1, 2) before a point on the screen is chosen:

1. PGF interprets a coordinate like (1, 2) in its *xy*-coordinate system as “add the current *x*-vector once and the current *y*-vector twice to obtain the new point”.
2. PGF applies its coordinate transformation matrix to the resulting coordinate. This yields the final position of the point inside the picture.
3. The backend driver (like dvips or pdftex) adds transformation commands such that the coordinate is shifted to the correct position in T_EX’s page coordinate system.
4. PDF (or PostScript) apply the canvas transformation matrix to the point, which can once more change the position on the page.
5. The viewer application or the printer applies the device transformation matrix to transform the coordinate to its final pixel coordinate on the screen or paper.

In reality, the process is even more involved, but the above should give the idea: A point is constantly transformed by changes of the coordinate system.

In TikZ, you only have access to the first two coordinate systems: The *xy*-coordinate system and the coordinate transformation matrix (these will be explained later). PGF also allows you to change the canvas transformation matrix, but you have to use commands of the core layer directly to do so and you “better know what you are doing” when you do this. The moment you start modifying the canvas matrix, PGF immediately loses track of all coordinates and shapes, anchors, and bounding box computations will no longer work.

25.2 The XY- and XYZ-Coordinate Systems

The first and easiest coordinate systems are PGF’s *xy*- and *xyz*-coordinate systems. The idea is very simple: Whenever you specify a coordinate like (2, 3) this means $2v_x + 3v_y$, where v_x is the current *x*-vector and v_y is the current *y*-vector. Similarly, the coordinate (1, 2, 3) means $v_x + 2v_y + 3v_z$.

Unlike other packages, PGF does not insist that v_x actually has a *y*-component of 0, that is, that it is a horizontal vector. Instead, the *x*-vector can point anywhere you want. Naturally, *normally* you will want the *x*-vector to point horizontally.

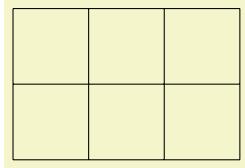
One undesirable effect of this flexibility is that it is not possible to provide mixed coordinates as in (1, 2pt). Life is hard.

To change the *x*-, *y*-, and *z*-vectors, you can use the following options:

`/tikz/x=<value>` (no default, initially 1cm)

If *<value>* is a dimension, the *x*-vector of PGF’s *xyz*-coordinate system is set up to point *<value>* to the right, that is, to (*<value>*, 0pt).

```
\begin{tikzpicture}
  \draw (0,0) -- +(1,0);
  \draw[x=2cm,color=red] (0,0.1) -- +(1,0);
\end{tikzpicture}
```



```
\tikz \draw[x=1.5cm] (0,0) grid (2,2);
```

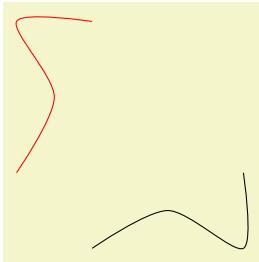
The last example shows that the size of steppings in grids, just like all other dimensions, are not affected by the *x*-vector. After all, the *x*-vector is only used to determine the coordinate of the upper right corner of the grid.

If $\langle value \rangle$ is a coordinate, the *x*-vector of PGF's *xyz*-coordinate system to the specified coordinate. If $\langle value \rangle$ contains a comma, it must be put in braces.



```
\begin{tikzpicture}
  \draw (0,0) -- (1,0);
  \draw[x={(2cm,0.5cm)},color=red] (0,0) -- (1,0);
\end{tikzpicture}
```

You can use this, for example, to exchange the meaning of the *x*- and *y*-coordinate.



```
\begin{tikzpicture}[smooth]
  \draw plot coordinates{(1,0) (2,0.5) (3,0) (3,1)};
  \draw[x={(.0cm,1cm)},y={(1cm,0cm)},color=red]
    plot coordinates{(1,0) (2,0.5) (3,0) (3,1)};
\end{tikzpicture}
```

/tikz/y= $\langle value \rangle$

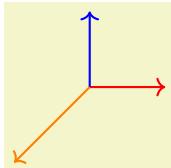
(no default, initially 1cm)

Works like the *x=* option, only if $\langle value \rangle$ is a dimension, the resulting vector points to $(0, \langle value \rangle)$.

/tikz/z= $\langle value \rangle$

(no default, initially -3.85mm)

Works like the *y=* option, but now a dimension is the point $(\langle value \rangle, \langle value \rangle)$.

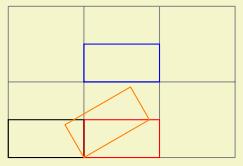


```
\begin{tikzpicture}[z=-1cm,->,thick]
  \draw[color=red] (0,0,0) -- (1,0,0);
  \draw[color=blue] (0,0,0) -- (0,1,0);
  \draw[color=orange] (0,0,0) -- (0,0,1);
\end{tikzpicture}
```

25.3 Coordinate Transformations

PGF and TikZ allow you to specify *coordinate transformations*. Whenever you specify a coordinate as in $(1,0)$ or $(1\text{cm},1\text{pt})$ or $(30:2\text{cm})$, this coordinate is first “reduced” to a position of the form “*x* points to the right and *y* points upwards”. For example, $(1\text{in},5\text{pt})$ is reduced to “ $72\frac{72}{100}$ points to the right and 5 points upwards” and $(90:100\text{pt})$ means “Opt to the right and 100 points upwards”.

The next step is to apply the current *coordinate transformation matrix* to the coordinate. For example, the coordinate transformation matrix might currently be set such that it adds a certain constant to the *x* value. Also, it might be set up such that it, say, exchanges the *x* and *y* value. In general, any “standard” transformation like translation, rotation, slanting, or scaling or any combination thereof is possible. (Internally, PGF keeps track of a coordinate transformation matrix very much like the concatenation matrix used by PDF or PostScript.)

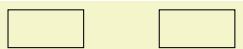


```
\begin{tikzpicture}
\draw[help lines] (0,0) grid (3,2);
\draw (0,0) rectangle (1,0.5);
\begin{scope}[xshift=1cm]
\draw [red] (0,0) rectangle (1,0.5);
\draw[yshift=1cm] [blue] (0,0) rectangle (1,0.5);
\draw[rotate=30] [orange] (0,0) rectangle (1,0.5);
\end{scope}
\end{tikzpicture}
```

The most important aspect of the coordinate transformation matrix is *that it applies to coordinates only!* In particular, the coordinate transformation has no effect on things like the line width or the dash pattern or the shading angle. In certain cases, it is not immediately clear whether the coordinate transformation matrix *should* apply to a certain dimension. For example, should the coordinate transformation matrix apply to grids? (It does.) And what about the size of arced corners? (It does not.) The general rule is: “If there is no ‘coordinate’ involved, even ‘indirectly’, the matrix is not applied.”. However, sometimes, you simply have to try or look it up in the documentation whether the matrix will be applied.

Setting the matrix cannot be done directly. Rather, all you can do is to “add” another transformation to the current matrix. However, all transformations are local to the current TeX-group. All transformations are added using graphic options, which are described below.

Transformations apply immediately when they are encountered “in the middle of a path” and they apply only to the coordinates on the path following the transformation option.



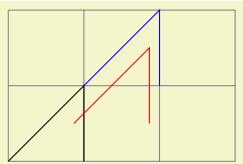
```
\tikz \draw (0,0) rectangle (1,0.5) [xshift=2cm] (0,0) rectangle (1,0.5);
```

A final word of warning: You should refrain from using “aggressive” transformations like a scaling of a factor of 10 000. The reason is that all transformations are done using TeX, which has a fairly low accuracy. Furthermore, in certain situations it is necessary that TikZ *inverts* the current transformation matrix and this will fail if the transformation matrix is badly conditioned or even singular (if you do not know what singular matrices are, you are blessed).

/tikz/shift={⟨coordinate⟩}

(no default)

Adds the ⟨coordinate⟩ to all coordinates.



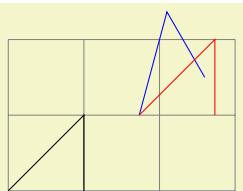
```
\begin{tikzpicture}
\draw[help lines] (0,0) grid (3,2);
\draw (0,0) -- (1,1) -- (1,0);
\draw[shift={(1,1)},blue] (0,0) -- (1,1) -- (1,0);
\draw[shift={(30:1cm)},red] (0,0) -- (1,1) -- (1,0);
\end{tikzpicture}
```

/tikz/shift only

(no value)

This option does not take any parameter. Its effect is to cancel all current transformations except for the shifting. This means that the origin will remain where it is, but any rotation around the origin or scaling relative to the origin or skewing will no longer have an effect.

This option is useful in situations where a complicated transformation is used to “get to a position”, but you then wish to draw something “normal” at this position.

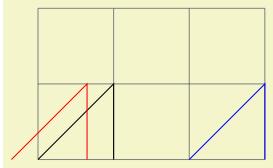


```
\begin{tikzpicture}
\draw[help lines] (0,0) grid (3,2);
\draw (0,0) -- (1,1) -- (1,0);
\draw[rotate=30,xshift=2cm,blue] (0,0) -- (1,1) -- (1,0);
\draw[rotate=30,xshift=2cm,shift only,red] (0,0) -- (1,1) -- (1,0);
\end{tikzpicture}
```

/tikz/xshift={⟨dimension⟩}

(no default)

Adds ⟨dimension⟩ to the *x* value of all coordinates.



```
\begin{tikzpicture}
\draw[help lines] (0,0) grid (3,2);
\draw (0,0) -- (1,1) -- (1,0);
\draw[xshift=2cm,blue] (0,0) -- (1,1) -- (1,0);
\draw[xshift=-10pt,red] (0,0) -- (1,1) -- (1,0);
\end{tikzpicture}
```

/tikz/yshift=⟨dimension⟩

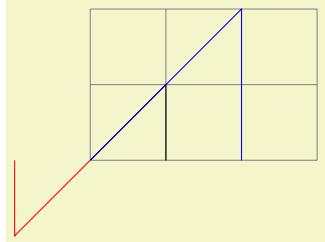
(no default)

Adds ⟨dimension⟩ to the *y* value of all coordinates.

/tikz/scale=⟨factor⟩

(no default)

Multiplies all coordinates by the given ⟨factor⟩. The ⟨factor⟩ should not be excessively large in absolute terms or very close to zero.

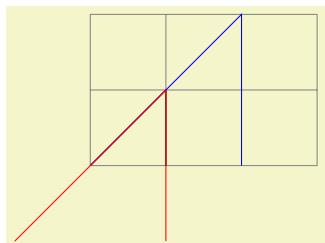


```
\begin{tikzpicture}
\draw[help lines] (0,0) grid (3,2);
\draw (0,0) -- (1,1) -- (1,0);
\draw[scale=2,blue] (0,0) -- (1,1) -- (1,0);
\draw[scale=-1,red] (0,0) -- (1,1) -- (1,0);
\end{tikzpicture}
```

/tikz/scale around={⟨factor⟩}:{⟨coordinate⟩}

(no default)

Scales the coordinate system by ⟨factor⟩, with the “origin of scaling” centered on ⟨coordinate⟩ rather than the origin.

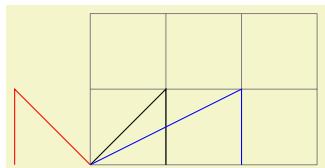


```
\begin{tikzpicture}
\draw[help lines] (0,0) grid (3,2);
\draw (0,0) -- (1,1) -- (1,0);
\draw[scale=2,blue] (0,0) -- (1,1) -- (1,0);
\draw[scale around={2:(1,1)},red] (0,0) -- (1,1) -- (1,0);
\end{tikzpicture}
```

/tikz/xscale=⟨factor⟩

(no default)

Multiplies only the *x*-value of all coordinates by the given ⟨factor⟩.



```
\begin{tikzpicture}
\draw[help lines] (0,0) grid (3,2);
\draw (0,0) -- (1,1) -- (1,0);
\draw[xscale=2,blue] (0,0) -- (1,1) -- (1,0);
\draw[xscale=-1,red] (0,0) -- (1,1) -- (1,0);
\end{tikzpicture}
```

/tikz/yscale=⟨factor⟩

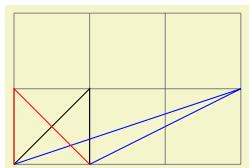
(no default)

Multiplies only the *y*-value of all coordinates by the given ⟨factor⟩.

/tikz/xslant=⟨factor⟩

(no default)

Slants the coordinate horizontally by the given ⟨factor⟩:

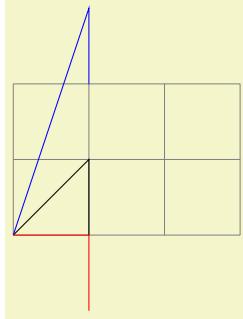


```
\begin{tikzpicture}
\draw[help lines] (0,0) grid (3,2);
\draw (0,0) -- (1,1) -- (1,0);
\draw[xslant=2,blue] (0,0) -- (1,1) -- (1,0);
\draw[xslant=-1,red] (0,0) -- (1,1) -- (1,0);
\end{tikzpicture}
```

`/tikz/yshift=⟨factor⟩`

(no default)

Slants the coordinate vertically by the given `⟨factor⟩`:

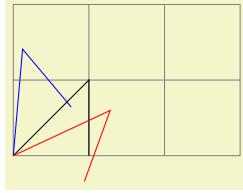


```
\begin{tikzpicture}
  \draw[help lines] (0,0) grid (3,2);
  \draw (0,0) -- (1,1) -- (1,0);
  \draw[yshift=2,blue] (0,0) -- (1,1) -- (1,0);
  \draw[yshift=-1,red] (0,0) -- (1,1) -- (1,0);
\end{tikzpicture}
```

`/tikz/rotate=⟨degree⟩`

(no default)

Rotates the coordinate system by `⟨degree⟩`:

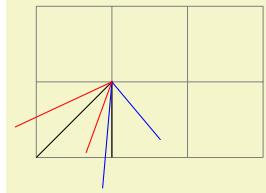


```
\begin{tikzpicture}
  \draw[help lines] (0,0) grid (3,2);
  \draw (0,0) -- (1,1) -- (1,0);
  \draw[rotate=40,blue] (0,0) -- (1,1) -- (1,0);
  \draw[rotate=-20,red] (0,0) -- (1,1) -- (1,0);
\end{tikzpicture}
```

`/tikz/rotate around={⟨degree⟩}:{⟨coordinate⟩}`

(no default)

Rotates the coordinate system by `⟨degree⟩` around the point `⟨coordinate⟩`.

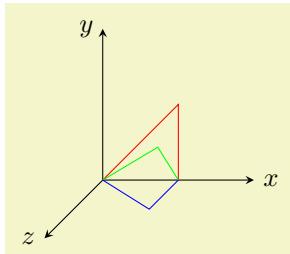


```
\begin{tikzpicture}
  \draw[help lines] (0,0) grid (3,2);
  \draw (0,0) -- (1,1) -- (1,0);
  \draw[rotate around={40:(1,1)},blue] (0,0) -- (1,1) -- (1,0);
  \draw[rotate around={-20:(1,1)},red] (0,0) -- (1,1) -- (1,0);
\end{tikzpicture}
```

`/tikz/rotate around x=⟨angle⟩`

(no default)

This key sets the x , y and z vectors of the PGF xyz -coordinate system so that they are rotated by `⟨angle⟩` around the axis corresponding to the x -vector. The rotation is applied so that when looking towards the origin along this axis, positive angles result in an anticlockwise rotation.



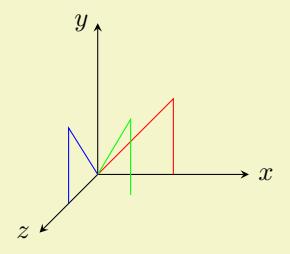
```
\begin{tikzpicture} [>=stealth]
  \draw [->] (0,0,0) -- (2,0,0) node [at end, right] {$x$};
  \draw [->] (0,0,0) -- (0,2,0) node [at end, left] {$y$};
  \draw [->] (0,0,0) -- (0,0,2) node [at end, left] {$z$};

  \draw [red, rotate around x=0] (0,0,0) -- (1,1,0) -- (1,0,0);
  \draw [green, rotate around x=45] (0,0,0) -- (1,1,0) -- (1,0,0);
  \draw [blue, rotate around x=90] (0,0,0) -- (1,1,0) -- (1,0,0);
\end{tikzpicture}
```

`/tikz/rotate around y=⟨angle⟩`

(no default)

This key sets the x , y and z vectors of the PGF xyz -coordinate system so that they are rotated by `⟨angle⟩` around the axis corresponding to the y -vector. The rotation is applied so that when looking towards the origin along this axis, positive angles result in an anticlockwise rotation.



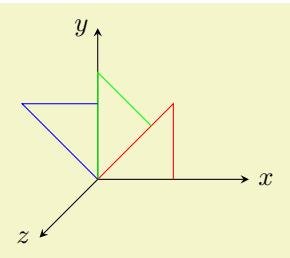
```
\begin{tikzpicture} [>=stealth]
  \draw [->] (0,0,0) -- (2,0,0) node [at end, right] {$x$};
  \draw [->] (0,0,0) -- (0,2,0) node [at end, left] {$y$};
  \draw [->] (0,0,0) -- (0,0,2) node [at end, left] {$z$};

  \draw [red, rotate around y=0] (0,0,0) -- (1,1,0) -- (1,0,0);
  \draw [green, rotate around y=-45] (0,0,0) -- (1,1,0) -- (1,0,0);
  \draw [blue, rotate around y=-90] (0,0,0) -- (1,1,0) -- (1,0,0);
\end{tikzpicture}
```

/tikz/rotate around z=<angle>

(no default)

This key sets the x , y and z vectors of the PGF xyz -coordinate system so that they are rotated by $\langle angle \rangle$ around the axis corresponding to the z -vector. The rotation is applied so that when looking towards the origin along this axis, positive angles result in an anticlockwise rotation.



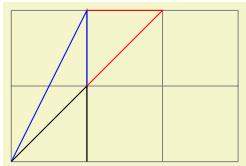
```
\begin{tikzpicture} [>=stealth]
  \draw [->] (0,0,0) -- (2,0,0) node [at end, right] {$x$};
  \draw [->] (0,0,0) -- (0,2,0) node [at end, left] {$y$};
  \draw [->] (0,0,0) -- (0,0,2) node [at end, left] {$z$};

  \draw [red, rotate around z=0] (0,0,0) -- (1,1) -- (1,0);
  \draw [green, rotate around z=45] (0,0,0) -- (1,1) -- (1,0);
  \draw [blue, rotate around z=90] (0,0,0) -- (1,1) -- (1,0);
\end{tikzpicture}
```

/tikz/cm={<a>,,<c>,<d>,<coordinate>}

(no default)

applies the following transformation to all coordinates: Let (x, y) be the coordinate to be transformed and let $\langle coordinate \rangle$ specify the point (t_x, t_y) . Then the new coordinate is given by $(\begin{smallmatrix} a & c \\ b & d \end{smallmatrix})(\begin{smallmatrix} x \\ y \end{smallmatrix}) + (\begin{smallmatrix} t_x \\ t_y \end{smallmatrix})$. Usually, you do not use this option directly.



```
\begin{tikzpicture}
  \draw[help lines] (0,0) grid (3,2);
  \draw (0,0) -- (1,1) -- (1,0);
  \draw[cm={1,1,0,1,(0,0)},blue] (0,0) -- (1,1) -- (1,0);
  \draw[cm={0,1,1,0,(1cm,1cm)},red] (0,0) -- (1,1) -- (1,0);
\end{tikzpicture}
```

/tikz/reset cm

(no value)

Completely resets the coordinate transformation matrix to the identity matrix. This will destroy not only the transformations applied in the current scope, but also all transformations inherited from surrounding scopes. Do not use this option, unless you really, really know what you are doing.

25.4 Canvas Transformations

A *canvas transformation*, see Section ?? for details, is best thought of as a transformation in which the drawing canvas is stretched or rotated. Imaging writing something on a balloon (the canvas) and then blowing air into the balloon: Not only does the text become larger, the thin lines also become larger. In particular, if you scale the canvas by a factor of two, all lines are twice as thick.

Canvas transformations should be used with great care. In most circumstances you do *not* want line widths to change in a picture as this creates visual inconsistency.

Just as important, when you use canvas transformations PGF loses track of positions of nodes and of picture sizes since it does not take the effect of canvas transformations into account when it computes coordinates of nodes (do not, however, rely on this; it may change in the future).

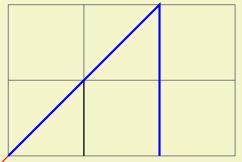
Finally, note that a canvas transformation always applies to a path as a whole, it is not possible (as for coordinate transformations) to use different transformations in different parts of a path.

In short, you should not use canvas transformations unless you really know what you are doing.

`/tikz/transform canvas=⟨options⟩`

(no default)

The `⟨options⟩` should contain coordinate transformations options like `scale` or `xshift`. Multiple options can be given, their effects accumulate in the usual manner. The effect of these `⟨options⟩` (immediately) changes the current canvas transformation matrix. The coordinate transformation matrix is not changed. Tracking of the picture size is (locally) switched off and the node coordinate will no longer be correct.



```
\begin{tikzpicture}
  \draw[help lines] (0,0) grid (3,2);
  \draw (0,0) -- (1,1) -- (1,0);
  \draw[transform canvas={scale=2},blue] (0,0) -- (1,1) -- (1,0);
  \draw[transform canvas={rotate=180},red] (0,0) -- (1,1) -- (1,0);
\end{tikzpicture}
```

26 Animations

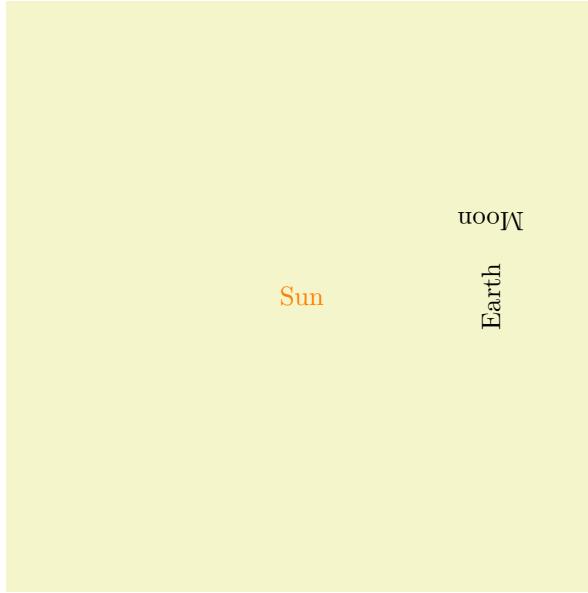
TikZ Library `animations`

```
\usetikzlibrary{animations} % LATEX and plain TEX
\usetikzlibrary[animations] % ConTEXt
```

This library must be loaded in order to use animations with TikZ.

26.1 Introduction

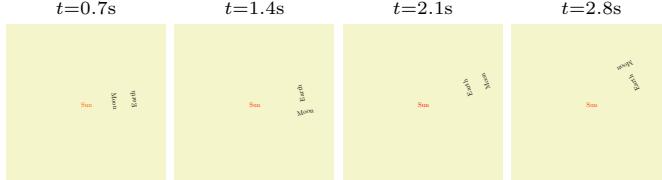
An *animation* changes the appearance of some part of a graphic over time. The archetypical animation is, of course, a *movement* of some part of a picture, but a change of, say, the opacity of a path is also an animation. TikZ allows you to specify such animations using special keys and notations.



```
\usetikzlibrary {animations}
\begin{tikzpicture}[
  animate/orbit/.style 2 args = {
    myself:shift = {
      along = {
        (0,0) circle [radius=#1]
      } sloped in #2s/10,
      repeats }} ]
\node :color = {0s = "orange",
  2s = "red",
  4s = "orange",
  repeats}
{Sun};

\begin{scope}[animate={orbit={2.5cm}{365}}]
  \node {Earth};
  \node [animate={orbit={1cm}{28}}] {Moon};
\end{scope}

\useasboundingbox (-3.8,-3.8) (3.8,3.8);
\end{tikzpicture}
```



Adding an animation to a TikZ picture is done as follows:

1. *Before* or *in the options* of the to-be-animated object you specify the object together with an *attribute* that you wish to animate. Attributes are things like the fill color or the line width or the position of the object.
2. You specify *when* this attribute should have *which* values using a so-called *timeline*. This is just a curve that specifies for each point in time which value the attribute should have.
3. You can additionally use further options to configure the animation, for instance you can specify that the animation should repeat or that it should only start when a certain object is clicked.

As a simple example, let us move a circle within thirty seconds by three centimeters to the left:



```
\usetikzlibrary {animations}
\tikz \draw :xshift = {0s = "0cm", 30s = "-3cm", repeats} (0,0) circle (5mm);
```

As can be seen, a special syntax is used in several places: Entries with a colon such as `:xshift` specify an attribute, values are specified in quotation marks. This syntax will be explained in more detail later on.

26.1.1 Animations Change Attributes

Before we plunge into the details of how animations are specified, it is important to understand what TikZ actually does when creating an animation: It does *not* (as all other animation packages do) precompute a sequence of pictures that are later somehow displayed in rapid succession. Neither does it insert an external video into the document. Rather, a TikZ animation is just an “annotation” in the output that a certain attribute of a certain object should change over time in some specific way when the object is displayed. It is the job of the document viewer application to actually compute and display the animation. The big advantage of this approach is that animations neither increase the output file sizes noticeably nor do they really slow down TeX: The hard and complicated calculations are done by the viewer application. The disadvantage is, of course, that a document viewer application must understand the annotations and actually compute and display the animations. The SVG format is a format for which this is possible, the popular PDF format is not. For the SVG format, there are actually different possible ways of “formulating” the animations (using SMIL or CSS or JavaScript) and they have different advantages and disadvantages.

To make a long story short: TikZ animations currently work only with SVG output (and use the SMIL “flavor” of describing animations). In future, it may well happen that other “flavor” of describing animations will be added, but it is very unlikely that PDF will ever support animations in a useful way.

It is, however, possible to create “snapshots” of an animation and insert these into PDF files (or any other kind of file including SVG files), see Section 26.6 for details. Snapshots are also useful for creating “printed versions” of animations and all of the small sequences of pictures in the manual that are used for showing what an animation key does have been creating using snapshots.

26.1.2 Limitations of the Animation System

There are a certain limitations of the animation system that you should keep in mind when considering how and when to use it:

1. As pointed out earlier, animations require a specific output format (currently only SVG is supported).
2. It is extremely difficult to animate “lines between moving nodes” correctly. Consider code like `\draw(a)--(b);` where `a` and `b` are nodes. Now, when you animate the position of `(a)`, the line connecting `(a)` and `(b)` will, unfortunately, not “move along” automatically (but it is easy to move the whole group of `(a)`, `(b)`, and the connecting line as whole). You must “cheat” and introduce some “virtual” nodes, which leads to rather complex and bloated code.
3. Animations are taken into consideration for bounding box computations, but only for shifts, not for rotations, scaling, or skewing and also possibly not when multiple shifts are active at the same time for the same object.

26.1.3 Concepts: (Graphic) Objects

During an animation an attribute of a certain “object” changes over time. The term “object” is deliberately a bit vague since there are numerous different “things” whose attributes can change. In detail, the following objects have attributes that can be animated:

1. Nodes, which are created by the `\node` command (and, also, internally by commands such as `\graph`). For nodes, different parts of the node can be animated separately; for instance, you can animate the color of the background path, but also the color of the text, and also the color of the foreground path (though most nodes do not have a foreground path) and also the color of different text parts (though only few nodes have multiple text parts).
2. Graphic scopes, which are created by numerous command, including the `{scope}` environment, the `\scopes` command, but also `\tikz` itself creates a graphic scope and so does each node and even each path.
3. View boxes, which can only be created using the `views` library.
4. Paths, which you create using the `\path` command or commands like `\draw` that call `\path` internally. However, the (usually background) path of a node can also be animated. Note that “animating the path” really means that the path itself should change over time; in essence, you can “warp” a path over time.

In all of these cases, you must either specify the animation inside the object's options using `animate` or use the `name` key to name the object and, then, refer to it in an `animate`. For nodes you can, of course, use the `(<node name>)` syntax to name the node. Recall that you must *always* specify the animation *before* the object is created; it is not possible to animate an already created object.

There is a special syntax for choosing the object of an animation, see Section 26.3.1, but you can also use the `object` key to choose them directly, see Section 26.2.3.

26.1.4 Concepts: Attributes

In addition to the to-be-animated object, you must also choose an *attribute* that you wish to animate. Attributes are things like the color of an object, the position, but also things like the line width. The syntax for choosing attributes and the list of attributes will be explained in detail later on.

Most attributes correspond directly to attributes that are directly supported by the backend driver (SVG), but this is not always the case. For instance, for a node, TikZ differentiates between the fill color, the draw (stroke) color, and the text color, while SVG treats the text color as a special case of the fill color. TikZ will do some internal mappings to ensure that you can animate the “TikZ attributes” even when they are not directly supported.

The same syntax that is used for specifying object is also used to specify attributes, see Section 26.3.1, but you could also set them directly using the `attribute` key see Section 26.2.4.

26.1.5 Concepts: Timelines

Once an object and an attribute have been chosen, a *timeline* needs to be established. This is, essentially, a curve that specifies for each “moment in time” which value the attribute should have.

A timeline has a *start* and an *end*, but the start need not be the “moment zero” (we will come to that) and may even be negative, while the end may be at infinity. You specify the timeline by specifying for certain points in time what the value is at that moment; for all other moments the value is then interpolated. For instance, if you specify that the attribute `:xshift` (the “horizontal position” of the object) is 0 mm at time 5 s and 10 mm at time 10 s, then at 7.5 s it will be 5 mm and at 9 s it will be 8 mm (assuming a linear interpolation). The resulting optical effect will be that the object *smoothly moves* by one centimeter to the right over a period of five seconds, starting five seconds after “moment zero”.

Now, what is the “moment zero”, the “beginning of an animation”? If nothing else is specified, an animation starts immediately when the graphic is shown and this is the moment zero relative to which the timeline is measured. However, it is also possible to change this. In particular, you can specify that the moment zero is when a particular *event* occurs such as the user clicking on another object or another animation ending or starting.

The interpolation of values is not always a straightforward affair. Firstly, for certain kinds of values is not clear how an interpolation should be computed. How does one interpolate between two paths? Between the colors red and green? Between the values `"true"` and `"false"`? In these cases, one must define carefully what the interpolation should be. Secondly, you may wish to use a non-linear interpolation, which is useful for “easing” motions: The visual effect of the movement specified above is that the object sits still from moment 0 for five seconds, then there is an “infinite acceleration” causing the object to suddenly move at the speed of 2 mm per second, then there is no acceleration at all for five seconds, causing the object to move for one centimeter, followed by an “infinite negative acceleration” that makes the object come to a full stop. As a viewer you experience these infinite accelerations as “unrealistic”, spoiling the effect of watching a (virtual) physical process. Non-linear interpolations allow you to avoid this effect.

Just as for specifying objects and attributes, there is also a special syntax for specifying times and values.

26.2 Creating an Animation

26.2.1 The Animate Key

In order to animate a picture, you create timelines for all objects and attributes that change during the animation. The key `animate` is used for creating these timelines.

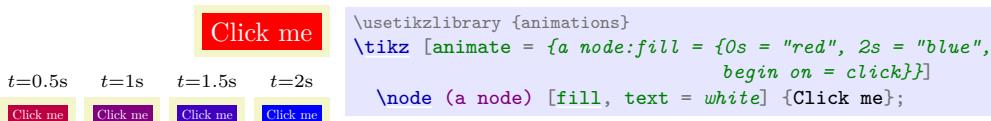
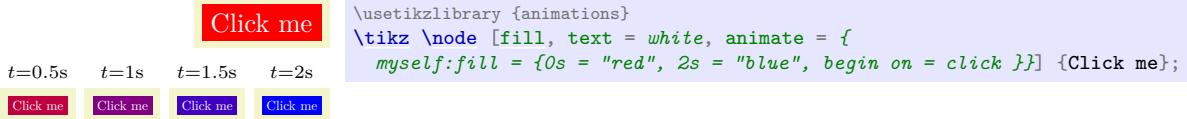
`/tikz/animate=(animation specification)`

(no default)

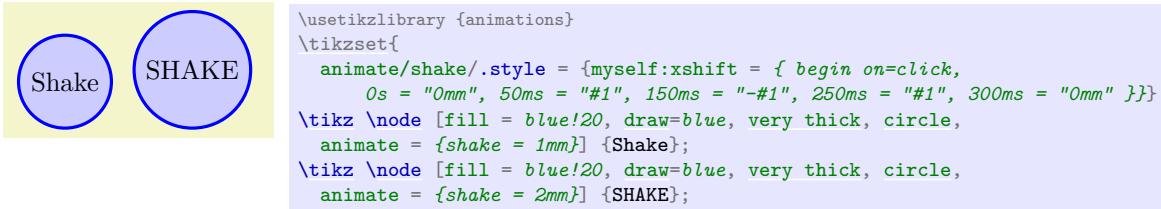
You must place all specifications of animations inside uses of `animate`. You can, and usually should, place the specification of all timelines of a single picture inside a single use of this key since it will reset the time and the fork time (explained in Section 26.2.6). You can, however, use this key several times,

in principle. Note that if you animate the same attribute of the same object in two different uses of `animate`, two separate timelines will result (and complicated rules are used to determine which one “wins” in case they specify conflicting values for the attribute at different times).

The key can be used at all places where a TikZ key is used; typically you will use it with a `{scope}` environment, inside the options of a node, or directly with the `\tikz` command:



The details of what, exactly, happens in the `<animation specification>` will be described in the rest of this section. However, basically, an `<animation specification>` is just a sequence of normal TikZ key–value pairs that get executed with the path prefix `/tikz/animate` and with some special syntax handlers installed. In particular, you can define styles for this key path and use them. For instance, we can define a `shake` animation like this:



Note that, as stressed earlier, you can only use the `animate` key to specify animations for objects that do not yet exist. The node and object names mentioned in a specification always refer to “upcoming” objects; already existing objects of the same name are not influenced.

You can use the `name` key inside `animate` to “name” the animation. Once named, you can later reference the animation in other animations; for instance, you can say that another animation should start when the present animation has ended.

26.2.2 Timeline Entries

The “job” of the options passed to the `animate` key is to specify the timelines of the animation of (a part of) a picture. For each object and each attribute there may or may not be a timeline and, if present, the timeline consist of sequences of pairs of times and values. Thus, the most basic entity of an animation specification is a tuple consisting of five parts, which are selected by five different keys:

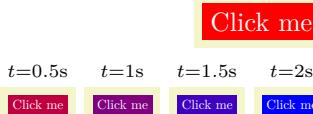
- `object` for selecting the object,
- `attribute` for selecting the attribute,
- `id` for selecting the timeline id (explained in Section 26.2.5),
- `time` for selecting a time, and
- `value` for selecting a value.

When all of these parts have been set up (using the above keys, which will be explained in more detail in a moment), you can use the following key to create an entry:

`/tikz/animate/entry` (no value)

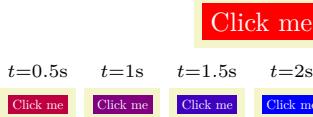
Each time this key is used in the options of `animate`, TikZ checks whether the five keys `object`, `attribute`, `id`, `time`, and `value` are set. If one of them is not set, nothing happens. (The `id` key is set to the value `default` by default, all other keys must be set explicitly.)

If all of these keys are set, a *time–value* pair is created and added to the timeline of attribute of the object. Additionally, all options starting with `/tikz/animate/options/`, which also influence the timeline like `begin on`, are also added to the timeline of the object–attribute pair.



```
\usetikzlibrary {animations}
\tikz [animate = {
    object = node, attribute = fill, time = 0s, value = red, entry,
    object = node, attribute = fill, time = 2s, value = blue, entry,
    object = node, attribute = fill, begin on = click, entry}]
\node (node) [fill, text=white] { Click me };
```

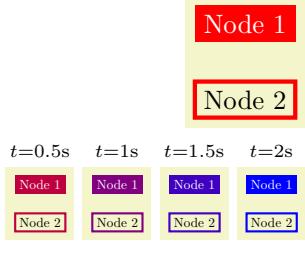
In the above example, it would not have been necessary to specify the object and the attribute in each line, they retain their values unless they are overwritten. Thus, we could also have written:



```
\usetikzlibrary {animations}
\tikz [animate = {
    object = node, attribute = fill, time = 0s, value = red, entry,
    time = 2s, value = blue, entry,
    begin on = click, entry}]
\node (node) [fill, text=white] { Click me };
```

Note, however, that in both examples we actually add the time–value pair (2s, blue) twice since the `time` and `value` keys also retain their settings and, thus, for the third `entry` they have the same values as before and a new pair is added. While this superfluous pair is not a problem in the example (it has no visual effect), we will see later on how such pairs can be avoided by using the `scope` key.

A sequence of calls of `entry` can freely switch between objects and attributes (that is, between timelines), but the times for any given timeline must be given in non-decreasing order:



```
\usetikzlibrary {animations}
\tikz [animate = {
    object = node, attribute = fill, time = 0s, value = red, entry,
    object = node2, attribute = draw, entry,
    object = node, attribute = fill, time = 2s, value = blue, entry,
    object = node2, attribute = draw, entry,
    object = node, attribute = fill, begin on = click, entry,
    object = node2, attribute = draw, begin on = click, entry}]
\node (node) [fill, text=white] { Node 1 };
\node (node2) [draw, ultra thick] at (0,-1) { Node 2 };
```

In the above example, we could not have exchanged the first two lines of the `animate` options with the third and fourth line since the values for time 0s must come before the values for time 2s.

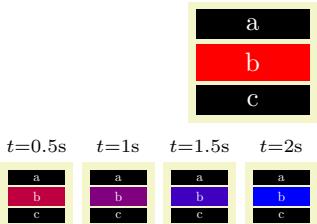
In the following, we have a closer look at the five keys that influence the `entry` key and then have a look at ways of grouping keys more easily.

26.2.3 Specifying Objects

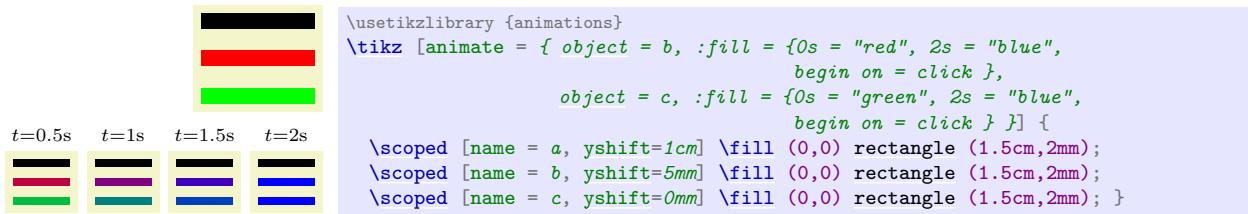
You use the `object` key to select the object(s) to which the next use of `entry` applies. There is also a special syntax for this, which is explained in Section 26.3.1.

`/tikz/animate/object=(list of objects)` (no default)

The `<list of objects>` is a comma-separated list of strings of the form `<object>.⟨type⟩`. All of the objects in the list are selected as to-be-animate object for the next use of the `entry` key. The objects referred to by `<object>` will be the *next* objects with the `name` key set to `<object>`. You can apply the `name` key to nodes (where you can also use the special parentheses-syntaxis and put the name in parentheses, it has the same effect), but also to scopes and paths. (The `name path` key is not the same as `name`; it is an older key from the intersections package and not related.)



```
\usetikzlibrary {animations}
\tikz [animate = { object = b, :fill = f0s = "red", 2s = "blue",
                    begin on = click }] {
    \node (a) [fill, text = white, minimum width=1.5cm] at (0,1cm) {a};
    \node (b) [fill, text = white, minimum width=1.5cm] at (0,5mm) {b};
    \node (c) [fill, text = white, minimum width=1.5cm] at (0,0mm) {c}; }
```



If the `<object>` name is never used later in the file, no animation is created.

The `<object>` may also be the special text `myself`. In this case, the referenced object is the scope or object to which the `animate` key is given. If an object is named `myself` (as in `\node (myself) ...`), you cannot reference this node using the `object` key, `myself` *always* refers to the object where the `animate` key is given (of course, you can animate the node named `myself` by placing the `animate` key inside the options of this node; you only cannot “remotely” add an animation to it).

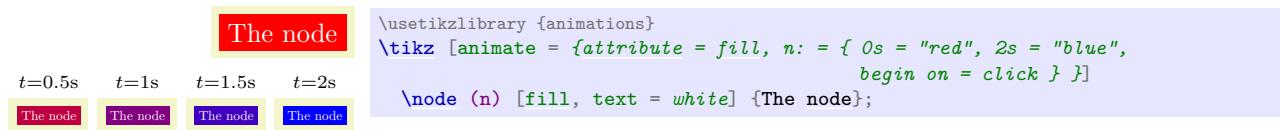
The `<object>` may be followed by a dot and a *type*. This is need in rare cases where you want to animate only a special “part” of an object that is not accessible in other ways. Normally, TikZ takes care of choosing these types automatically, you only need to set these “if you know what you are doing”.

26.2.4 Specifying Attributes

`/tikz/animate/attribute=<list of attributes>`

(no default)

The list of attributes must be a comma-separated list of attribute names. The timelines specified later will apply to all of these attributes (and to all objects previously selected using `object`). Possible attributes include colors, positions, line width, but even the paths themselves. The exact list of possible attributes is documented in Section 26.4.



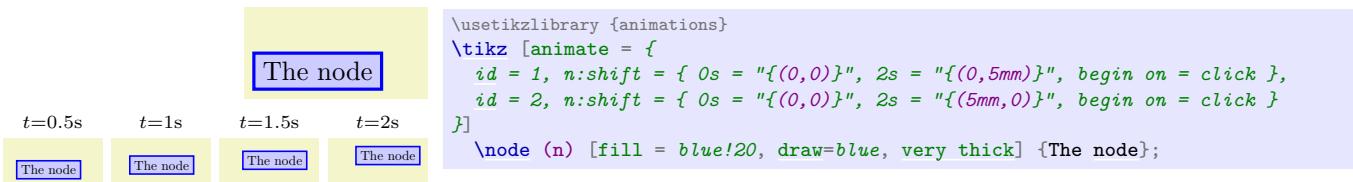
26.2.5 Specifying IDs

`/tikz/animate/id=<id>`

(no default, initially `default`)

Timelines are use to defined how the values of an attribute of an object change over time. In many cases, you will have at most one timeline for each object–attribute pair, but, sometimes, you may wish to have more than one timeline for the same object and the same attribute. For instance, you might have a timeline that specifies a changing `shift` of a node in some direction and, at the same time, another timeline that specifies an additional `shift` in some other direction(s). The problem is that there is only one `shift` attribute and it would be difficult to compute the joint effect of the two timelines.

For this purpose, timelines are actually identified not only by the object–attribute pair but, in reality, by the triple consisting of the object, the attribute, and the value of this key. We can now specify two separate timelines:



The default value of `id` is `default`.

Because of the possibility of creating multiple timelines for the same attribute, it may happen that there is more than one timeline active that is “trying to modify” a given attribute. In this case, the following rules are used to determine, which timeline “wins”:

1. If no animation is active at the current time (all animation either have not yet started or they have already ended), then the `base` value given in the animation encountered last in the code is used. (If there are no base values, the attribute is taken from the surrounding scope and the animations have “no effect”.)

2. If there are several active animations, the one that has started last is used and its value is used.
3. If there are several active animations that have started at the same time, the one that comes last in the code is used.

Note that these rules do not apply to transformations of the canvas since these are always additive (or, phrased differently, they are always all active and the effects accumulate).

26.2.6 Specifying Times

`/tikz/animate/time=<time>later`

(no default)

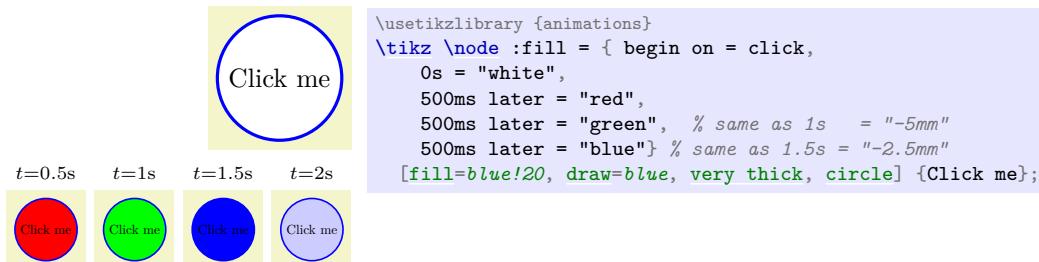
Sets the time for the next time–value pair in a call of `entry` to `<time>` plus the current fork time. The text `later` is optional. Both “fork times” and the optional `later` will be explained in a moment.

Time Parsing. The `<time>` is parsed using the command `\pgfparser{time}`, which is essentially the same as the usual math parser of TikZ, and the result is interpreted as a time in seconds. Thus, a `<time>` of `2+3` means “5 seconds” and a `<time>` of `2*(2.1)` means “4.2 seconds”. (You could even specify silly times like `1in`, which results in the time “72.27 seconds”. Please do not do that.) The “essentially” refers to the fact that some extras are installed when the time parser is running:

- The postfix operator `s` is added, which has no effect. Thus, when you write `5s` you get the same results as `5`, which is exactly 5 seconds as desired.
- The postfix operator `ms` is added, which divides a number by 1000, so `2ms` equals 0.002s.
- The postfix operator `min` is added, which multiplies a number by 60.
- The postfix operator `h` is added, which multiplies a number by 3600.
- The infix operator `:` is redefined, so that it multiplies its first argument by 60 and adds the second. This implies that `1:20` equals 80s and `01:00:00` equals 3600s.
- The parsing of octal numbers is switched off to allow things like `01:08` for 68s.

Note that you cannot use the colon syntax for times in things like `01:20 = "0"` would (falsely) be interpreted as: “For the object named `01` and its attribute named `20`, do something.” You can, however, use `01:20` in arguments to the `time` key, meaning that you would have to write instead: `time = 1:20, "0"`, possibly surround by a `scope`.

Relative Times. You can suffix a `time` key with “`later`”. In this case, the `<time>` is interpreted as an offset to the time in the previous use of the `time` key:



In reality, the offset is not taken to just any previous use of the `time` key, but to the most recent use of this key or of the `resume` key in the current local T_EX scope. Here is an example:

```
time = 2s,
time = 1s later, % same as time = 3s
time = 500ms later, % same as time = 3.5s
time = 4s,
time = 1s later, % same as time = 5s
scope = { % opens a local scope
  time = 1s later, % same as time = 6s
  time = 10s
  time = 1s later % same as time = 11s
},
% closes the scope, most recent time is 5s once more
time = 2s later % same as time = 7s
```

Fork Times. The time meant by the value `<time>` passed to the `time` key is not used directly. Rather, TikZ adds the current *fork time* to it, which is `0s` by default. You can change the fork time using the following key:

`/tikz/animate/fork=<t>` (default 0s later)

Sets the fork time for the local scope to $\langle t \rangle$ and sets the current time to 0s. In this scope, when you use “absolute” times like 0s or 2s, you actually refer to later times that have started as $\langle t \rangle$.

One application of forks is in the definition of keys that add a certain part to a longer animation. Consider for instance the definition of a `highlight` key:



In the above example, we could also have written 0.1s later instead of 0.2s and, indeed, the whole style could have been defined using only times with later, eliminating the need for the `fork` key. However, using forks you can specify absolute times for things happening in a conceptual “subprocess” and also relative times. The name `fork` for the key is also borrowed from operating system theory, where a “fork” is the spawning of an independent process.

Remembering and Resuming Times. When you have a complicated animation with a long timeline, you will sometimes wish to start some animation when some other animation has reached a certain moment; but this moment is only reached through heavy use of `later` times and/or forks. In such situations, the following keys are useful:

`/tikz/animate/remember=<macroname>` (no default)

This key stores the current time (the time of the last use of the `time` key) globally in the macro `<macroname>`. This time will include the offset of the fork time:

```
time = 2s,
fork = 2s later,    % fork time is now 4s
time = 1s,          % local time is 1s, absolute time is 5s (1s + fork time)
time = 1s later,    % local time is 2s, absolute time is 6s (2s + fork time)
remember = \mytime % \mytime is now 6s
```

`/tikz/animate/resume=<absolute time>` (no default)

The $\langle \text{absolute time} \rangle$ is evaluated using `\pgfparseftime` and, then, the current time is set to the resulting time minus the fork time. When the $\langle \text{absolute time} \rangle$ is a macro previously set using `remember`, the net effect of this is that we return to the exact “moment” in the global time line when `remember` was used.

```
fork = 4s,
time = 1s,
remember = \mytime % \mytime is now 5s
fork = 2s,          % fork time is now 2s, local time is 0s
resume   = \mytime % fork time is still 2s, local time is 3s
```

Using `resume` you can easily implement a “join” operation for forked times. You simply remember the times at the ends of the forks and then resume the maximum time of these remembered times:

```

scope = {
  fork,
  time = 1s later,
  ...
  remember = \forka
},
scope = {
  fork,
  time = 5s later,
  ...
  remember = \forkb
},
scope = {
  fork,
  time = 2s later,
  ...
  remember = \forkc
},
resume = {max(\forka,\forkb,\forkc)} % "join" the three forks

```

26.2.7 Values

`/tikz/animate/value=<value>` (no default)

This key sets the value of the next time–value pair created by `entry` to `<value>`. The syntax of the `<value>` is not fixed, it depends on the type of the attribute. For instance, for an attribute like `opacity` the `<value>` must be an expression that can be evaluated to a number between 0 and 1; for the attribute `color` the `<value>` must, instead, be a color; and so on. Take care that when a value contains a comma, you must surround it by braces as in "`\{(1,1)\}`".

The allowed texts for the `<value>` is always the same as the one you would pass to the TikZ option of the same name. For instance, since the TikZ option `shift` expects a coordinate, you use coordinates as `<value>` with the usual TikZ syntax (including all sorts of extensions, the animation system calls the standard TikZ parsing routines). The same is true of dimensions, scalar values, colors, and so on.

In addition to the values normally use for setting the attribute, you can also (sometimes) use the special text `current value` as `<value>`. This means that the value of the point in the timeline should be whatever the value the attribute has at the beginning of the timeline. For instance, when you write

```
animate = { obj:color = { 0s = "current value", 2s = "white" } }
```

the color of `obj` will change from whatever color it currently has to white in two seconds. This is especially useful when several animations are triggered by user events and the current color of `obj` cannot be determined beforehand.

There are several limitations on the use of the text `current value`, which had to be imposed partly because of the limited support of this feature in SVG:

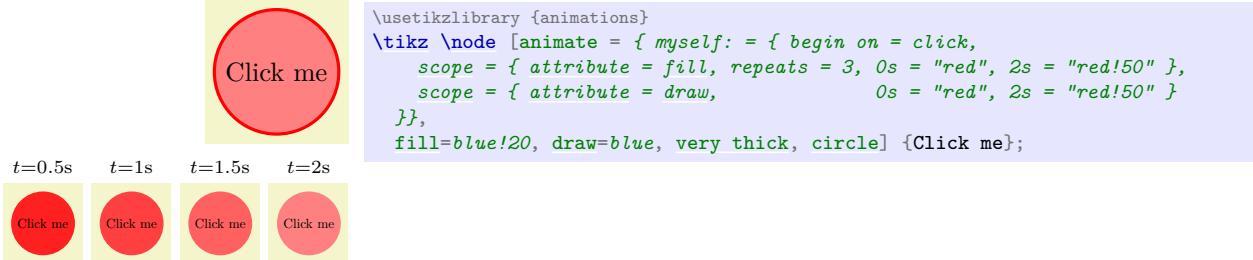
- You can use `current value` only with the first time in a timeline.
- You can only have two times in a timeline that starts with `current value`.
- You cannot use `current value` for timelines of which you wish to take a snapshot.

26.2.8 Scopes

When you specify multiple timelines at the same time, it is often useful and sometimes even necessary to have keys be set only locally. The following key makes this easy:

`/tikz/animate/scope=<options>` (no default)

Executed the `<options>` inside a TeX scope. In particular, all settings made inside the scope have no effect after the end of the `scope`.



Without the use of the `scope` key, the `repeats` key would also affect the `draw` attribute.

While the `scope` key is useful for structuring timeline code, it also keeps the current time local to the scope, that is, if you use something like `1s later` after the scope, this will refer to one second after the last use of `time before` the scope. The times set inside the `scope` do not matter. While this is desirable effect for forks, you may also sometimes wish to synchronize the local time after the scope with the last time reached in the scope. The following key makes this easy:

`/tikz/animate-sync=<options>` (no default)

A shorthand for `scope={<options>} , remember=\temp, resume=\temp` where `\temp` is actually an internal name. The effect is that after a `sync` the local time just continues as if the scope were not present – but regarding everything else the effects are local to the `sync` scope.

26.3 Syntactic Simplifications

In the previous subsection we saw how timelines can be created by specifying the individual entries of the timelines sequentially. However, most of the time you will wish to use a simpler syntax that makes it easier to specify animations. This syntax is only available inside the `animate` key (it is switched on at the beginning) and consists of three “parts”: The colon syntax, the time syntax, and the quote syntax.

26.3.1 The Colon Syntax I: Specifying Objects and Attributes

Inside the `<animation specification>` passed to the `animate` key, you can specify an object and an attribute of this object using the following syntax, whose use is detected by the presence of a colon inside a key:

`<object name(s)>:<attribute(s)> ={<options>}`

or

`<object name(s)>:<attribute(s)>_<id> ={<options>}`

In the place to the left of an equal sign, where you would normally use a key, you can instead place an object name and an attribute separated by a colon. Additionally, the attribute may be followed by an underscore and an `<id>`, which identifies the timeline (see Section 26.2.5).

Each of these values may be missing, in which case it is not changed from its previous value.

The effect of the above code is the same as:

`sync = { object = <objects>, attribute = <attribute>, id = <id>, <options>, entry }`

although when the object, the attribute, or the id is left empty in the colon syntax, the corresponding setting will be missing in the above call of `sync`. Note that because of the `sync` the last time used inside the `<options>` will be available afterwards as the last time. Also note that an `entry` is added at the end, so any settings of keys like `begin` or `repeats` inside the `<options>` will get added to the timeline.

Let us now have a look at some examples. First, we set the `<object name>` to `mynode` and `othernode` and the `<attribute>` to `opacity` and to `color`:

```
animate = {
  mynode:opacity    = { 0s = "1",   5s = "0" },
  mynode:color      = { 0s = "red", 5s = "blue" },
  othernode:opacity = { 0s = "1",   5s = "0" },
}
```

Next, we do the same, but “in two steps”: First, we set the object to `mynode`, but leave the attribute open and, then, set the attribute, but leave the object:

```

animate = {
  mynode: = {
    :opacity      = { 0s = "1",   5s = "0" },
    :color        = { 0s = "red",  5s = "blue" }
  },
  othernode:opacity = { 0s = "1",   5s = "0" },
}

```

Note how both in `mynode:` and in `:opacity` and `:color` you must provide the colon. Its presence signals that an object–attribute pair is being specified; only now either the object or the attribute is missing.

We can also do it the other way round:

```

animate = {
  :opacity = {
    mynode:      = { 0s = "1",   5s = "0" },
    othernode:   = { 0s = "1",   5s = "0" }
  },
  mynode:color = { 0s = "red",  5s = "blue" }
}

```

Finally, if several objects should get the exact same values, we can also group them:

```

animate = {
  {mynode, othernode}:opacity = { 0s = "1",   5s = "0" },
  mynode:color               = { 0s = "red",  5s = "blue" }
}

```

As mentioned earlier, all references to objects will be interpreted to future objects, never to objects already created. Furthermore, also as mentioned earlier, TikZ allows you to specify `myself` as `<object>`, which is interpreted as the scope or node where the `animate` is given (you cannot animate a node or scope named `myself`, this special name always refers to the current node). In order to have all attributes refer to the current object, you write:

```

\begin{scope} [animate = {
  myself: = { % Animate the attribute of the scope
    :opacity = { ... },
    :xshift = { ... }
  }
}]
...
\end{scope}

```

The list of permissible attributes is given in Section 26.4.

26.3.2 The Colon Syntax II: Animating Myself

A frequent use of the `animate` key is for animating attributes of the current object `myself`. In these cases, it is a bit length to write

```
[animate = { myself: = { :some attribute = {...} } } ]
```

in the options of a node or a scope. For this reason, TikZ allows you to use a special syntax with nodes and scopes:

1. In a `<node specification>`, which is everything following a `node` command up to the content of the node (which is surrounded by curly braces), you can write

```
:some attribute = {<options>}
```

and this will have the same effect as if you had written

```
[animate = { myself: = { :some attribute = {<options>} } } ]
```

Note that you can use this syntax repeatedly, but each use creates a new use of the `animate` key, resulting in a new timeline. In order to create complex timelines for several objects, use the `animate` key.

2. For the commands `\tikz`, `\scoped` and the environments `{tikzpicture}` and `{scope}`, when they are followed immediately by

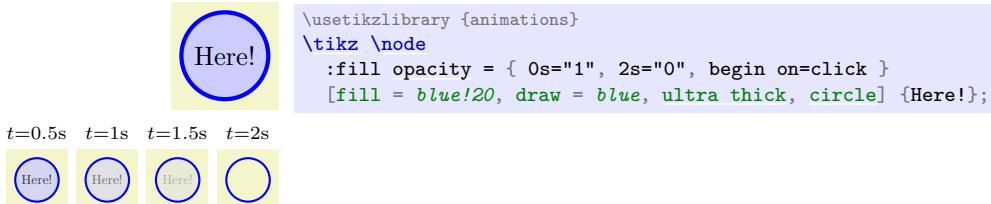
```
:some attribute = {<options>}
```

then

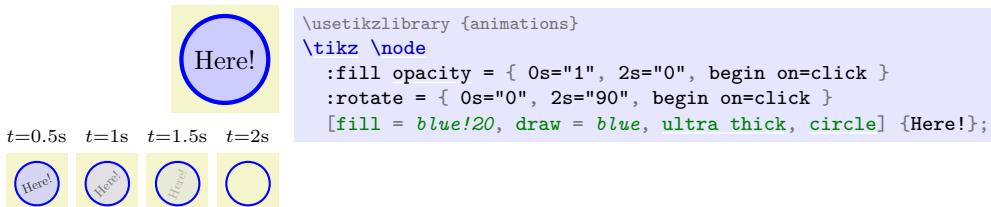
```
animate = { myself: = { :some attribute = {<options>} } }
```

is added to the options of the command or scope. Again, you can use the syntax repeatedly. Note that when an opening square bracket is encountered, this special parsing stops.

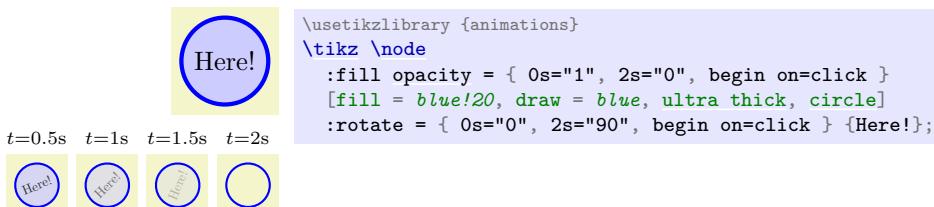
Let us have a look at some examples. First, we use the syntax to set the fill opacity of a node:



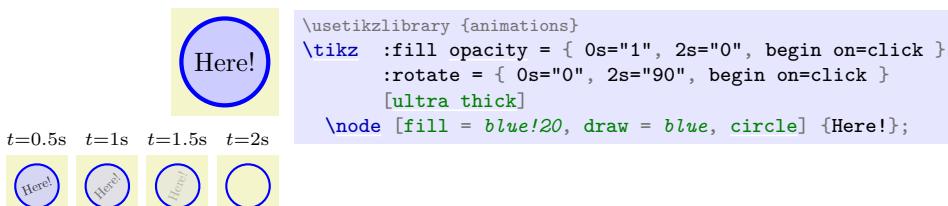
Next, we additionally rotate the node:



Note that there is no comma between consecutive uses of the colon syntax in this case. We could have exchanged the order of the options and the uses of the colon syntax:



We can also use the special syntax with the `\tikz` command itself:



Note that we could *not* have moved the `[ultra thick]` options before `:rotate` since the options in square brackets end the special parsing.

26.3.3 The Time Syntax: Specifying Times

For each object–attribute pair you must specify the *timeline* of the attribute. This is a curve that specifies for each “moment in time” which value the attribute should have. In the simplest case, you specify such a time–value pair as follows:

$$\langle \text{time} \rangle = \langle \text{value} \rangle$$

When you specify time–value pairs, you must specify the times in chronological order (so earlier times come first), but you may specify the same time several times (this is useful in situations where you have a “jump” from one value to another at a certain moment in time: you first specify the value “from which the attribute jumps” and then you specify the value “to which the attribute jumps” for the same moment).

The above syntax is just a special case of a more general situation. Let us start with the times. The general syntax for specifying times is as follows:

$$\langle \text{time} \rangle = \langle \text{options} \rangle$$

Here, $\langle \text{time} \rangle$ is a text that “looks like a time”, which means that:

1. It is not a key and does not contain a colon and does not start with a quotation mark.
2. It starts with a digit, a plus or minus sign, a dot, or a parenthesis.

If these two things are the case, the above code is transformed to the following call:

```
sync = {time = <time>, <options>, entry}
```

26.3.4 The Quote Syntax: Specifying Values

We saw already in several examples that values are put in quotation marks (similar to the way this is done in XML). This quote syntax is as follows:

```
"<value>" base = <options>
```

This syntax is triggered whenever a key starts with a quotation mark⁵ (and note that when the *<value>* contains a comma, you have to surround it by curly braces *inside* the quotation marks as in "*{(1,1)}*"). Then, the following code is executed:

```
sync = {value = <value>, <options>, entry}
```

This means that when you write `1s = "red"`, what actually happens is that TikZ executes the following:

```
sync = { time = 1s, sync = { value = red, entry }, entry }
```

Note that the second entry has no effect since no value is specified and the `entry` key only “takes action” when both a time and a value have been specified. Thus, only the innermost `entry` does, indeed, create a time–value pair as desired.

In addition to the above, if you have added `base` after the closing quote, the following gets executed before the above `sync`:

```
base = {value = <value>}
```

This makes it easy to specify base values for timelines.

Interestingly, instead of `1s="red"` you can also write `"red"=1s`. Let us now have a look at situations where this can be useful.

26.3.5 Timesheets

Using the `sync` key or using the three different syntactic constructs introduced earlier (the color syntax, the time syntax, the value syntax), you can organize the specification of an animation in different ways. Basically, the two most useful ways are the following:

1. You first select an object and an attribute for which you wish to establish a timeline and then provide the time–value pairs in a sequence:

```
animate = [
  obj:color = {
    0s = "red",
    2s = "blue",
    1s later = "green",
    1s later = "green!50!black",
    10s = "black"
  }
]
```

When you specify timelines for several attributes of the same object, you can group these together:

```
animate = [
  obj: = {
    :color = { 0s = "red", 2s = "green" },
    :opacity = { 0s = "1", 2s = "0" }
  }
]
```

In this way of specifying animations the “object comes first”.

⁵Of catcode 12 for those knowledgeable of such things.

2. Alternatively, you can also group the animation by time and, for each “moment” (known as *keyframes*) you specify which values the attributes of the object(s) have:

```
animate = {
  0s = {
    obj:color = "red",
    obj:opacity = "1"
  },
  2s = {
    obj:color = "green",
    obj:opacity = "0"
  }
}
```

Naturally, in this case it would have been better to “move the object outside”:

```
animate = {
  obj: = {
    0s = {
      :color = "red",
      :opacity = "1"
    },
    2s = {
      :color = "green",
      :opacity = "0"
    }
  }
}
```

When there are several objects involved, we can mix all of these approaches:

```
animate = {
  0s = {
    obj: = {
      :color = "red",
      :opacity = "1"
    },
    main node: = {
      :color = "black"
    }
  },
  2s = {
    obj: = {
      :color = "green",
      :opacity = "0"
    },
    main node: = {
      :color = "white"
    }
  }
}
```

26.4 The Attributes That Can Be Animated

The following *<attributes>* are permissible (actually, the attribute names do not include a colon, but since they will almost always be used with the colon syntax, it makes it easier to identify them):

- :dash phase
- :dash pattern
- :dash
- :draw opacity
- :draw
- :fill opacity
- :fill
- :line width

- `:opacity`
- `:position`
- `:path`
- `:rotate`
- `:scale`
- `:stage`
- `:text opacity`
- `:text`
- `:translate`
- `:view`
- `:visible`
- `:xscale`
- `:xshift`
- `:xskew`
- `:xslant`
- `:yscale`
- `:yshift`
- `:yskew`
- `:ylslant`

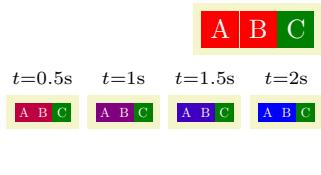
These attributes are detailed in the following sections, but here is a quick overview of those that do not have a TikZ key of the same name (and which thus do not just animate the attribute set using this key):

- `:shift` allows you to add an animated shifting of the canvas, just like TikZ's `shift` key. However, in conjunction with the `along` key, you can also specify the shifting along a path rather than via a timeline of coordinates.
- `:position` works similar to `:shift`, only the coordinates are not relative movements (no “shifts”), but refer to “absolute positions” in the picture.
- `:path` allows you to animate a path (it will morph). The “values” are now paths themselves.
- `:view` allows you to animate the view box of a view.
- `:visible` decides whether an object is visible at all.
- `:stage` is identical to `:visible`, but when the object is not animated, it will be hidden by default.

26.4.1 Animating Color, Opacity, and Visibility

You can animate the color of the target object of an animation using the attributes `fill`, `draw`, and `text`. When the target of a color animation is a scope, you animate the color “used in this scope” for filling or stroking. However, when an object inside the scope has its color set explicitly, this color overrules the color of the scope.

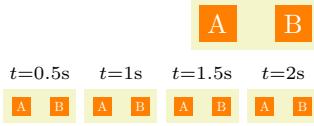
Animation attribute `:fill`, `:draw`



```
\usetikzlibrary {animations}
\tikz :fill = {0s = "red", 2s = "blue", begin on = click}
          [text = white, fill = orange] {
    \node [fill]                      at (0mm,0) {A};
    \node [fill]                      at (5mm,0) {B};
    \node [fill = green!50!black] at (1cm,0) {C};
}
```

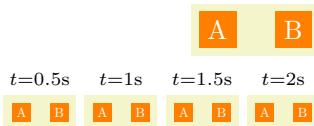
Animation attribute :text

The `text` attribute only applies to nodes and you need to directly animate the `text` attribute of each node individually.



```
\usetikzlibrary {animations}
\tikz [my anim/.style={ animate = {
    myself:text = {0s = "red", 2s = "blue", begin on = click}}},
    text = white, fill = orange ] {
    \node [fill, my anim] at (0,0) {A};
    \node [fill, my anim] at (1,0) {B};
}
```

Unlike the `fill` and `draw` colors, you cannot animate the `text` color for scopes:



```
\usetikzlibrary {animations}
\tikz [animate = {myself:text = {0s = "red", 2s = "blue",
begin on = click}},
    text = white, fill = orange ] {
    \node [fill] at (0,0) {A};
    \node [fill] at (1,0) {B};
}
```

Animation attribute :color

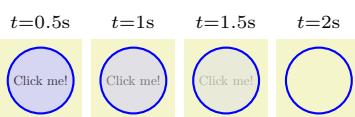
The `color` attribute is not really an attribute. Rather, it is a shorthand for `{draw,fill,text}`. This means that `color` does not start a separate timeline, but continues the `draw` timeline, the `fill` timeline, and the `text` timeline.

Animation attribute :opacity, :fill opacity, :stroke opacity

Similarly to the color, you can also set the opacity used for filling and for drawing using the attributes `fill opacity` and `draw opacity`, which are exactly the same as the usual TikZ keys of the same names.



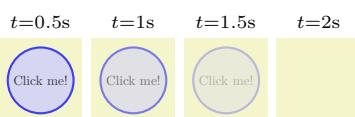
```
\usetikzlibrary {animations}
\tikz \node [fill opacity = { 0s="1", 2s="0", begin on=click }] [fill = blue!20, draw = blue, ultra thick, circle] {Click me!};
```



Unlike colors, where there is no joint attribute for filling and stroking, there is a single `opacity` attribute in addition to the above two attributes. If supported by the driver, it treats the graphic object to which it is applied as a transparency group. In essence, “this attribute does what you want” at least in most situations.

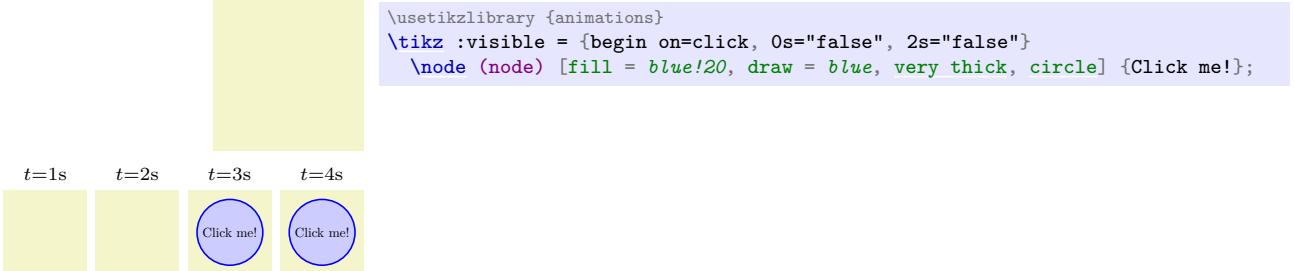


```
\usetikzlibrary {animations}
\tikz \node [opacity = { 0s="1", 2s="0", begin on=click }] [fill = blue!20, draw = blue, ultra thick, circle] {Click me!};
```

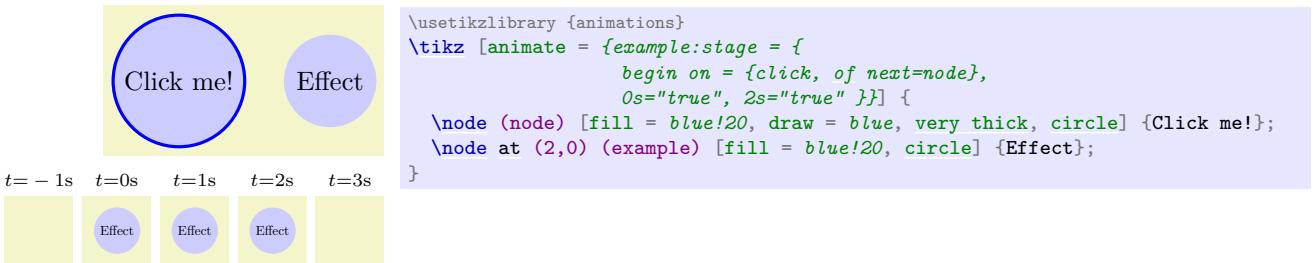


Animation attribute :visible, :stage

The difference between the `visible` attribute and an opacity of 0 is that an invisible object cannot be clicked and does not need to be rendered. The (only) two possible values for this attribute are `false` and `true`.



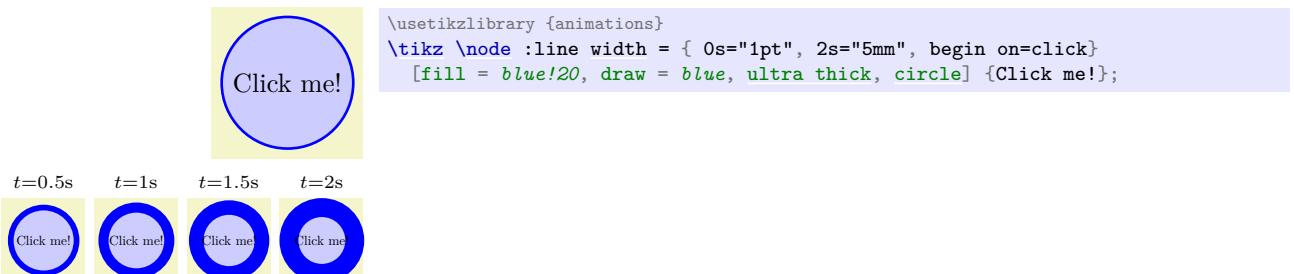
This `stage` attribute is the same as the `visible` attribute, only `base="false"` is set by default. This means that the object is *only* visible when you explicitly during the time the entries are set to `true`. The idea behind the name “stage” is that the object is normally “off stage” and when you explicitly set the “stage attribute” to `true` the object “enters” the stage and “leaves” once more when it is no longer “on stage”.



26.4.2 Animating Paths and their Rendering

The attributes of the appearance of a path that you can animate include the line width and the dash pattern, the path itself, as well as the arrow tips attached to the paths. Animating the line width and the dash pattern is easy since the animation attributes simply have that same names as the properties that they animate and the syntax for setting is also the same:

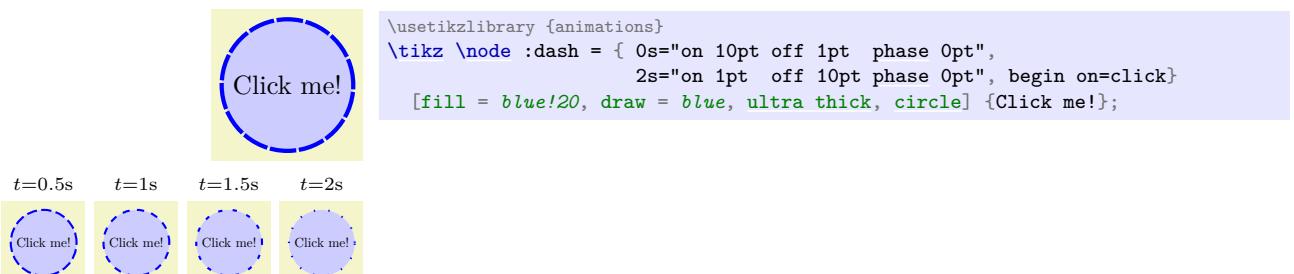
Animation attribute :line width

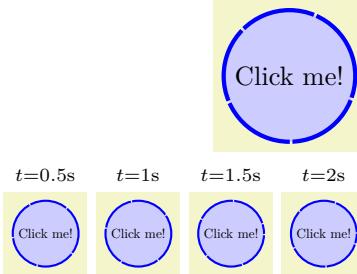


Note that you must specify number (or expressions that evaluate to numbers) as values, you cannot say `thin` or `thick` (these are styles, internally, and you also cannot say `line width=thick`).

Animation attribute :dash, :dash phase, :dash phase

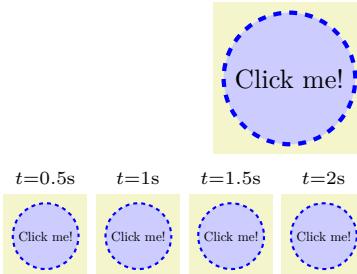
The values for an animation of the dashing are specifications (see the `dash` key for details) consisting of a sequence of `on` and `off` numbers. In each value of the animation the length of these sequences *must* be identical. The interpolation of the values is done for each position of the sequences individually, and also on the phase.





```
\usetikzlibrary {animations}
\tikz \node :dash = { 0s="on 1cm off 1pt phase 0pt",
                      2s="on 1cm off 1pt phase 1cm", begin on=click}
[fill = blue!20, draw = blue, ultra thick, circle] {Click me!};
```

This **dash** key allows you to animate the dash phase only. However, due to the way dashing is handled by certain drivers, the dash pattern is also set, namely to the current dash pattern that is in force when the animation is created.

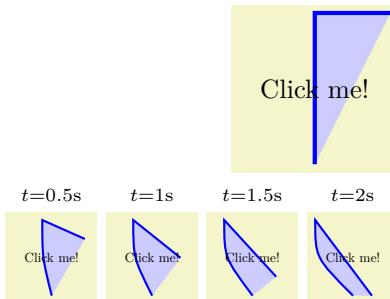


```
\usetikzlibrary {animations}
\tikz \node :dash phase = { 0s="0pt", 2s="1cm", begin on=click}
[fill = blue!20, draw = blue, ultra thick, circle, dashed] {Click me!};
```

The above attributes “only” influence how the path is rendered. You can, however, also animate the path itself:

Animation attribute :path

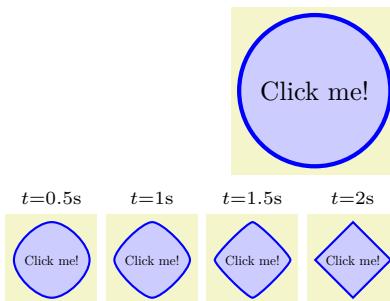
When you animate a path, the values are, of course, paths themselves:



```
\usetikzlibrary {animations}
\tikz \node :path = {
  0s = "{(0,-1) .. controls (0,0) and (0,0) .. (0,1) -- (1,1)}",
  2s = "{(0,-1) .. controls (-1,0) and (-1,0) .. (-1,1) -- (.5,-1)}",
  begin on=click }
[fill = blue!20, draw = blue, ultra thick, circle] {Click me!};
```

There are a number of things to keep in mind when you animate a path:

- The path “values” are parsed and executed in an especially protected scope to ensure that they have only little side effects, but you should not do “fancy things” on these paths.
- As for the dash pattern, you must ensure that all paths in the timeline have the same structure (same sequence of path construction commands); only the coordinates may differ. In particular, you cannot say that the path at 1s is a rectangle using `rectangle` and at 2s is a circle using `circle`. Instead, you would have to ensure that at both times the path consists of appropriate Bézier curves (which is cumbersome as the following example shows, where we used the fact that a circle consists of four Bézier curves):



```
\usetikzlibrary {animations}
\tikz \node :path = {
  0s = "(0,0) circle [radius=1cm]",
  2s = "(0,0)
        (1,0) .. controls +(0,0) and +(0,0) .. (0,1)
        .. controls +(0,0) and +(0,0) .. (-1,0)
        .. controls +(0,0) and +(0,0) .. (0,-1)
        .. controls +(0,0) and +(0,0) .. (1,0)
        -- cycle (0,0)",
  begin on=click }
[fill = blue!20, draw = blue, ultra thick, circle] {Click me!};
```

- You must specify arrow tips for an animated path in a special way, namely using the `arrows` key for *animations*, not the normal `arrows` key (see below).

`/tikz/animate/arrows=(arrow spec)` (no default)

This key only has an effect on `:path` animations. It causes the arrow tips specified in `(arrow spec)` to be added to the path during the animation (the syntax is the same as for the normal `arrows` key). If you have several different animations for a paths, these may contain different arrow tips, but each animation must stick to one kind of arrow tips.

What happens internally when this key is used is the following: The specified arrow tips are rendered internally as so-called *markers*, which are small graphics that can be placed at the beginning and ends of paths and which “rotate along” as a path changes. Note that these markers are used *only* in the context of animated paths, the arrow tips of normal, “static” paths are drawn without the use of markers. Normally, there is no visual difference between an arrow tip drawn using markers or those drawn for static paths, but in rare cases there may be differences. You should only add arrows to open path consisting of a single segment with sufficiently long first and last segments (so that TikZ can shorten these segments correctly when necessary).

As pointed out earlier, the only way to add arrow tips to a path that is animated is using this key, you can *not* say something like

```
\draw :path = { 1s = "[(0,0) -- (1,0)]", 2s = "[{(0,1) -- (1,0)}" }
  [->] (0,0) -- (1,0);
```

This will raise an error since you try to animate a path (`:path = ...`) that has normal arrow tips attached (`[->]`).

Instead, you must specify the arrow tips inside the animation command:

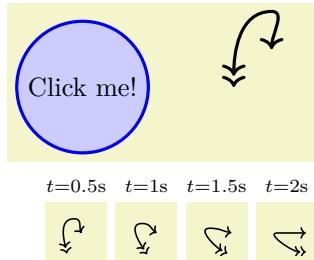
```
\draw :path = { 1s = "[(0,0) -- (1,0)]", 2s = "[{(0,1) -- (1,0)}", arrows = -> }
  (0,0) -- (1,0);
```

However, the above code now has a big shortcoming: While the animation is *not* running, *no* arrow tip is shown (the `arrows` key only applies to the animation).

The trick is to use the `base` key. It allows you to install a path as the “base” path that is used when no animation is running and the arrows specified for the animation will also be used for the base. All told, the “correct” way to specify the animation is the following (note that no static path is specified, any specified path would be overruled by the `base` path anyway):

```
\draw :path = { 1s = "[(0,0) -- (1,0)]" base, 2s = "[{(0,1) -- (1,0)}", arrows = -> };
```

Here is an example:



```
\usetikzlibrary {animations}
\tikz [very thick] {
  \node (node) at (-2,0)
    [fill = blue!20, draw = blue, very thick, circle] {Click me!};
  \draw :path = {
    0s = "[(0,0) to [out=90, in=180] (.5,1) to [out=0, in=90] (.5,.5)]" base,
    2s = "[{(1,0) to [out=180, in=180] (.25,.5) to [out=0, in=180] (1,.5)}",
    arrows = <.=>, begin on = {click, of=node} }; }
```

`/tikz/animate/shorten < = (dimension)` (no default)

`/tikz/animate/shorten > = (dimension)` (no default)

For animated paths, just as the key `arrows` has to be passed to the animation (to `:path`) instead of to the static path, the keys `shorten >` and `shorten <` also have to be passed to the `:path` key.

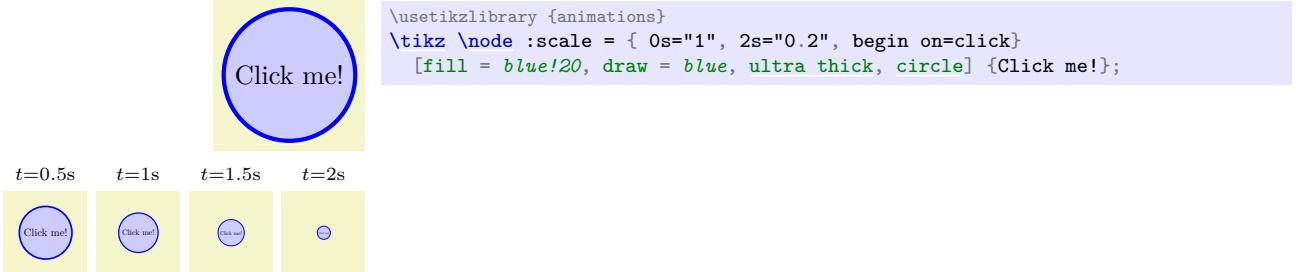
26.4.3 Animating Transformations: Relative Transformations

In order to animate the canvas transformation matrix, you do not animate an attribute called “`:transform`”. Rather, there are several attributes that all manipulate the canvas transformation matrix in

different ways. These keys, taken in appropriate combination, allow you to achieve any particular canvas transformation matrix. All keys that animate the transformation matrix always accumulate.

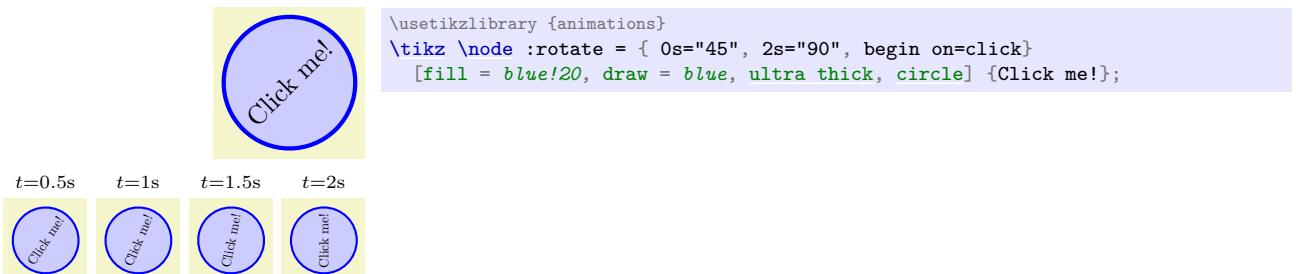
Let us start with the “standard” attributes that are also available as keys in TikZ:

Animation attribute :`scale`, :`xscale`, :`yscale`



Animation attribute :`rotate`

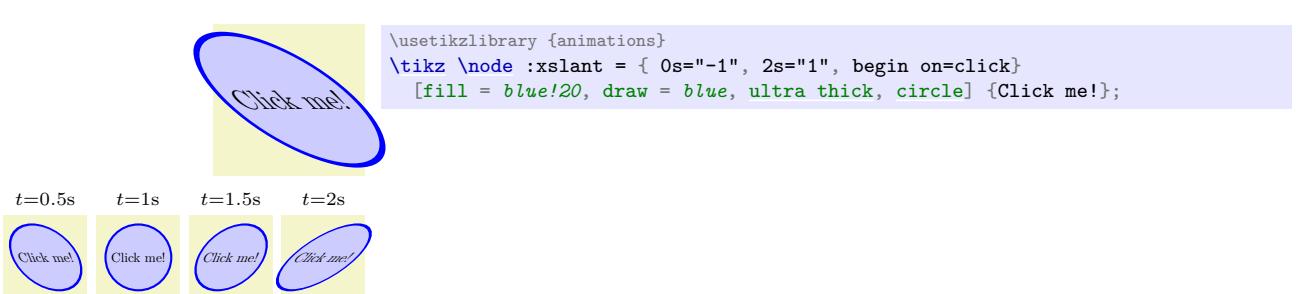
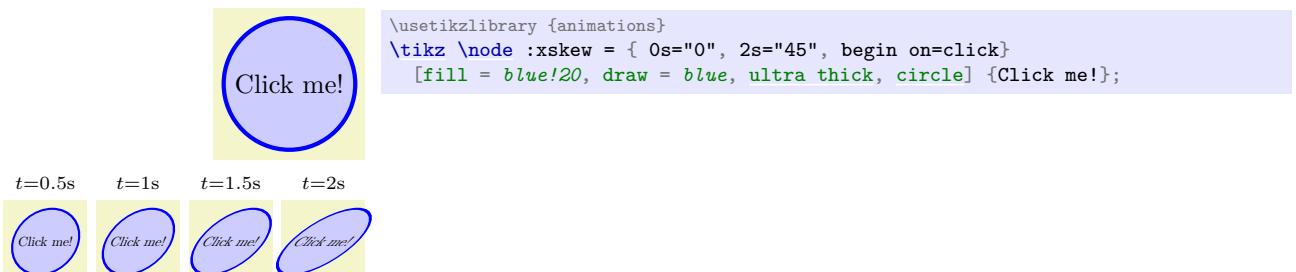
The `rotate` key adds an animation of the rotation:



Note that there is no `rotate around` attribute, but you can use the `origin` key to change the origin of the rotation.

Animation attribute :`xskew`, :`yskew`, :`xslant`, :`yslant`

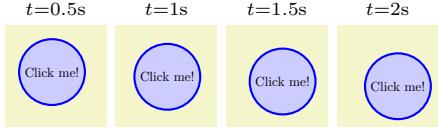
The keys add an animation of the skew (given in degrees) or slant (given as in the `xslant` and `yslant` key):



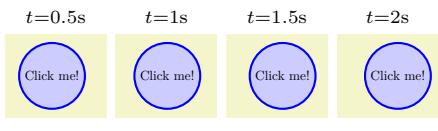
Animation attribute :`xshift`, :`yshift`



```
\usetikzlibrary {animations}
\tikz \node :shift = { 0s="{}{(0,0)}", 2s="{}{(5mm,-5mm)}",
begin on=click}
[fill = blue!20, draw = blue, ultra thick, circle] {Click me!};
```



```
\usetikzlibrary {animations}
\tikz \node :xshift = { 0s="{}{Opt", 2s="{}{5mm", begin on=click}
[fill = blue!20, draw = blue, ultra thick, circle] {Click me!};
```

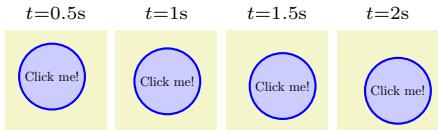


Animation attribute :shift

This :shift attribute can be animated in two ways. First, you can simply specify a sequence of coordinates in the same way as you would use the shift key in TikZ:



```
\usetikzlibrary {animations}
\tikz \node :shift = { 0s = "({(0,0)}", 2s = "({(5mm,-5mm)})",
begin on = click }
[fill = blue!20, draw = blue, ultra thick, circle] {Click me!};
```

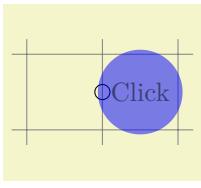


However, you can also specify the sequence of positions along which the shift should occur in a different way, namely by *specifying a path along which the object should be moved*. This is often not only more natural to do, but also allows you to specify movements along curves.

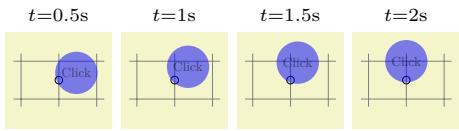
`/tikz/animate/options/along={⟨path⟩}{⟨sloped or upright⟩}in⟨time⟩`

(no default)

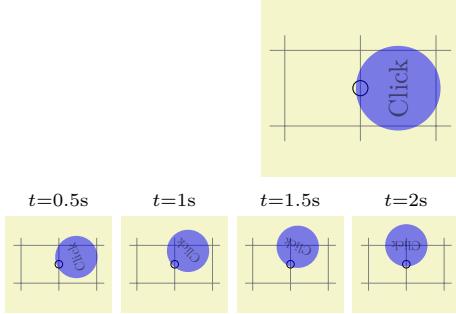
Use this key with a :shift (or a :position) to make TikZ shift the object by the coordinates along the ⟨path⟩. When this key is used, the values may no longer be coordinates, but must be fractions of the distance along the path. A value of "0" refers to the beginning of the path and "1" refers to the end:



```
\usetikzlibrary {animations}
\tikz {
\draw [help lines] (-0.2,-0.2) grid (2.2,1.2);
\draw (1,.5) circle [radius=1mm];
\node :shift = {
    along = {({(0,0)} circle[radius=5mm]} upright,
    0s="{}{0", 2s=".25", begin on=click }
    at (1,.5) [fill = blue, opacity=.5, circle] {Click};
}
```



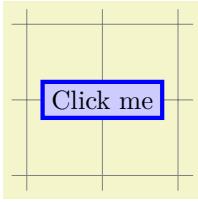
Following the ⟨path⟩, which must be put in braces, you must either specify upright or sloped. In the first case, the to-be-animated object is moved along the path normally (and stays “upright”), whereas when you use sloped, the object will be continuously rotated so that it always points along the path.



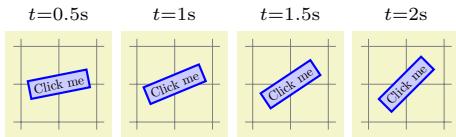
```
\usetikzlibrary {animations}
\begin{tikzpicture}
  \draw [help lines] (-0.2,-0.2) grid (2.2,1.2);
  \draw (1,.5) circle [radius=1mm];
  \node :shift = {
    along = {(0,0)} circle [radius=5mm] sloped,
    0s="0", 2s=".25", begin on=click
  }
  at (1,.5) [fill = blue, opacity=.5, circle] {Click};
\end{tikzpicture}
```

In most motion animations that use `along`, you will set the value for `0s` to "0" and the value for some specific `<time>` to "1". Because of this, you can add `in <time>` after the path, to achieve exactly this effect.

For the above attributes, it is not immediately clear which coordinate system should be used for the animation. That is, when you move an object 1cm "to the right", where is "the right"? By default, movements and transformations like `:shift` or `:scale` are relative to the *animation coordinate system*, which defaults to the local coordinate system of the to-be-animated object. Consider the following example:

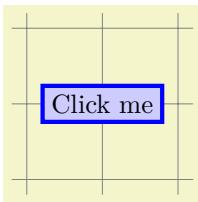


```
\usetikzlibrary {animations}
\begin{tikzpicture}
  \draw [help lines] (-0.2,-0.2) grid (2.2,2.2);
  \node :rotate = {
    0s="0", 2s="45", begin on=click
  }
  at (1,1) [fill = blue!20, draw = blue, ultra thick] {Click me};
\end{tikzpicture}
```

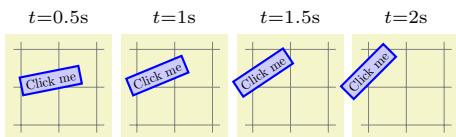


Note how the node rotates around its center even though this center is at position $(1,1)$ in the picture's coordinate system. This is because `at (1,1)` actually only does a shift of the coordinate system and the node is then drawn at the origin of this shifted coordinate system. Since this shifted coordinate system becomes the animation coordinate system, the rotation "around the origin" is actually a rotation around the origin of the animation coordinate system, which is at $(1,1)$ in the picture's coordinate system.

Let us, by comparison, do a rotation of a scope surrounding the node where the origin is not (yet) shifted:



```
\usetikzlibrary {animations}
\begin{tikzpicture}
  \draw [help lines] (-0.2,-0.2) grid (2.2,2.2);
  \begin{scopeds} :rotate = {
    0s="0", 2s="45", begin on={click, of next=n}
  }
    \node (n) at (1,1) [fill = blue!20, draw = blue, ultra thick] {Click me};
  \end{scopeds}
\end{tikzpicture}
```

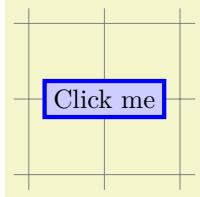


Now the rotation is really around the origin of the picture.

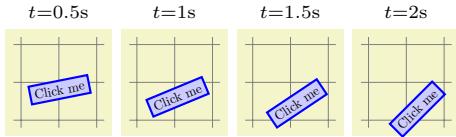
Most of the time the animation coordinate system will be setup in the way "you expect", but you can modify it using the following keys:

`/tikz/animate/options/origin=<coordinate>` (no default)

Shifts the animation coordinate system by `<coordinate>`. This has the effect that the "origin" for scalings and rotations gets shifted by this amount. In the following example, the point around which the rotation is done is the right border at $(2,1)$ since the origin of the animation is at $(1,1)$ relative to the picture's origin and the `origin` key shifts it one centimeter to the right.



```
\usetikzlibrary {animations}
\begin{tikzpicture}
  \draw [help lines] (-0.2,-0.2) grid (2.2,2.2);
  \node :rotate = { 0s="0", 2s="45", begin on=click,
                    origin = {(1,0)}}
    at (1,1) [fill = blue!20, draw = blue, ultra thick] {Click me};
\end{tikzpicture}
```



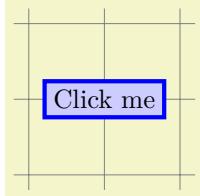
`/tikz/animate/options/transform=(transformation keys)`

(no default)

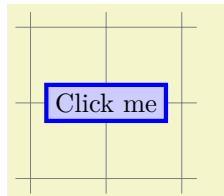
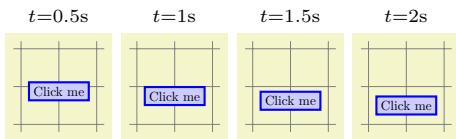
While the `origin` key does only a shift, the `transform` key allows you to add an arbitrary transformation to the animation coordinate system using keys like `shift`, `rotate` or even `reset cm` and `cm`. In particular, `origin=<c>` has the same effect as `transform = {shift=<c>}`. Note that the transformation only influences the animation, not the object itself.

As an example, when you say `transform={scale=2}`, an `:xshift` with a value of "1cm" will actually shift the object by 2cm. Similarly, after you say `transform={rotate=90,scale=2}`, the same `:xshift` of "1cm" will actually shift the object by 2cm upwards.

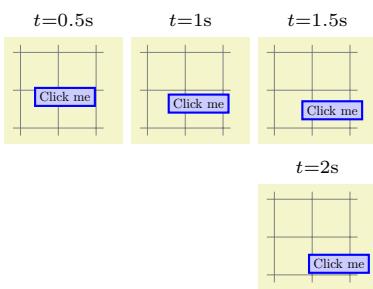
Note that, internally, TikZ has to invert the transformation matrix resulting from the `<transformation keys>` (plus the original animation transformation matrix), which can be numerically unstable when you use ill-conditioned transformations.



```
\usetikzlibrary {animations}
\begin{tikzpicture}
  \draw [help lines] (-0.2,-0.2) grid (2.2,2.2);
  \node :xshift = { 0s="0cm", 2s="5mm", begin on=click,
                    transform = {rotate=-90} }
    at (1,1) [fill = blue!20, draw = blue, ultra thick] {Click me};
\end{tikzpicture}
```



```
\usetikzlibrary {animations}
\begin{tikzpicture}
  \draw [help lines] (-0.2,-0.2) grid (2.2,2.2);
  \node :xshift = { 0s="0cm", 2s="5mm", begin on=click,
                    transform = {rotate=-45, scale=2} }
    at (1,1) [fill = blue!20, draw = blue, ultra thick] {Click me};
\end{tikzpicture}
```



26.4.4 Animating Transformations: Positioning

The attributes for specifying transformations and, in particular, the `:shift` attribute are always expressed in the local animation coordinate system. This makes it easy to “shift around a node a little bit”, but makes it hard to move a node “from one position to another” since coordinates need to be expressed relative to the node’s coordinate system, which leads to all sorts of problems: Suppose you wish to have a

node move from $(1, 1)$ to $(2, 1)$ and then to $(2, 0)$. Now, if the node has already been placed at $(1, 1)$ in the usual manner using `at`, then from the “node’s point of view” you need to move the node to $(0, 0)$, $(1, 0)$, and $(1, -1)$. To make matters worse, when you use named coordinates as in

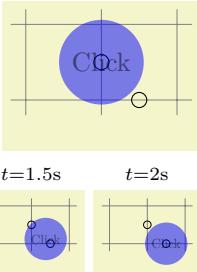
```
\coordinate(A) at (1,1);
\coordinate(B) at (2,1);
\coordinate(C) at (2,0);
```

and then say that the movement should be from `(A)` to `(B)` to `(C)`, what should you expect? On the one hand, `(A)` and $(1, 1)$ should normally be interchangeable; on the other hand, `(A)` is a specific point in the plane, no matter from which coordinate system we look at it. It turns out that TikZ will stick to the second interpretation and actually turn `(A)` into $(0, 0)$ when it is parsed in the local coordinate system of a node starting at `(A)` – while $(1, 1)$ will stay the same.

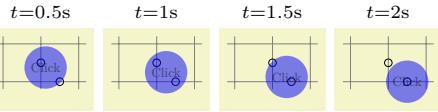
Because of all these confusing effects, there is another attribute `:position`, which is similar to a `:shift`, but the coordinates are not interpreted in the local coordinate system of the node, but in the coordinate system that is in force when the `animate` key is used. For a node, this is *prior* to the setup of the node’s coordinate system and, thus, usually the picture’s coordinate system.

Animation attribute `:position`

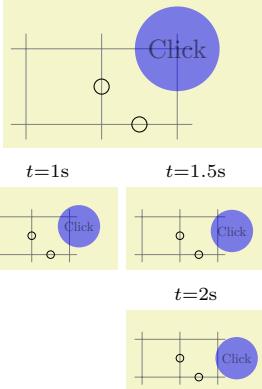
Compare the two animations, one with `:position`, one with `:shift`.



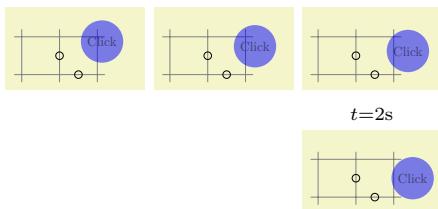
```
\usetikzlibrary {animations}
\tikz {
  \draw [help lines] (-0.2,-0.2) grid (2.2,1.2);
  \draw (1,.5) circle [radius=1mm] (1.5,0) circle [radius=1mm];
  \node :position = { 0s="{{(1,.5)}}", 2s="{{(1.5,0)}}", begin on=click }
    at (1,.5) [fill = blue, opacity=.5, circle] {Click};
}
```



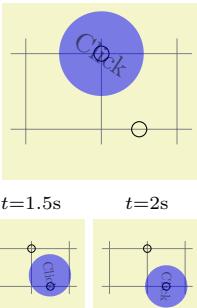
Compare this to a shift:



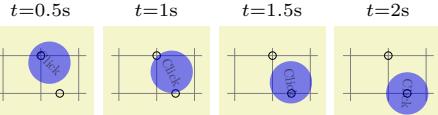
```
\usetikzlibrary {animations}
\tikz {
  \draw [help lines] (-0.2,-0.2) grid (2.2,1.2);
  \draw (1,.5) circle [radius=1mm] (1.5,0) circle [radius=1mm];
  \node :shift = { 0s="{{(1,.5)}}", 2s="{{(1.5,0)}}", begin on=click }
    at (1,.5) [fill = blue, opacity=.5, circle] {Click};
}
```



You can use the `along` key with `:position` in the same way as with `:shift`, which is especially useful for specifying that a node “travels” between positions of the canvas:



```
\usetikzlibrary {animations}
\tikz {
  \draw [help lines] (-0.2,-0.2) grid (2.2,1.2);
  \draw (1,1) circle [radius=1mm] (1.5,0) circle [radius=1mm];
  \node :position = {
    along = {{(1,1)} to[bend left] {(1.5,0)}} sloped in 2s,
    begin on = click }
    at (1,1) [fill = blue, opacity=.5, circle] {Click};
}
```



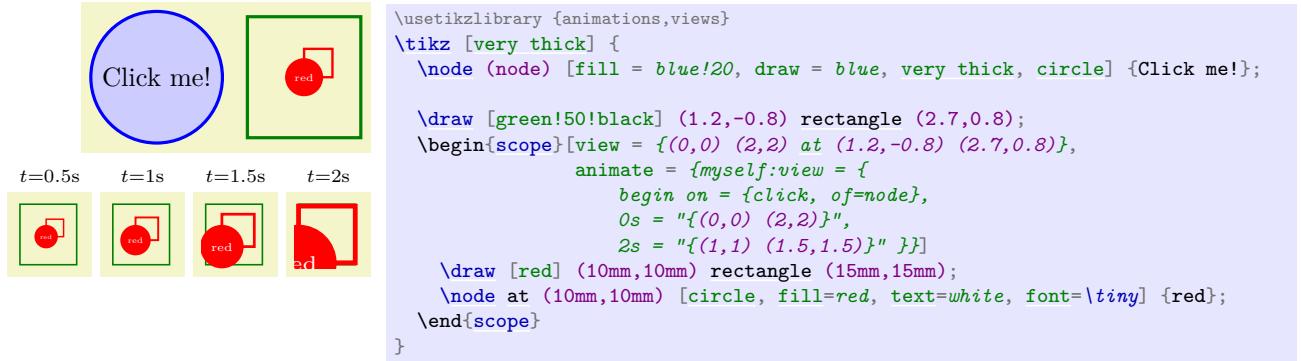
26.4.5 Animating Transformations: Views

The final method of changing the transformation matrix is to animate a *view*.

Animation attribute :view

A view is a canvas transformation that shifts and scales the canvas in such a way that a certain rectangle “matches” another rectangle: The idea is that you “look through” a “window” (the view) and “see” a certain area of the canvas. View animation do not allow you to do anything that cannot also be done using the `shift` and `scale` keys in combination, but it often much more natural to animate which area of a graphic you wish to see than to compute and animate a scaling and shift explicitly.

In order to use a view, you first need to create a view, which is done using the `meet` or `slice` keys from the `views` library, see Section ???. You can then animate the view using the `view` attribute. The values passed to the `entry` key follow the same syntax as the views in the `views` library (though you only animate the to-be-viewed rectangle).



26.5 Controlling the Timeline

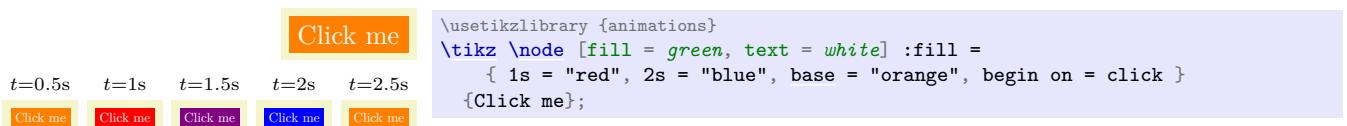
We can already specify timelines by giving a sequence of times in non-decreasing order along with corresponding values. In this section we have a look at further options that allow us to extend or control the timeline.

26.5.1 Before and After the Timeline: Value Filling

When you specify the timeline, you specify it for a certain interval $[t_1, t_2]$. By default, outside this interval the animation has no effect on the to-be-animated attribute. The following keys allows you to change this:

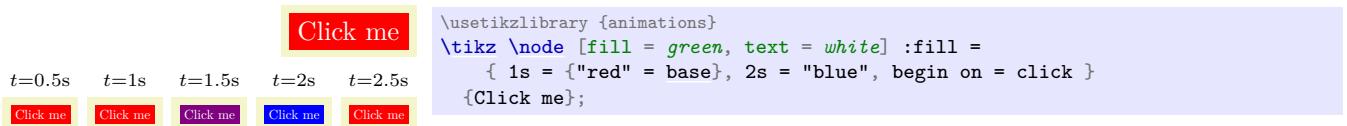
`/tikz/animate/base=<options>` (no default)

A “base” value is a value that is used for the attribute whenever the timeline is *not* active:

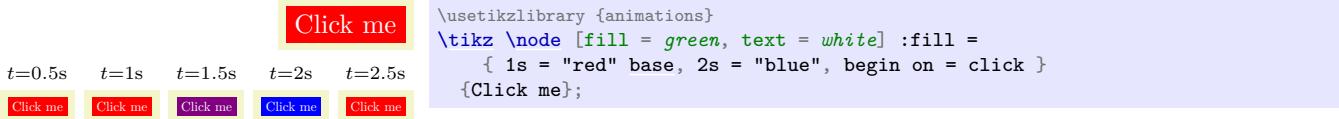


Syntactically, the `base` key works much like special time syntax: It sets up a local `sync` scope and executes the `<options>` in it and creates an `entry`. However, instead of setting the `time` attribute to a time, it sets it to a special value that tells TikZ that when the entry is created, the current `<value>` should be used as the `base` value.

This means that you can write `base = "orange"` as in the above example to set the base. However, you can also use the `base` key in other ways; most noticeably, you can use it *after* some value:

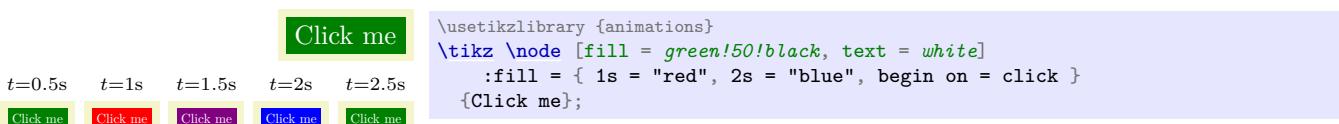
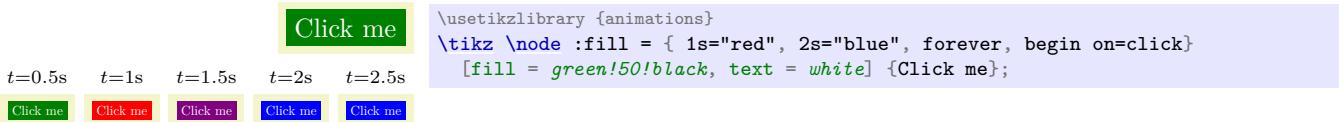


Instead of using `base` as a key, you can also add `base` directly after the quotes of a value. This is particularly useful for setting up a base value that is also used in a timeline:



/tikz/animate/options/forever (no value)

This key causes the timeline to continue “forever” after the last time with the last value. You can also think of this as having the animation “freeze” at the end.



/tikz/animate/options/freeze (no value)

An alias for `forever`.

26.5.2 Beginning and Ending Timelines

The `<time>` used with the first use of the `entry` key in a timeline is the start time and the `<time>` in the last `entry` key is the stop time. However, this leaves open the question of when the whole timeline is to be started: The moment the document is opened? When the page is displayed? When the user scrolls to the to-be-animated object? When some other object is clicked? The key `begin`, and also the key `end`, allow you to specify answers to these questions.

/tikz/animate/options/begin=<time> (no default)

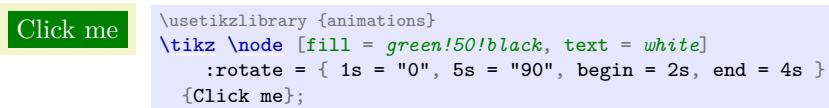
This key specifies when the “moment `0s`” should be relative to the moment when the current graphic is first displayed. You can use this key multiple times, in which case the timeline is restarted for each of the times specified (if it is already running, it will be reset). If no `begin` key is given at all, the effect is the same as if `begin=0s` had been specified.

It is permissible to set `<time>` to a negative value.

Note that this key has no effect for snapshots.

/tikz/animate/options/end=<time> (no default)

This key will truncate the timeline so that it ends `<time>` after the display of the graphic, provided the timeline begins before the specified end time. For instance, if you specify a timeline starting at `2s` and ending at `5s` and you set `begin` to `1s` and `end` to `4s`, the timeline will run, relative to the moment when the graphic is displayed from `3s` to `4s`.



Instead of specifying the beginning of the timeline relative to the moment the to-be-animated graphic is displayed, you can also set the “moment `0s`” to the moment a specific `event` happens using the following key:

/tikz/animate/options/begin on=<options> (no default)

The `<options>` will be executed with the path `/pgf/animation/events` and will cause a new beginning to be added to the list of possible beginnings for the timeline (so the uses of this key accumulate). Each “beginning” is just another possible “moment `0s`” for the timeline. For instance, when the `<options>` are set to `click`, then each time the graph is clicked a moment `0s` starts for the timeline.

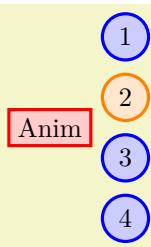
Most events are “caused” or “happen to” some object. For instance, the `click` event happens when you click on a certain object. In order to specify this object, use the following two keys inside the `<options>`: `of` and `of next`. If neither of these keys are given, the to-be-animated object is used.

/pgf/animation/events/of=<id>.<type>

(no default)

This specifies a graphic object id in the same way as the `whom` key, also with an optional `<type>`. This is the object that “causes” the event to happen.

Unlike the `whom` key, which always refers to a not-yet-existing object, this key always refers to an already existing object, namely to the most recent use of the `<id>`. In the following example, the referenced object is the node with the label 2 since it is the most recently referenced node with `<id>` X.

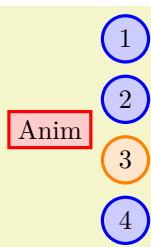


```
\usetikzlibrary {animations}
\begin{tikzpicture} [very thick]
    \node (X) at (1,1.2) [fill = blue!20, draw = blue, circle] {1};
    \node (X) at (1,0.4) [fill = orange!20, draw = orange, circle] {2};
    \node (node) :rotate = {0s="0", 2s="90", begin on = {click, of = X}}
        [fill = red!20, draw = red, rectangle] {Anim};
    \node (X) at (1,-0.4) [fill = blue!20, draw = blue, circle] {3};
    \node (X) at (1,-1.2) [fill = blue!20, draw = blue, circle] {4};
\end{tikzpicture}
```

/pgf/animation/events/of next=<id>.<type>

(no default)

This key works like the `of` key, only it refers to a future (actually, the next) object with the given `<id>`, not to a previous one. This, in the next example, the referenced node is the one with label 3.



```
\usetikzlibrary {animations}
\begin{tikzpicture} [very thick]
    \node (X) at (1,1.2) [fill = blue!20, draw = blue, circle] {1};
    \node (X) at (1,0.4) [fill = blue!20, draw = blue, circle] {2};
    \node (node) :rotate = {
        0s="0", 2s="90", begin on = {click, of next = X}
        [fill = red!20, draw = red, rectangle] {Anim};
    }
    \node (X) at (1,-0.4) [fill = orange!20, draw = orange, circle] {3};
    \node (X) at (1,-1.2) [fill = blue!20, draw = blue, circle] {4};
\end{tikzpicture}
```

The following key allows you to specify the event that should cause the animation to start:

/pgf/animation/events/event=<event name>

(no default)

Specifies the name of the event whose occurrence should start the timeline. Which events are supported depends on the device on which the animation is displayed, the output format (SVG or some other format), and the setup of scripts, but here is a list of events supported by “plain SVG”: `click`, `focusin`, `focusout`, `mousedown`, `mouseup`, `mouseover`, `mousemove`, `mouseout`, `begin`, `end`. However, the following keys make using these events simpler:

/pgf/animate/events/click

(no value)

This is a shorthand for `event=click`. This event gets triggered when the user clicks on the triggering object with a mouse (or something equivalent).



```
\usetikzlibrary {animations}
\begin{tikzpicture}
    \node [fill = blue!20, draw = blue, circle, ultra thick] {Here!};
\end{tikzpicture}
```

/pgf/animation/events/mouse down

(no value)

Shorthand for `event=mousedown`. The event gets triggered when the user presses a mouse button down on the object.



```
\usetikzlibrary {animations}
\begin{tikzpicture}
    \node [fill = blue!20, draw = blue, circle, ultra thick] {Here!};
\end{tikzpicture}
```

/pgf/animation/events/mouse up

(no value)

Shorthand for `event=mouseup` and gets triggered, of course, when a pressed button is released on the object.



```
\usetikzlibrary {animations}
\tikz \node :rotate = { 0s="0", 2s="90", begin on = {mouse up} }
[fill = blue!20, draw = blue, circle, ultra thick] {Here!};
```

/pgf/animation/events/mouse over

(no value)

Shorthand for `event=mouseover`. The event gets triggered the moment the mouse cursor moves over the object.



```
\usetikzlibrary {animations}
\tikz \node :rotate = { 0s="0", 2s="90", begin on = {mouse over} }
[fill = blue!20, draw = blue, circle, ultra thick] {Here!};
```

/pgf/animation/events/mouse move

(no value)

Shorthand for `event=mousmove`. The event gets triggered lots of times, namely each time the mouse moves while being “over” the object.



```
\usetikzlibrary {animations}
\tikz \node :rotate = { 0s="0", 2s="90", begin on = {mouse move} }
[fill = blue!20, draw = blue, circle, ultra thick] {Here!};
```

/pgf/animation/events/mouse out

(no value)

Shorthand for `event=mouseout`. The opposite of `mouse over`: triggered when the mouse leaves the object.



```
\usetikzlibrary {animations}
\tikz \node :rotate = { 0s="0", 2s="90", begin on = {mouse out} }
[fill = blue!20, draw = blue, circle, ultra thick] {Here!};
```

/pgf/animation/events/begin

(no value)

Shorthand for `event=begin`. The “begin” refers to the beginning of another animation, namely the one referenced by `of` or `of whom`. This means that the current animation will begin when some other animation begins.



```
\usetikzlibrary {animations}
\tikz \node [animate = {
myself:rotate = { 0s="0", 2s="90", begin on = {begin, of next=anim} },
myself:xshift = { 0s="0mm", 2s="5mm", begin on = {click}, name=anim }
},
fill = blue!20, draw = blue, circle, ultra thick] {Here!};
```

/pgf/animation/events/end

(no value)

Shorthand for `event=end`. Again, the “end” refers to the end of another animation, namely the one referenced by `of` or `of whom`. This means that the current animation will *begin* when some other animation *ends*.



```
\usetikzlibrary {animations}
\tikz \node [animate = {
myself:rotate = { 0s="0", 2s="90", begin on = {end, of next=anim} },
myself:xshift = { 0s="0mm", 2s="5mm", begin on = {click}, name=anim }
},
fill = blue!20, draw = blue, circle, ultra thick] {Here!};
```

`/pgf/animation/events/focus in`

(no value)

This is a shorthand for `event=focusin`. This event gets triggered when the graphic object gets the focus (this usually makes sense only for text input fields).

`/pgf/animation/events/focus out`

(no value)

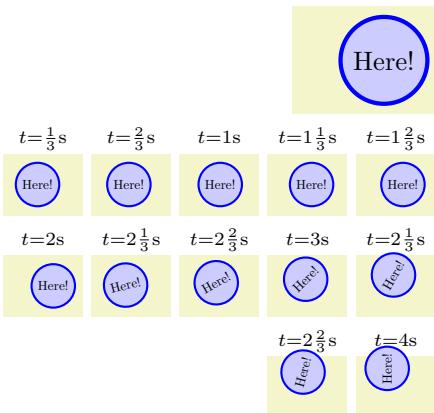
This is a shorthand for `event=focusout`.

In addition to the events specified using the generic `event` key, there are two further events that take a parameter:

`/pgf/animation/events/repeat=<number>`

(no default)

The event is triggered when a repeating animation has been repeated `<number>` times.



```
\usetikzlibrary {animations}
\begin{tikzpicture}
\node [animate = { myself: = {
    :rotate = { 0s="0", 2s="90", begin on = {repeat = 2, of next = anim },
    begin snapshot = 2 },
    :xshift = { 0s="0mm", 2s="5mm", begin on=click, name=anim, repeats=4 }},
    fill = blue!20, draw = blue, circle, ultra thick}] {Here!};

```

`/pgf/animation/events/key=<key>`

(no default)

The event is triggered when the keyboard key `<key>` has been pressed. For security reasons, a viewer may suppress this.

Having specified the event, you can also specify a delay relative to this event:

`/pgf/animation/events/delay=<time>`

(no default)

Specifies that the timeline should not start with the event, but, rather, be delayed by `<time>`.

When you use `begin on` to start an animation when a certain event is triggered, it is not clear what should happen when the event is triggered *again*. Should this be ignored completely? Should it only be ignored while the animation is running? The following key allows you to specify when should happen:

`/tikz/animate/options/restart=<choice>`

(default `true`)

You can set `<choice>` to one of the following:

- `true` means that the animation will restart each time the event is triggered. If the animation is already running, it will be reset to its beginning.
- `false` means that once the animation has started once, it will never be restarted.



```
\usetikzlibrary {animations}
\begin{tikzpicture}
\node [node :rotate = { 0s="0", 2s="90",
    restart = false, begin on = {click}}]
    [fill = blue!20, draw = blue, circle, ultra thick] {Here!};

```

- `never` means the same as `false`.
- `when not active` means that the animation will restart when the event is triggered, but *not* while the animation is running.



```
\usetikzlibrary {animations}
\begin{tikzpicture}
\node [node :rotate = { 0s="0", 2s="90",
    restart = when not active, begin on = {click}}]
    [fill = blue!20, draw = blue, circle, ultra thick] {Here!};

```

Just like `begin on` specifies when a timeline begins relative to some event, the `end on` allows you to stop it early when some event happens:

`/tikz/animate/options/end on=<options>`

(no default)

Works exactly like `begin on`, one possible end of the timeline is specified using the `<options>`.

26.5.3 Repeating Timelines and Accumulation

`/tikz/animate/options/repeats=<specification>`

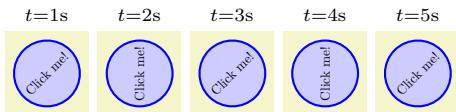
(no default)

Use this key to specify that the timeline animation should repeat at the end. The `<specification>` must consist of two parts, each of which may be empty. The first part is one of the following:

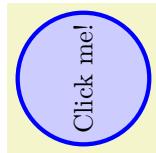
- Empty, in which case the timeline repeats forever.



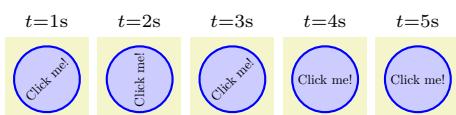
```
\usetikzlibrary {animations}
\tikz \node :rotate = { 0s = "0", 2s = "90",
                        repeats, begin on = click }
[fill = blue!20, draw = blue, ultra thick, circle] {Click me!};
```



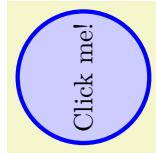
- A `<number>` (like 2 or 3.25), in which case the timeline repeats `<number>` times.



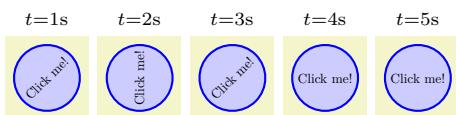
```
\usetikzlibrary {animations}
\tikz \node :rotate = { 0s = "0", 2s = "90",
                        repeats = 1.75, begin on = click }
[fill = blue!20, draw = blue, ultra thick, circle] {Click me!};
```



- The text “`for <time>`” (like `for 2s` or `for 300ms`), in which case the timeline repeats however often necessary so that it stops exactly after `<time>`.



```
\usetikzlibrary {animations}
\tikz \node :rotate = { 0s = "0", 2s = "90",
                        repeats = for 3.5s, begin on = click }
[fill = blue!20, draw = blue, ultra thick, circle] {Click me!};
```

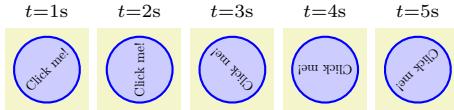


The second part of the specification must be one of the following:

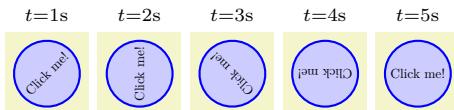
- Empty, in which case each time the timeline is restarted, the attribute’s value undergoes the same series of values it did previously.
- The text `accumulating`. This has the effect that each time the timeline is restarted, the last values specified by the timeline are *added* to the value from the previous iteration(s). A typical example is an animation that shifts a scope by, say, 1 cm over a time of 1 s. Now, if you repeat this five times, normally the scope will shift 1 cm for 1 s then “jump back”, shift again, jump back, and so on for five times. In contrast, when the repeats are accumulating, the scope will move by 5 cm over 5 s in total.



```
\usetikzlibrary {animations}
\tikz \node :rotate = { 0s = "0", 2s = "90", begin on = click,
repeats = accumulating }
[fill = blue!20, draw = blue, ultra thick, circle] {Click me!};
```



```
\usetikzlibrary {animations}
\tikz \node :rotate = { 0s = "0", 2s = "90", begin on = click,
repeats = for 4s accumulating }
[fill = blue!20, draw = blue, ultra thick, circle] {Click me!};
```



`/tikz/animate/options/repeat=<specification>`

(no default)

An alias for `repeats`.

26.5.4 Smoothing and Jumping Timelines

Your specification of the timeline will consist of a sequence of times along with values that the attribute should have at these “key times”. Between these key times, the attribute’s value needs to be interpolated.

Suppose that an animation is supposed to interpolate a attribute’s value between the two values 50 and 100 over a time of 10 s. The simplest way of doing so is to do a linear interpolation, where the value at, say, 1 s is 55, at 2 s it is 60, and so on. Unfortunately, the linear interpolation does not “look” nice in many cases since the acceleration of a linear interpolation is zero during the animation, but infinite at the beginning and at the end; which looks “jerky”.

To avoid this, you can specify that the time–attribute curve should not be a straight line, but rather a curve. You specify this curve using a spline. The most logical “coordinate rectangle” used for this spline in our example would be (0s,50) and (10s,100) and we would like to specify something like

```
(0s,50) .. controls (5s,50) and (9s,100) .. (10s,100)
```

This would result in a time–attribute curve where the attribute at 50 changes slowly at 0 s and also arrives slowly at 100 at 10 s, but speeds up between these values.

We call the first control point (5s,50) the “exit control” and call (9s,100) the “entry control”: The first control dictates how quickly or slowly a time point is left, the second dictates how quickly or slowly we enter the next one.

The control points are, however, not specified in the coordinate system indicated above. Rather, the rectangle (0s,50) to (10s, 100) gets normalized to (0,0) to (1,1). The control point (5s,50) would thus become (0.5,0) and (9s,100) becomes (0.9,1).

`/tikz/animate/options/exit control={<time fraction>}-{<value fraction>}`

(no default)

Specifies an exit control using two values as above. The spline from above would be specified as follows:

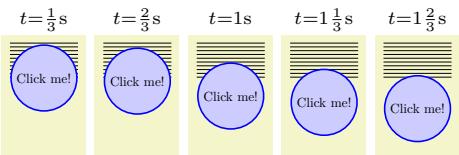
```
exit control={0.5}{0},
entry control={0.9}{1},
0s = "50",
10s = "100"
```

Note that the curve specified using exit and entry controls must be “well-behaved” in the sense that exactly one value must be specified for each point in time in the time interval.

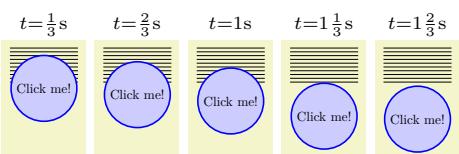
In the next three example, we first specify a “smooth” exit from the start position, then a smooth arrival at the end position, and, finally both.



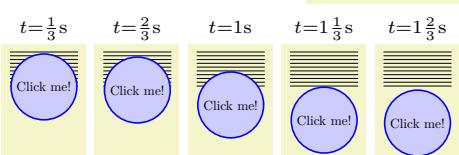
```
\usetikzlibrary {animations}
\tikz {
  \foreach \i in {0,0.1,...,1} \draw (-0.9,.9-\i) -- ++(1.8,0);
  \node [yshift = { begin on = click,
    0s = { exit control = {1}{0}, "0cm" },
    1s = "-5mm",
    2s = "-10mm" } ]
    [fill = blue!20, draw = blue, very thick, circle] {Click me!};
}
```



```
\usetikzlibrary {animations}
\tikz {
  \foreach \i in {0,0.1,...,1} \draw (-0.9,.9-\i) -- ++(1.8,0);
  \node [yshift = { begin on = click,
    0s = "0cm",
    1s = "-5mm",
    2s = { entry control = {0}{1}, "-10mm" } }]
    [fill = blue!20, draw = blue, very thick, circle] {Click me!};
```



```
\usetikzlibrary {animations}
\tikz {
  \foreach \i in {0,0.1,...,1} \draw (-0.9,.9-\i) -- ++(1.8,0);
  \node [yshift = { begin on = click,
    0s = { exit control = {1}{0}, "0cm" },
    1s = "-5mm",
    2s = { entry control = {0}{1}, "-10mm" } }]
    [fill = blue!20, draw = blue, very thick, circle] {Click me!};
```



/tikz/animate/options/entry control={<time fraction>}-{<value fraction>} (no default)

Works like `exit control`.

/tikz/animate/options/ease in={<fraction>} (default 0.5)

A shorthand for `entry control={1-<fraction>}{1}`.

/tikz/animate/options/ease out={<fraction>} (default 0.5)

A shorthand for `exit control={<fraction>}{1}`.

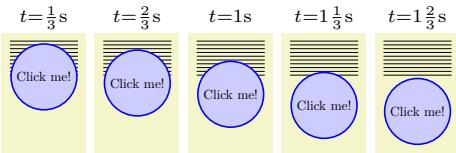
/tikz/animate/options/ease={<fraction>} (default 0.5)

A shorthand for `ease in={<fraction>}, ease out={<fraction>}`.

Note that since for the first time the entry control is ignored and, similarly, for the last time the exit control is ignored, using the `ease` key with an animation having only two times is particularly easy, since we only need to set `ease` once:



```
\usetikzlibrary {animations}
\begin{tikzpicture}
\foreach \i in {0,0.1,...,1} \draw (-0.9,.9-\i) -- ++(1.8,0);
\node [yshift = { begin on = click, ease, 0s = "0cm", 2s = "-10mm" }]
[fill = blue!20, draw = blue, very thick, circle] {Click me!};
\end{tikzpicture}
```



The opposite of having a smooth curve between two values, is to have a “jump” from one value to the next. There are two keys for this:

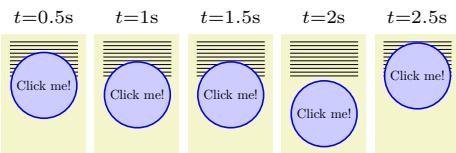
`/tikz/animate/options/stay`

(no value)

Specifies that inside the time interval the value “stays put” at the first value till the end of the interval, where it will jump to the second value. This is similar to an exit control where the curve is “infinitely flat”.



```
\usetikzlibrary {animations}
\begin{tikzpicture}
\foreach \i in {0,0.1,...,1} \draw (-0.9,.9-\i) -- ++(1.8,0);
\node [yshift = { begin on = click,
0s = "0cm",
1s = {stay, "-5mm"}, 
2s = "-10mm" }]
[fill = blue!20, draw = blue, very thick, circle] {Click me!};
\end{tikzpicture}
```



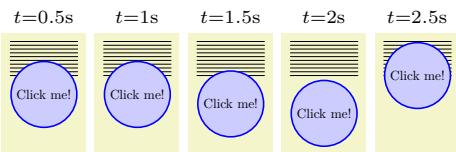
`/tikz/animate/options/jump`

(no value)

Works like the `stay` key, but will cause the value to “jump to” the new value right at the beginning of the time interval. It is similar to an entry control specifying a “flat” curve.



```
\usetikzlibrary {animations}
\begin{tikzpicture}
\foreach \i in {0,0.1,...,1} \draw (-0.9,.9-\i) -- ++(1.8,0);
\node [yshift = { begin on = click,
0s = "0cm",
1s = {jump, "-5mm"}, 
2s = "-10mm" }]
[fill = blue!20, draw = blue, very thick, circle] {Click me!};
\end{tikzpicture}
```



26.6 Snapshots

Snapshots are a way of taking a “photographic snapshot” of an animation at a certain time and then insert these into PDF files (or, for that matter, Postscript files or files in any other format, including SVG): You specify a time like `2s` and then TikZ will compute what the animation “would look like after 2s” and insert the necessary graphics command for rendering the graphic objects in the correct way. Since this computation is done by TikZ and since only “normal” graphics command are inserted into the output, snapshots work with all output formats.

Apart from providing a fallback for PDF, snapshots are very useful by themselves: They make it easy to “show” how an animation unfolds on paper. For this, you simply typeset the same picture with the same animation several times (using a simple `\foreach` loop), but each time you set a different snapshot time. This will result in a sequence of pictures that depict the animation at different points in time and which can then be inserted alongside each other into the printed document. This approach has been used with the examples of animations in this manual.



```
\usetikzlibrary {animations}
\foreach \t in {0.5, 1, 1.5, 2}
  \tikz [make snapshot of = \t]
    \fill :fill = {0s="black", 2s="red"} (0,0) circle [radius = 5mm];
```

Creating snapshots is done using the following key:

`/tikz/make snapshot of=<time>`

(no default)

When this key is used in a `\TeX` scope, animation commands given in the scope do not add animation code to the output. Instead, TikZ computes the values the attributes of the animation would have at the specified `<time>` and inserts the necessary system layer command to set the attribute to the computed values (some care has been taken to make this computation match the computations done by viewer applications as best as possible).



```
\usetikzlibrary {animations}
\tikz [make snapshot of = 1s] {
  \fill :fill = { 0s = "black", 2s = "white" } (0,0) rectangle ++(1,1);
  \fill :fill = { 1s = "black", 3s = "white" } (2,0) rectangle ++(1,1);
}
```

The moment `<time>` is best thought of as `<time>` seconds after the “moment zero” where all timelines start by default. Now, “real” animation may start at different time through user interaction, which clearly makes no sense for snapshots. Nevertheless, you will sometimes wish to have more control over when a timeline starts for the purposes of taking snapshots. You can use the following key for this:

`/tikz/animate/options/begin snapshot=<start time>`

(no default)

Use this key on a timeline to specify that, only for purposes of taking snapshots, the timeline starts at `<start time>` rather than at “moment zero”. (Think of this as saying that the animation starts when a virtual user clicks on the animation and this click occurs `<start time>` seconds after the general “moment zero”, causing the animation to “lag behind” by this amount of time.) Computationally, for the timeline the `<start time>` is subtracted from the snapshot’s `<time>` when the value needs to be determined:



```
\usetikzlibrary {animations}
\tikz [make snapshot of = 1s] {
  \fill :fill = { 0s = "black", 2s = "white" ,
    begin snapshot = 1s } (0,0) rectangle ++(1,1);
  \fill :fill = { 1s = "black", 3s = "white" } (2,0) rectangle ++(1,1);
}
```

The computations of the values the animation “would have” are done entirely by TikZ, which has the big advantage is that no support from the viewer application or the output format is needed – snapshots work with all output formats, not just with SVG. However, computations done by TikZ are not always very precise and can be slow because of `\TeX`’s limitations. In addition, there are some further limitations when it comes to TikZ’s computation of snapshot values:

- As mentioned above, except for `begin snapshot`, other commands for specifying the beginning or end of a timeline based on user interaction make no sense for timelines: The keys `begin`, `begin on`, `end`, and `end on` are silently ignored.
- The value `current value` for a value is forbidden since this value is almost impossible to compute by TikZ.
- Accumulating repeats of a motion are (currently) not supported, but should not rely on this.

When `<time>` is empty, “snapshot taking” is switched off and animation commands are inserted once more.

/tikz/make snapshot after=(*time*)

(no default)

Works exactly like `make snapshot of`, only the $\langle time \rangle$ is interpreted as $\langle time \rangle + \epsilon$. This only makes a difference at the end of a timeline and when there are two or more values specified for the same time: When there are several values specified for time t , a normal snapshot for time t uses the first value given for the attribute. In contrast, this command would use the last one given. Similarly, when an animation timeline ends at time t , a normal snapshot of time t would use the last value of the timeline, while this key would not apply the animation at all (it has already ended at time $t + \epsilon$).



```
\usetikzlibrary {animations}
\tikz [make snapshot of = 2s]
  \fill :fill = { 0s = "green", 2s = "red" } (0,0) rectangle ++(1,1);
\tikz [make snapshot after = 2s]
  \fill :fill = { 0s = "green", 2s = "red" } (0,0) rectangle ++(1,1);
```

/tikz/make snapshot if necessary=(*time*)

(default 0s)

This key makes a snapshot of $\langle time \rangle$ only when the output format does not provide support for animations; if the output format supports animations (like SVG), then the command has no effect and animations are created normally.

This manual is typeset with the following being set once are for all in preamble:

```
\tikzset{make snapshot if necessary}
```

Because of this setting, in the PDF version of this document, all animations are shown at the value they would have at moment 0s. In contrast, in the SVG version, the animations are created normally.

In both versions, the smaller pictures showing how the animation proceeds over time are created using `make snapshot of` for the indicated times.

Part IV

Graph Drawing

by Till Tantau et al.

Graph drawing algorithms do the tough work of computing a layout of a graph for you. TikZ comes with powerful such algorithms, but you can also implement new algorithms in the Lua programming language.

You need to use LuaTeX to typeset this part of the manual (and, also, to use algorithmic graph drawing).

27 Introduction to Algorithmic Graph Drawing

by Till Tantau

This section of the manual can only be typeset using LuaTeX.

28 Using Graph Drawing in TikZ

by Till Tantau

TikZ Library `graphdrawing`

```
\usetikzlibrary{graphdrawing} % LATEX and plain TEX  
\usetikzlibrary[graphdrawing] % ConTEXt
```

This package provides capabilities for automatic graph drawing. It requires that the document is typeset using LuaT_EX. This package should work with LuaT_EX 0.54 or higher.

This section of the manual can only be typeset using LuaT_EX.

29 Using Graph Drawing in PGF

by Till Tantau

PGF Library `graphdrawing`

```
\usepgflibrary{graphdrawing} % LATEX and plain TEX  
\usepgflibrary[graphdrawing] % ConTEXt
```

This package provides the core support for graph drawing inside PGF. It does so by providing PGF macros for controlling the graph drawing system, but also implements the binding to the graph drawing system (see Section 39 for details on bindings).

This section of the manual can only be typeset using `LuaTEX`.

30 Graph Drawing Layouts: Trees

by Till Tantau

This section of the manual can only be typeset using LuaTeX.

31 Graph Drawing Algorithms: Layered Layouts

by Till Tantau and Jannis Pohlmann

This section of the manual can only be typeset using LuaTeX.

32 Graph Drawing Algorithms: Force-Based Methods

by Till Tantau and Jannis Pohlmann

This section of the manual can only be typeset using LuaTeX.

33 Graph Drawing Algorithms: Circular Layouts

by Till Tantau

This section of the manual can only be typeset using LuaTeX.

34 Graph Drawing Layouts: Phylogenetic Trees

by Sarah Mäusle and Till Tantau

This section of the manual can only be typeset using LuaTeX.

35 Graph Drawing Algorithms: Edge Routing

by Till Tantau

This section of the manual can only be typeset using LuaTeX.

36 The Algorithm Layer

by Till Tantau

This section of the manual can only be typeset using LuaTeX.

37 Writing Graph Drawing Algorithms in C

by Till Tantau This section of the manual can only be typeset using LuaTeX.

38 The Display Layer

by Till Tantau

This section of the manual can only be typeset using LuaTeX.

39 The Binding Layer

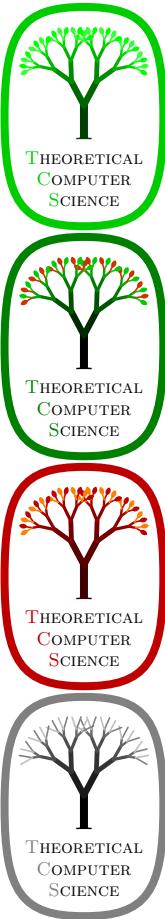
This section of the manual can only be typeset using `LuaTeX`.

Part V

Libraries

by Till Tantau

In this part the library packages are documented. They provide additional predefined graphic objects like new arrow heads or new plot marks, but sometimes also extensions of the basic PGF or TikZ system. The libraries are not loaded by default since many users will not need them.



```
\usepackage{arrows,trees}
\begin{tikzset{
    ld/.style={level distance=#1},lw/.style={line width=#1},
    level 1/.style=[ld=4.5mm,trunk,lw=1ex,sibling angle=60],
    level 2/.style=[ld=3.5mm,trunk!80!leaf a,lw=.8ex,sibling angle=56],
    level 3/.style=[ld=2.75mm,trunk!60!leaf a,lw=.6ex,sibling angle=52],
    level 4/.style=[ld=2mm,trunk!40!leaf a,lw=.4ex,sibling angle=48],
    level 5/.style=[ld=1mm,trunk!20!leaf a,lw=.3ex,sibling angle=44],
    level 6/.style=[ld=1.75mm,leaf a,lw=.2ex,sibling angle=40],
}

\pgfarrowsdeclare{leaf}{leaf}
{\pgfarrowsleftextend{-2pt} \pgfarrowsrightextend{1pt}}
{
    \pgfpathmoveto{\pgfpoint{-2pt}{0pt}}
    \pgfpatharc{150}{30}{1.8pt}
    \pgfpatharc{-30}{-150}{1.8pt}
    \pgfusepathqfill
}

\newcommand{\logo}[5]
{
    \colorlet{border}{#1}
    \colorlet{trunk}{#2}
    \colorlet{leaf a}{#3}
    \colorlet{leaf b}{#4}
    \begin{tikzpicture}[scriptsize,scshape]
        \draw[border, line width=1ex, yshift=.3cm,
            yscale=1.45, xscale=1.05, looseness=1.42]
            (1,0) to [out=90, in=0] (0,1) to [out=180, in=90] (-1,0)
            to [out=-90, in=-180] (0,-1) to [out=0, in=-90] (1,0) -- cycle;

        \coordinate (root) [grow cyclic, rotate=90]
        child {
            child [line cap=round] foreach \a in {0,1} {
                child foreach \b in {0,1} {
                    child foreach \c in {0,1} {
                        child foreach \d in {0,1} {
                            child foreach \leafcolor in {leaf a,leaf b} {
                                edge from parent [color=\leafcolor,-#5]
                            }
                        }
                    }
                }
            }
        edge from parent [shorten >=-1pt, serif cm-, line cap=butt]
    };

    \node [align=center,below] at (0pt,-.5ex)
    { \textcolor{border}{T}heoretical \textcolor{border}{C}omputer \textcolor{border}{S}cience };
    \end{tikzpicture}
}
\begin{minipage}{3cm}
\logo[green!80!black]{green!25!black}{green!80!leaf}\\
\logo[green!50!black]{black}{green!80!black}{red!80!green}{leaf}\\
\logo[red!75!black]{red!25!black}{red!75!black}{orange}{leaf}\\
\logo[black!50]{black}{black!50}{black!25}{}
\end{minipage}
```

40 Three Dimensional Drawing Library

TikZ Library 3d

```
\usetikzlibrary{3d} % LATEX and plain TEX  
\usetikzlibrary[3d] % ConTEX
```

This package provides some styles and options for drawing three dimensional shapes.

40.1 Coordinate Systems

Coordinate system xyz cylindrical

The `xyz cylindrical` coordinate system allows to you specify a point in terms of cylindrical coordinates, sometimes also referred to as cylindrical polar coordinates or polar cylindrical coordinates. It is very similar to the `canvas polar` and `xy polar` coordinate systems with the difference that you provide an elevation over the xy -plane using the `z` key.

`/tikz/cs/angle=(degrees)` (no default, initially 0)

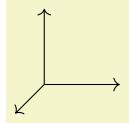
The angle of the coordinate interpreted in the ellipse whose axes are the x -vector and the y -vector.

`/tikz/cs/radius=(number)` (no default, initially 0)

A factor by which the x -vector and y -vector are multiplied prior to forming the ellipse.

`/tikz/cs/z=(number)` (no default, initially 0)

Factor by which the z -vector is multiplied.



```
\usetikzlibrary {3d}  
\begin{tikzpicture} [->]  
  \draw (0,0,0) -- (xyz cylindrical cs:radius=1);  
  \draw (0,0,0) -- (xyz cylindrical cs:radius=1,angle=90);  
  \draw (0,0,0) -- (xyz cylindrical cs:z=1);  
\end{tikzpicture}
```

Coordinate system xyz spherical

The `xyz spherical` coordinate system allows you to specify a point in terms of spherical coordinates.

`/tikz/cs/radius=(number)` (no default, initially 0)

Factor by which the x -, y -, and z -vector are multiplied.

`/tikz/cs/latitude=(degrees)` (no default, initially 0)

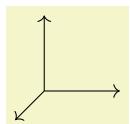
Angle of the coordinate between the y - and z -vector, measured from the y -vector.

`/tikz/cs/longitude=(degrees)` (no default, initially 0)

Angle of the coordinate between the x - and y -vector, measured from the y -vector.

`/tikz/cs/angle=(degrees)` (no default, initially 0)

Same as `longitude`.



```
\usetikzlibrary {3d}  
\begin{tikzpicture} [->]  
  \draw (0,0,0) -- (xyz spherical cs:radius=1);  
  \draw (0,0,0) -- (xyz spherical cs:radius=1,latitude=90);  
  \draw (0,0,0) -- (xyz spherical cs:radius=1,longitude=90);  
\end{tikzpicture}
```

40.2 Coordinate Planes

Sometimes drawing with full three dimensional coordinates is not necessary and it suffices to draw in two dimensions but in a different coordinate plane. The following options help you to switch to a different plane.

40.2.1 Switching to an arbitrary plane

`/tikz/plane origin=<point>` (no default, initially $(0,0)$)

Origin of the plane.

`/tikz/plane x=<point>` (no default, initially $(1,0)$)

Unit vector of the x -direction in the new plane.

`/tikz/plane y=<point>` (no default, initially $(0,1)$)

Unit vector of the y -direction in the new plane.

`/tikz/canvas is plane` (no value)

Perform the transformation into the new canvas plane using the units above. Note that you have to set the units *before* calling `canvas is plane`.



```
\usetikzlibrary {3d}
\begin{tikzpicture}[
->,
plane x={(0.707,-0.707)},
plane y={(0.707,0.707)},
canvas is plane,
]
\draw (0,0) -- (1,0);
\draw (0,0) -- (0,1);
\end{tikzpicture}
```

40.2.2 Predefined planes

`/tikz/canvas is xy plane at z=<dimension>` (no default)

A plane with

- `plane origin={(0,0,<dimension>)}`,
- `plane x={(1,0,<dimension>)}`, and
- `plane y={(0,1,<dimension>)}`.

`/tikz/canvas is yx plane at z=<dimension>` (no default)

A plane with

- `plane origin={(0,0,<dimension>)}`,
- `plane x={(0,1,<dimension>)}`, and
- `plane y={(1,0,<dimension>)}`.

`/tikz/canvas is xz plane at y=<dimension>` (no default)

A plane with

- `plane origin={(0,<dimension>,0)}`,
- `plane x={(1,<dimension>,0)}`, and
- `plane y={(0,<dimension>,1)}`.

`/tikz/canvas is zx plane at y=<dimension>` (no default)

A plane with

- `plane origin={(0,<dimension>,0)}`,
- `plane x={(0,<dimension>,1)}`, and
- `plane y={(1,<dimension>,0)}`.

`/tikz/canvas is yz plane at x=<dimension>` (no default)

A plane with

- `plane origin={(<dimension>,0,0)}`,

- `plane x={⟨dimension⟩,1,0}`, and
- `plane y={⟨dimension⟩,0,1}`.

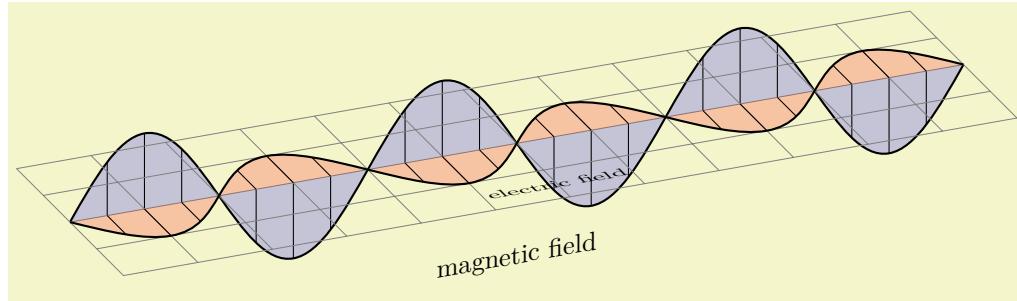
`/tikz/canvas is zy plane at x=⟨dimension⟩`

(no default)

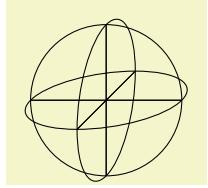
A plane with

- `plane origin={⟨dimension⟩,0,0}`,
- `plane x={⟨dimension⟩,0,1}`, and
- `plane y={⟨dimension⟩,1,0}`.

40.3 Examples



```
\usetikzlibrary {3d}
\begin{tikzpicture} [z={(10:10mm)},x={(-45:5mm)}]
 \def\wave{
  \draw[fill,thick,fill opacity=.2]
  (0,0) sin (1,1) cos (2,0) sin (3,-1) cos (4,0)
  sin (5,1) cos (6,0) sin (7,-1) cos (8,0)
  sin (9,1) cos (10,0)sin (11,-1)cos (12,0);
 \foreach \shift in {0,4,8}
 {
  \begin{scope}[xshift=\shift cm,thin]
  \draw (.5,0) -- (0.5,0 |- 45:1cm);
  \draw (1,0) -- (1,1);
  \draw (1.5,0) -- (1.5,0 |- 45:1cm);
  \draw (2.5,0) -- (2.5,0 |- -45:1cm);
  \draw (3,0) -- (3,-1);
  \draw (3.5,0) -- (3.5,0 |- -45:1cm);
  \end{scope}
 }
 }
 \begin{scope}[canvas is zy plane at x=0,fill=blue]
 \wave
 \node at (6,-1.5) [transform shape] {magnetic field};
 \end{scope}
 \begin{scope}[canvas is zx plane at y=0,fill=red]
 \draw[help lines] (0,-2) grid (12,2);
 \wave
 \node at (6,1.5) [rotate=180,xscale=-1,transform shape] {electric field};
 \end{scope}
\end{tikzpicture}
```



```
\usetikzlibrary {3d}
\begin{tikzpicture}
  \begin{scope}[canvas is zy plane at x=0]
    \draw (0,0) circle (1cm);
    \draw (-1,0) -- (1,0) (0,-1) -- (0,1);
  \end{scope}

  \begin{scope}[canvas is zx plane at y=0]
    \draw (0,0) circle (1cm);
    \draw (-1,0) -- (1,0) (0,-1) -- (0,1);
  \end{scope}

  \begin{scope}[canvas is xy plane at z=0]
    \draw (0,0) circle (1cm);
    \draw (-1,0) -- (1,0) (0,-1) -- (0,1);
  \end{scope}
\end{tikzpicture}
```

41 Angle Library

TikZ Library `angles`

```
\usetikzlibrary{angles} % LATEX and plain TEX
\usetikzlibrary[angles] % ConTeXt
```

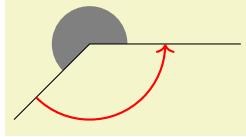
This library defines pic types for drawing angles.

Pic type `angle=<A>--<C>`

This pic adds a drawing of an angle to the current path. This “drawing of an angle” consist of a “sector” or “wedge” or “slice” whose pointed end is at point $\langle B \rangle$ and whose straight sides lie on the lines from $\langle B \rangle$ to $\langle A \rangle$ and from $\langle B \rangle$ to $\langle C \rangle$. The length of these lines is governed by the following key:

`/tikz/angle radius=<dimension>` (no default, initially 5mm)

The length of the sides of the angle’s wedge:

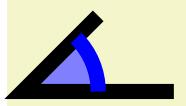


```
\usetikzlibrary {angles}
\tikz \draw (2,0) coordinate (A) -- (0,0) coordinate (B)
          -- (-1,-1) coordinate (C)
          pic [fill=black!50] {angle = A--B--C}
          pic [draw,>,red,thick,angle radius=1cm] {angle = C--B--A};
```

The three points $\langle A \rangle$, $\langle B \rangle$, and $\langle C \rangle$ *must* be the names of nodes or coordinates; you cannot use direct coordinates like “(1,1)” here.

You can leave out the three points, in this case the text `A--B--C` is used; so in the above examples we could just have written `{angle}` in the first pic.

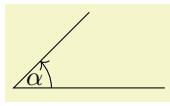
Concerning the sector that makes up the drawing of the angle, the angular part of this sector is drawn in front of the path if the `draw` option is given to the `pic`, while filled sector is drawn behind the `pic`, provided an option like `fill` or `shade` is passed to the `pic`. The following example shows the difference:



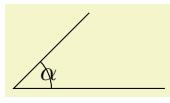
```
\usetikzlibrary {angles}
\tikz \draw [line width=2mm]
      (2,0) coordinate (A) -- (0,0) coordinate (B)
      -- (1,1) coordinate (C)
      pic [draw=blue, fill=blue!50, angle radius=1cm] {angle};
```

When `pic text` is set (which you typically do by using the quotes syntax), a node will be created whose name is empty (and, thus, inherits the `pic`’s name) and which will be at the half-way angle between the lines to $\langle A \rangle$ and $\langle C \rangle$ and whose distance from $\langle B \rangle$ is `angle radius` times the following factor:

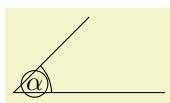
`/tikz/angle eccentricity=<factor>` (no default, initially 0.6)



```
\usetikzlibrary {angles,quotes}
\tikz \draw (2,0) coordinate (A) -- (0,0) coordinate (B)
          -- (1,1) coordinate (C)
          pic ["$\alpha$", draw, ->] {angle};
```



```
\usetikzlibrary {angles,quotes}
\tikz \draw (2,0) coordinate (A) -- (0,0) coordinate (B)
          -- (1,1) coordinate (C)
          pic ["$\alpha$", draw, angle eccentricity=1] {angle};
```

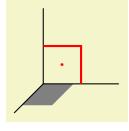


```
\usetikzlibrary {angles,quotes}
\tikz {
  \draw (2,0) coordinate (A) -- (0,0) coordinate (B)
  -- (1,1) coordinate (C)
  pic (alpha) ["$\alpha$", draw] {angle};

  \draw (alpha) circle [radius=5pt];
}
```

Pic type `right angle=<A>--<C>`

This pic adds a drawing of a right angle to the current path. It works in the same way as `angle` pic.



```
\usetikzlibrary {angles}
\begin{tikzpicture}
\draw (0,0,0) coordinate (O)
(1,0,0) coordinate (A) -- (0)
(0,0,1) coordinate (B) -- (0)
(0,1,0) coordinate (C) -- (0)
\pic [fill=gray,angle radius=4mm] {right angle = A--O--B};
\pic [draw,red,thick,angle eccentricity=.5,pic text=.] {right angle = A--O--C};
\end{tikzpicture}
```

42 Arrow Tip Library

The libraries `arrows` and `arrows.spaced` from older versions of PGF are still available for compatibility, but they are considered deprecated.

The standard arrow tips, which are loaded by the library `arrows.meta`, are documented in Section 16.5.

43 Automata Drawing Library

TikZ Library `automata`

```
\usetikzlibrary{automata} % LATEX and plain TEX
\usetikzlibrary[automata] % ConTEX
```

This packages provides shapes and styles for drawing finite state automata and Turing machines.

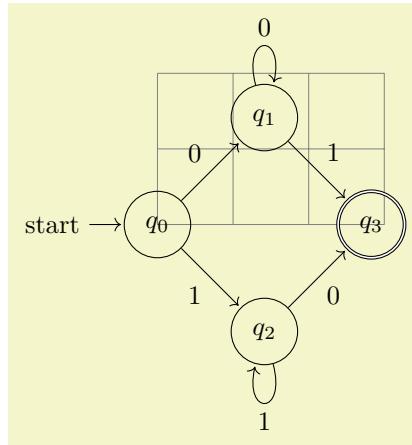
43.1 Drawing Automata

The `automata` (drawing) library is intended to make it easy to draw finite automata and Turing machines. It does not cover every situation imaginable, but most finite automata and Turing machines found in text books can be drawn in a nice and convenient fashion using this library.

To draw an automaton, proceed as follows:

1. For each state of the automaton, there should be one node with the option `state`.
2. To place the states, you can either use absolute positions or relative positions, using options like `above` or `right`.
3. Give a unique name to each state node.
4. Accepting and initial states are indicated by adding the options `accepting` and `initial`, respectively, to the state nodes.
5. Once the states are fixed, the edges can be added. For this, the `edge` operation is most useful. It is, however, also possible to add edges after each node has been placed.
6. For loops, use the `edge [loop]` operation.

Let us now see how this works for a real example. Let us consider a nondeterministic four state automaton that checks whether an input contains the sequence 0^*1 or the sequence 1^*0 .



```
\usetikzlibrary {automata,positioning}
\begin{tikzpicture}[shorten >=1pt,node distance=2cm, on grid, auto]
\draw [help lines] (0,0) grid (3,2);

\node[state,initial] (q_0) {$q_0$};
\node[state] (q_1) [above right=of q_0] {$q_1$};
\node[state] (q_2) [below right=of q_0] {$q_2$};
\node[state,accepting] (q_3) [below right=of q_1] {$q_3$};

\path[->] (q_0) edge node {} (q_1)
          edge node {} (q_2)
          (q_1) edge node {} (q_1)
          edge [swap] node {} (q_3)
          (q_2) edge [loop above] node {} (q_2)
          edge [loop below] node {} (q_3);
\end{tikzpicture}
```

43.2 States With and Without Output

The `state` style actually just “selects” a default underlying style. Thus, you can define multiple new complicated state style and then simply set the `state` style to your given style to get the desired kind of styles.

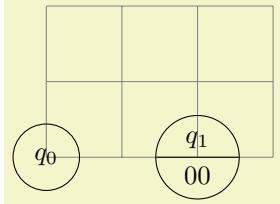
By default, the following state styles are defined:

`/tikz/state without output` (style, no value)

This node style causes nodes to be drawn as circles. Also, this style calls `every state`.

`/tikz/state with output` (style, no value)

This node style causes nodes to be drawn as split circles, that is, using the `circle split` shape. In the upper part of the shape you have the name of the style, in the lower part the output is placed. To specify the output, use the command `\nodepart{lower}` inside the node. This style also calls `every state`.

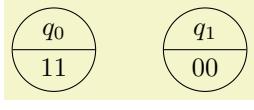


```
\usetikzlibrary {automata}
\begin{tikzpicture}
  \draw[help lines] (0,0) grid (3,2);

  \node[state without output] {$q_0$};
  \node[state with output] at (2,0) {$q_1$};
    \nodepart{lower} 00;
\end{tikzpicture}
```

`/tikz/state` (style, initially `state without output`)

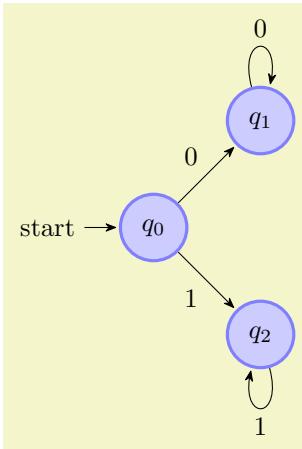
You should redefine it to something else, if you wish to use states of a different nature.



```
\usetikzlibrary {automata}
\begin{tikzpicture}[style=state with output]
  \node[state] {$q_0$};
  \node[state] at (2,0) {$q_1$};
    \nodepart{lower} 00;
\end{tikzpicture}
```

`/tikz/every state` (style, initially empty)

This style is used by `state with output` and also by `state without output`. By default, it does nothing, but you can use it to make your state look more fancy:



```
\usetikzlibrary {arrows.meta,automata,positioning}
\begin{tikzpicture}[shorten >=1pt,node distance=2cm, on grid,>={Stealth[round]},
  every state/.style={draw=blue!50, very thick, fill=blue!20}]
  \node[state,initial] (q_0) {$q_0$};
  \node[state] (q_1) [above right=of q_0] {$q_1$};
  \node[state] (q_2) [below right=of q_0] {$q_2$};

  \path[->] (q_0) edge node [above left] {0} (q_1)
    edge node [below left] {1} (q_2)
    (q_1) edge [loop above] node {} ();
  \path[->] (q_1) edge node [above left] {0} (q_1)
    (q_2) edge [loop below] node {} ();
\end{tikzpicture}
```

43.3 Initial and Accepting States

The styles `initial` and `accepting` are similar to the `state` style as they also just select an “underlying” style, which installs the actual settings for initial and accepting states.

Let us start with the initial states.

`/tikz/initial` (style, initially `initial by arrow`)

This style is used to draw initial states.

`/tikz/initial by arrow` (style, no value)

This style causes an arrow and, possibly, some text to be added to the node. The arrow points from the text to the node. The node text and the direction and the distance can be set using the following key:

`/tikz/initial text=<text>` (no default, initially `start`)

This key sets the text to be used. Use an empty text to suppress all text.

`/tikz/initial where=<direction>` (no default, initially `left`)

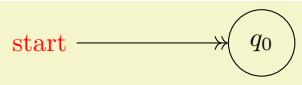
Set the place where the text should be shown. Allowed values are `above`, `below`, `left`, and `right`.

`/tikz/initial distance=<distance>` (no default, initially `3ex`)

Sets the length of the arrow leading from the text to the state node.

`/tikz/every initial by arrow` (style, initially empty)

This style is executed at the beginning of every path that contains the arrow and the text. You can use it to, say, make the text red or whatever.



```
\usetikzlibrary {automata}
\begin{tikzpicture} [every initial by arrow/.style={text=red,->}]
    \node[state,initial,initial distance=2cm] {$q_0$};
\end{tikzpicture}
```

`/tikz/initial above` (style, no value)

This is a shorthand for `initial by arrow,initial where=above`.

`/tikz/initial below` (style, no value)

Works similarly to the previous option.

`/tikz/initial left` (style, no value)

Works similarly to the previous option.

`/tikz/initial right` (style, no value)

Works similarly to the previous option.

`/tikz/initial by diamond` (style, no value)

This style uses a diamond to indicate an initial node.

For the accepting states, the situation is similar: There is also an `accepting` style that selects the way accepting states are rendered. There are now two options: First, `accepting by arrow`, which works the same way as `initial by arrow`, only with the direction of arrow reversed, and `accepting by double`, where accepting states get a double line around them.

`/tikz/accepting` (style, initially `accepting by double`)

This style is used to draw accepting states. You can replace this by the style `accepting by arrow` to get accepting states with an arrow leaving them.

`/tikz/accepting by double` (style, no value)

This style causes a double line to be drawn around a state.

`/tikz/accepting by arrow` (style, no value)

This style causes an arrow and, possibly, some text to be added to the node. The arrow points to the text from the node.

The same options as for initial states can be used, only with `initial` replaced by `accepting`:

`/tikz/accepting text=<text>` (no default, initially empty)

This key sets the text to be used.

`/tikz/accepting where=<direction>` (no default, initially `right`)

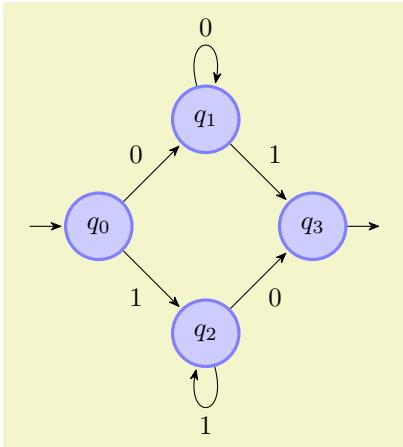
Set the place where the text should be shown. Allowed values are `above`, `below`, `left`, and `right`.

`/tikz/initial distance=<distance>` (no default, initially `3ex`)

Sets the length of the arrow leading from the text to the state node.

`/tikz/every accepting by arrow` (style, initially empty)

Executed at the beginning of every path that contains the arrow and the text.



```
\usetikzlibrary {arrows.meta,automata,positioning}
\begin{tikzpicture} [shorten >=1pt,node distance=2cm, on grid,>={Stealth[round]}, initial text=,
  every state/.style={draw=blue!50, very thick, fill=blue!20},
  accepting/.style=accepting by arrow]

\node[state,initial] (q_0) {$q_0$};
\node[state] (q_1) [above right=of q_0] {$q_1$};
\node[state] (q_2) [below right=of q_0] {$q_2$};
\node[state,accepting] (q_3) [below right=of q_1] {$q_3$};

\path[->] (q_0) edge node [above left] {0} (q_1)
            edge node [below left] {1} (q_2)
            (q_1) edge node [above right] {1} (q_3)
            edge [loop above] node {} ();
            (q_2) edge node [below right] {0} (q_3)
            edge [loop below] node {} ();

\end{tikzpicture}
```

`/tikz/accepting above` (style, no value)

This is a shorthand for `accepting by arrow, accepting where=above`.

`/tikz/accepting below` (style, no value)

Works similarly to the previous option.

`/tikz/accepting left` (style, no value)

Works similarly to the previous option.

`/tikz/accepting right` (style, no value)

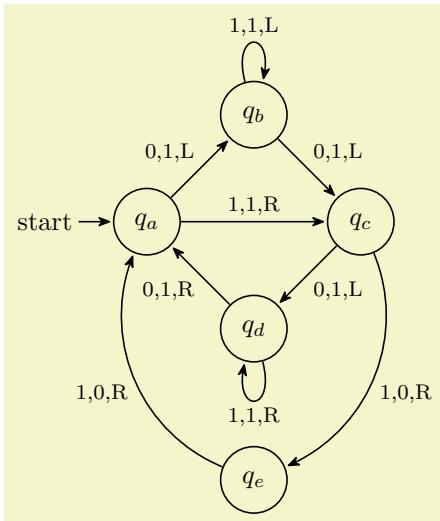
Works similarly to the previous option.

43.4 Examples

In the following example, we once more typeset the automaton presented in the previous sections. This time, we use the following rule for accepting/initial state: Initial states are red, accepting states are green, and normal states are orange. Then, we must find a path from a red state to a green state.



The next example is the current candidate for the five-state busiest beaver:



44 Babel Library

TikZ Library `babel`

```
\usetikzlibrary{babel} % LATEX and plain TEX  
\usetikzlibrary[babel] % ConTEXt
```

A tiny library that make the interaction with the `babel` package easier. Despite the name, it may also be useful in other contexts, namely whenever the catcodes of important symbols are changed globally. Normally, using this library is always a good idea; it is not always loaded by default since in some rare cases it may break old code.

The problems this library tries to fix have to do with the so-called “catcodes” of symbols used inside TikZ. In normal T^EX operation, symbols like ! or " are “normal” characters and the TikZ parser expects them to be. Some packages, most noticeably the `babel` package, aggressively change these character codes so that for instance a semicolon gets a little extra space in `french` mode or a quotation mark followed by a vertical bar breaks ligatures in `german` mode.

Unfortunately, TikZ expects the character codes of some symbols to be “normal”. In some important cases it will tolerate changed character codes, but when the changes made by `babel` (or some other package) are too “aggressive”, compilation of TikZ code will fail.

The `babel` library of TikZ is intended to help out in this situation. All this library does is to set the following two keys to `true`. You can, however, also set these keys directly and also switch them off or on individually and independently of this library.

/tikz/handle active characters in code=⟨true or false⟩ (no default, initially `false`)

When this key is set, at the beginning of every `\tikz` command and every `{tikzpicture}`, the character codes of all symbols used by TikZ are reset to their normal values. Furthermore, at the beginning of each node, the catcodes are restored to the values they had prior to the current picture.

The net effect of this is that, in most cases, symbols having a special character code can be used nicely both in TikZ code and also in node texts.

In the following, slightly silly, example we make the dot an active character and define it in some strange way. Now, in the later TikZ command, the dot in `3.0cm` may no longer be active and setting the `handle...` option achieves exactly this. However, as can be seen, the dot is once more active inside the node.

hallø people	<code>\catcode`\.=\active \def.{\o} \tikz [handle active characters in code] \node [draw, minimum width=3.0cm] {hall. pe.ople};</code>
--------------	---

/tikz/handle active characters in nodes=⟨true or false⟩ (no default, initially `false`)

This key is needed for a special situation: As explained for the `handle ... code` key, that key switches off all special meaning of symbols and switches them back on again at the beginning of nodes. However, there is one situation when this is not possible: When some text has already been read by T^EX, the catcodes can no longer change. Now, for normal nodes this is not a problem since their contents has not been read at the moment the catcodes are restored. In contrast for label nodes for edges, nodes produced by the `graph` and `quotes` libraries, and some others nodes, their text *has* already been read when the catcodes get adjusted.

The present key may help in such situations: It causes the text of all such “indirectly created” nodes to be surrounded by a call to the `\scantokens` command. This command attempts to reread an already read text, but allows catcodes to change. As users of this command will know, it is not a perfect substitute for directly reading the text by T^EX, but it normally has the desired effect.

føø hallø people	<code>\catcode`\.=\active \def.{\o} \tikz [handle active characters in code, handle active characters in nodes] \node [draw, label=f..] {hall. pe.ople};</code>
---------------------	--

45 Background Library

TikZ Library `backgrounds`

```
\usetikzlibrary{backgrounds} % LATEX and plain TEX  
\usetikzlibrary[backgrounds] % ConTEXt
```

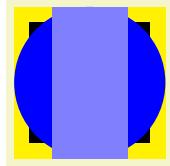
This library defines “backgrounds” for pictures. This does not refer to background pictures, but rather to frames drawn around and behind pictures. For example, this package allows you to just add the `framed` option to a picture to get a rectangular box around your picture or `gridded` to put a grid behind your picture.

The first use of this library is to make the following key available:

`/tikz/on background layer=<options>` (no default)

This key can (only) be used with a `{scope}` or `\scopeds`. It will cause everything inside the scope to be typeset on a background layer.

The `<options>` will be executed *inside* background scope. This is useful since *other* options passed to the `{scope}` environment will be executed *before* the actual background material starts and, thus, will have no effect on it.



```
\usetikzlibrary {backgrounds}  
\begin{tikzpicture}  
  % On main layer:  
  \fill[blue] (0,0) circle (1cm);  
  
  \begin{scope}[on background layer={color=yellow}]  
    \fill (-1,-1) rectangle (1,1);  
  \end{scope}  
  
  \begin{scope}[on background layer]  
    \fill[black] (-.8,-.8) rectangle (.8,.8);  
  \end{scope}  
  
  % On main layer again:  
  \fill[blue!50] (-.5,-1) rectangle (.5,1);  
\end{tikzpicture}
```

A scope with this option set should not be “deeply nested” inside the picture since changes to the graphic state (like the color or the transformation matrix) “do not survive a layer switch”, see also Section ?? for details. In particular, setting, say, the line width at the beginning of a picture will not have an effect on the background picture.

For this reason, it may be useful to setup the following style:

`/tikz/every on background layer` (style, no value)

This style is executed at the beginning of each background layer. If you have a global setup in `every picture`, you should consider putting that part of it that concerns the graphics state into this style.

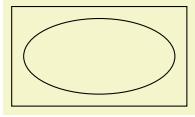


```
\usetikzlibrary {backgrounds}  
\tikzset{  
  every picture/.style={line width=1ex},  
  every on background layer/.style={every picture}  
}  
\begin{tikzpicture}  
  \draw [->] (0,0) -- (2,1);  
  
  \scopeds[on background layer]  
  \draw [red] (0,1) -- (2,0);  
\end{tikzpicture}
```

When this package is loaded, the following styles become available:

`/tikz/show background rectangle` (style, no value)

This style causes a rectangle to be drawn behind your graphic. This style option must be given to the `{tikzpicture}` environment or to the `\tikz` command.



```
\usetikzlibrary {backgrounds}
\begin{tikzpicture}[show background rectangle]
  \draw (0,0) ellipse (10mm and 5mm);
\end{tikzpicture}
```

The size of the background rectangle is determined as follows: We start with the bounding box of the picture. Then, a certain separator distance is added on the sides. This distance can be different for the *x*- and *y*-directions and can be set using the following options:

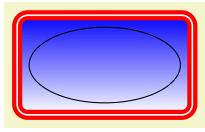
<code>/tikz/inner frame xsep=<dimension></code>	(no default, initially <code>1ex</code>)
Sets the additional horizontal separator distance for the background rectangle.	
<code>/tikz/inner frame ysep=<dimension></code>	(no default, initially <code>1ex</code>)
Same for the vertical separator distance.	
<code>/tikz/inner frame sep=<dimension></code>	(no default)
Sets the horizontal and vertical separator distances simultaneously.	

The following two styles make setting the inner separator a bit easier to remember:

<code>/tikz/tight background</code>	(style, no value)
Sets the inner frame separator to 0pt. The background rectangle will have the size of the bounding box.	
<code>/tikz/loose background</code>	(style, no value)
Sets the inner frame separator to 2ex.	

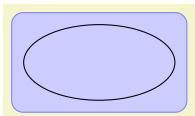
You can influence how the background rectangle is rendered by setting the following style:

<code>/tikz/background rectangle</code>	(style, initially <code>draw</code>)
This style dictates how the background rectangle is drawn or filled. The default setting causes the path of the background rectangle to be drawn in the usual way. Setting this style to, say, <code>fill=blue!20</code> causes a light blue background to be added to the picture. You can also use more fancy settings as shown in the following example:	



```
\usetikzlibrary {backgrounds}
\begin{tikzpicture}
  [background rectangle/.style=
   {double, ultra thick, draw=red, top color=blue, rounded corners},
   show background rectangle]
  \draw (0,0) ellipse (10mm and 5mm);
\end{tikzpicture}
```

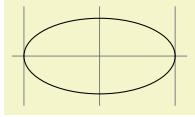
Naturally, no one in their right mind would use the above, but here is a nice background:



```
\usetikzlibrary {backgrounds}
\begin{tikzpicture}
  [background rectangle/.style=
   {draw=blue!50, fill=blue!20, rounded corners=1ex},
   show background rectangle]
  \draw (0,0) ellipse (10mm and 5mm);
\end{tikzpicture}
```

<code>/tikz/framed</code>	(style, no value)
This is a shorthand for <code>show background rectangle</code> .	

<code>/tikz/show background grid</code>	(style, no value)
This style behaves similarly to the <code>show background rectangle</code> style, but it will not use a rectangle path, but a grid. The lower left and upper right corner of the grid is computed in the same way as for the background rectangle:	



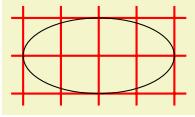
```
\usetikzlibrary {backgrounds}
\begin{tikzpicture}[show background grid]
  \draw (0,0) ellipse (10mm and 5mm);
\end{tikzpicture}
```

You can influence the background grid by setting the following style:

/tikz/background grid

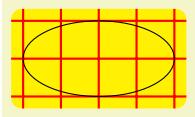
(style, initially `draw,help lines`)

This style dictates how the background grid path is drawn.



```
\usetikzlibrary {backgrounds}
\begin{tikzpicture}
  [background grid/.style={thick,draw=red,step=.5cm},
   show background grid]
  \draw (0,0) ellipse (10mm and 5mm);
\end{tikzpicture}
```

This option can be combined with the `framed` option (use the `framed` option first):



```
\usetikzlibrary {backgrounds}
\tikzset{background grid/.style={thick,draw=red,step=.5cm},
         background rectangle/.style={rounded corners,fill=yellow}}
\begin{tikzpicture}[framed,gridded]
  \draw (0,0) ellipse (10mm and 5mm);
\end{tikzpicture}
```

/tikz/gridded

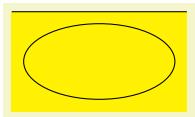
(style, no value)

This is a shorthand for `show background grid`.

/tikz/show background top

(style, no value)

This style causes a single line to be drawn at the top of the background rectangle. Normally, the line coincides exactly with the top line of the background rectangle:



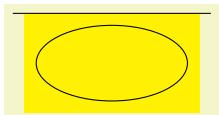
```
\usetikzlibrary {backgrounds}
\begin{tikzpicture}[
  background rectangle/.style={fill=yellow},
  framed,show background top]
  \draw (0,0) ellipse (10mm and 5mm);
\end{tikzpicture}
```

The following option allows you to lengthen (or shorten) the line:

/tikz/outer frame xsep=<dimension>

(no default, initially `0pt`)

The `<dimension>` is added at the left and right side of the line.



```
\usetikzlibrary {backgrounds}
\begin{tikzpicture}[
  background rectangle/.style={fill=yellow},
  framed,
  show background top,
  outer frame xsep=1ex]
  \draw (0,0) ellipse (10mm and 5mm);
\end{tikzpicture}
```

/tikz/outer frame ysep=<dimension>

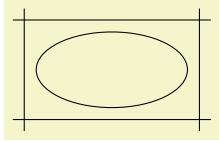
(no default, initially `0pt`)

This option does not apply to the top line, but to the left and right lines, see below.

/tikz/outer frame sep=<dimension>

(no default)

Sets both the *x*- and *y*-separation.

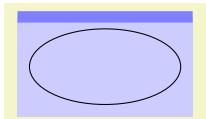


```
\usetikzlibrary {backgrounds}
\begin{tikzpicture}
[background rectangle={fill=blue!20},
 outer frame sep=1ex,%
 show background top,%
 show background bottom,%
 show background left,%
 show background right]
\draw (0,0) ellipse (10mm and 5mm);
\end{tikzpicture}
```

You can influence how the line is drawn grid by setting the following style:

/tikz/background top

(style, initially **draw**)



```
\usetikzlibrary {backgrounds}
\tikzset{background rectangle/.style={fill=blue!20},
         background top/.style={draw=blue!50,line width=1ex}}
\begin{tikzpicture}[framed,show background top]
\draw (0,0) ellipse (10mm and 5mm);
\end{tikzpicture}
```

/tikz/show background bottom

(style, no value)

Works like the style for the top line.

/tikz/show background left

(style, no value)

Works similarly.

/tikz/show background right

(style, no value)

Works similarly.

46 Bounding Boxes for Bézier Curves

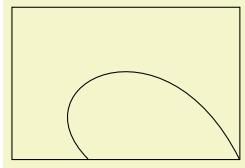
TikZ Library `bbox`

```
\usepgflibrary{bbox} % LATEX and plain TEX and pure pgf
\usepgflibrary[bbox] % ConTeXt and pure pgf
\usetikzlibrary{bbox} % LATEX and plain TEX when using TikZ
\usetikzlibrary[bbox] % ConTeXt when using TikZ
```

This library provides methods to determine tight bounding boxes for Bézier curves.

46.1 Current Status

TikZ determines the bounding box of (cubic) Bézier curves by establishing the smallest rectangle that contains the end point and the two control points of the curve. This may lead to drastic overestimates of the bounding box.



```
\begin{tikzpicture}
  \draw (0,0) .. controls (-1,1) and (1,2) .. (2,0);
  \draw (current bounding box.south west) rectangle
    (current bounding box.north east);
\end{tikzpicture}
```

46.2 Computing the Bounding Box

Establishing the precise bounding box has been discussed in various places, the following discussion uses in part the results from <https://pomax.github.io/bezierinfo/>. What is a cubic Bezier curve? A cubic Bezier curve running from (x_0, y_0) to (x_1, y_1) with control points (x_a, y_a) and (x_b, y_b) can be parametrized by

$$\gamma(t) = \begin{pmatrix} x(t) \\ y(t) \end{pmatrix} = \begin{pmatrix} t^3 x_1 + 3t^2(1-t)x_b + (1-t)^3 x_0 + 3t(1-t)^2 x_a \\ t^3 y_1 + 3t^2(1-t)y_b + (1-t)^3 y_0 + 3t(1-t)^2 y_a \end{pmatrix}, \quad (1)$$

where t runs from 0 to 1 (and $\gamma(0) = (x_0, y_0)$ and $\gamma(1) = (x_1, y_1)$). Surely, the bounding box has to contain (x_0, y_0) and (x_1, y_1) . If the functions $x(t)$ and $y(t)$ have extrema in the interval $[0, 1]$, then the bounding box will in general be larger than that. In order to determine the extrema of the curve, all we need to find the extrema of the functions $x(t)$ and $y(t)$ for $0 \leq t \leq 1$. That is, we need to find the solutions of the quadratic equations

$$\frac{dx}{dt}(t) = 0 \quad \text{and} \quad \frac{dy}{dt}(t) = 0. \quad (2)$$

Let's discuss x , y is analogous. If the discriminant

$$d := (x_a - x_b)^2 + (x_1 - x_b)(x_0 - x_a) \quad (3)$$

is greater than 0, there are two solutions

$$t_{\pm} = \frac{x_0 - 2x_a + x_b \pm \sqrt{d}}{x_0 - x_1 - 3(x_a - x_b)}. \quad (4)$$

In this case, we need to make sure that the bounding box contains, say $(x(t_-), y_0)$ and $(x(t_+), y_0)$. If $d \leq 0$, the bounding box does not need to be increased in the x direction. One can plug t_{\pm} back into (1), this yields

$$x_- = \frac{1}{(x_0 - x_1 - 3x_a + 3x_b)^2} \left[x_0^2 x_1 + x_0 x_1^2 - 3x_0 x_1 x_a + 6x_1 x_a^2 + 2x_a^3 - 3(x_0 + x_a)(x_1 + x_a)x_b + 3(2x_0 - x_a)x_b^2 + 2x_b^3 - 2\sqrt{d}(x_0 x_1 - x_1 x_a + x_a^2 - (x_0 + x_a)x_b + x_b^2) \right], \quad (5a)$$

$$x_+ = \frac{1}{(x_0 - x_1 - 3x_a + 3x_b)^2} \left[x_0^2 x_1 + x_0 x_1^2 - 3x_0 x_1 x_a + 6x_1 x_a^2 + 2x_a^3 - 3(x_0 + x_a)(x_1 + x_a)x_b + 3(2x_0 - x_a)x_b^2 + 2x_b^3 + 2\sqrt{d}(x_0 x_1 - x_1 x_a + x_a^2 - (x_0 + x_a)x_b + x_b^2) \right]. \quad (5b)$$

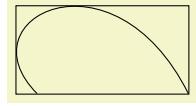
As already mentioned, the analogous statements apply to $y(t)$.

This procedure is implemented in the `bbox` library. It installs a single key by which the tight bounding box algorithm can be turned on and off.

`/pgf/bezier bounding box=<boolean>` (default `true`)

Turn the tight bounding box algorithm on and off.

Caveat: As can be seen from the derivations, the necessary computations involve the squaring of lengths, which can easily lead to `dimension too large` errors. The library tries to account for large numbers by appropriate normalization, such that it works in most cases, but errors may still occur.



```
\begin{tikzpicture}[bezier bounding box=true]
  \draw (0,0) .. controls (-1,1) and (1,2) .. (2,0);
  \draw (current bounding box.south west) rectangle
    (current bounding box.north east);
\end{tikzpicture}
```

47 Calc Library

TikZ Library `calc`

```
\usetikzlibrary{calc} % LATEX and plain TEX  
\usetikzlibrary[calc] % ConTeXt
```

The library allows advanced Coordinate Calculations. It is documented in all detail in Section 13.5 on page 125.

48 Calendar Library

TikZ Library `calendar`

```
\usetikzlibrary{calendar} % LATEX and plain TEX  
\usetikzlibrary[calendar] % ConTEX
```

The library defines the `\calendar` command, which can be used to typeset calendars. The command relies on the `\pgfcalendar` command from the `pgfcalendar` package, which is loaded automatically.

The `\calendar` command is quite configurable, allowing you to produce all kinds of different calendars.

48.1 Calendar Command

The core command for creating calendars in TikZ is the `\calendar` command. It is available only inside `{tikzpicture}` environments (similar to, say, the `\draw` command).

```
\calendar<calendar specification>;
```

The syntax for this command is similar to commands like `\node` or `\matrix`. However, it has its complete own parser and only those commands described in the following will be recognized, nothing else. Note, furthermore, that a `<calendar specification>` is not a path specification, indeed, no path is created for the calendar.

The specification syntax. The `<calendar specification>` must be a sequence of elements, each of which has one of the following structures:

- `[(options)]`

You provide `(options)` in square brackets as in `[red, draw=none]`. These `(options)` can be any TikZ option and they apply to the whole calendar. You can provide this element multiple times, the effect accumulates.

- `((name))`

This has the same effect as saying `[name=(name)]`. The effect of providing a `(name)` is explained later. Note already that a *calendar is not a node* and the `(name)` is *not the name of a node*.

- `at ((coordinate))`

This has the same effect as saying `[at=((coordinate))]`.

- `if (<date condition>) <options or commands> else <else options or commands>`

The effect of such an `if` is explained later.

At the beginning of every calendar, the following style is used:

```
/tikz/every calendar
```

 (style, initially empty)

This style is used with every calendar.

The date range. The overall effect of the `\calendar` command is to execute code for each day of a range of dates. This range of dates is set using the following option:

```
/tikz/dates=<start date>to<end date>
```

 (no default)

This option specifies the date range. Both the start and end date are specified as described on page ???. In short: You can provide ISO-format type dates like 2006-01-02, you can replace the day of month by `last` to refer to the last day of a month (so 2006-02-last is the same as 2006-02-28), and you can add a plus sign followed by a number to specify an offset (so 2006-01-01+-1 is the same as 2005-12-31).

It will be useful to fix two pieces of terminology for the following descriptions: The `\calendar` command iterates over the dates in the range. The *current date* refers to the current date the command is processing as it iterates over the dates. For each current date code is executed, which will be called the *current date code*. The current date code consists of different parts, to be detailed later.

The central part of the current date code is the execution of the code `\tikzdaycode`. By default, this code simply produces a node whose text is set to the day of month. This means that unless further action is taken, all days of a calendar will be put on top of each other! To avoid this, you must modify the current date code to shift days around appropriately. Predefined arrangements like `day list downward`

or `week list` do this for you, but you can define arrangements yourself. Since defining an arrangement is a bit tricky, it is explained only later on. For the time being, let us use a predefined arrangement to produce our first calendar:

	1	2
3	4	5
6	7	8
9		
10	11	12
13	14	15
16		
17	18	19
20	21	22
23		
24	25	26
27	28	29
30		
31		

```
\usetikzlibrary {calendar}
\tikz \calendar [dates=2000-01-01 to 2000-01-31,week list];
```

Changing the spacing. In the above calendar, the spacing between the days is determined by numerous options. Most arrangements do not use all of these options, but only those that apply naturally.

/tikz/day xshift=(dimension) (no default, initially `3.5ex`)

Specifies the horizontal shift between days. This is not the gap between days, but the shift between the anchors of their nodes.

	1	2
3	4	5
6	7	8
9		
10	11	12
13	14	15
16		
17	18	19
20	21	22
23		
24	25	26
27	28	29
30		
31		

```
\usetikzlibrary {calendar}
\tikz \calendar [dates=2000-01-01 to 2000-01-31,week list,day xshift=3ex];
```

/tikz/day yshift=(dimension) (no default, initially `3ex`)

Specifies the vertical shift between days. Again, this is the shift between the anchors of their nodes.

	1	2
3	4	5
6	7	8
9		
10	11	12
13	14	15
16		
17	18	19
20	21	22
23		
24	25	26
27	28	29
30		
31		

```
\usetikzlibrary {calendar}
\tikz \calendar [dates=2000-01-01 to 2000-01-31,week list,day yshift=2ex];
```

/tikz/month xshift=(dimension) (no default)

Specifies an additional horizontal shift between different months.

/tikz/month yshift=(dimension) (no default)

Specifies an additional vertical shift between different months.

	1	2
3	4	5
6	7	8
9		
10	11	12
13	14	15
16		
17	18	19
20	21	22
23		
24	25	26
27	28	29
30		
31	1	2
3	4	5
6		
7	8	9
10	11	12
13		
14	15	16
17	18	19
20		
21	22	23
24	25	26
27		
28	29	

```
\usetikzlibrary {calendar}
\tikz \calendar [dates=2000-01-01 to 2000-02-last,week list,
                month yshift=Opt];
```

							1	2
3	4	5	6	7	8	9		
10	11	12	13	14	15	16		
17	18	19	20	21	22	23		
24	25	26	27	28	29	30		
31								

```
\usetikzlibrary {calendar}
\begin{tikzpicture}
\draw [help lines] (0,0) grid (3,2);
\matrix [anchor=cal-2000-01-20.center] at (2,2)
{ \calendar (cal) [dates=2000-01-01 to 2000-02-last,week list];
\node {\scriptsize \texttt{\$}}; };
\end{tikzpicture}
```

	1	2	3	4	5	6
7	8	9	10	11	12	13
14	15	16	17	18	19	20
21	22	23	24	25	26	27
28	29					

Changing the position of the calendar. The calendar is placed in such a way that, normally, the anchor of the first day label is at the origin. This can be changed by using the `at` option. When you say `at={(1,1)}`, this anchor of the first day will lie at coordinate (1,1).

In general, arrangements will not always place the anchor of the first day at the origin. Sometimes, additional spacing rules get in the way. There are different ways of addressing this problem: First, you can just ignore it. Since calendars are often placed in their own `\tikzpicture` and since their size is computed automatically, the exact position of the origin often does not matter at all. Second, you can put the calendar inside a node as in `...node {\tikz \calendar ...}`. This allows you to position the node in the normal ways using the node's anchors. Third, you can be very clever and use a single-cell matrix. The advantage is that a matrix allows you to provide any anchor of any node inside the matrix as an anchor for the whole matrix. For example, the following calendar is placed in such a way the center of 2000-01-20 lies on the position (2,2):

			1	2				
3	4	5	6	7	8	9		
10	11	12	13	14	15	16		
17	18	19	20	21	22	23		
24	25	26	27	28	29	30		
31								

```
\usetikzlibrary {calendar}
\begin{tikzpicture}
\draw [help lines] (0,0) grid (3,2);
\matrix [anchor=cal-2000-01-20.center] at (2,2)
{ \calendar (cal) [dates=2000-01-01 to 2000-01-31,week list];
\node {\scriptsize \texttt{\$}}; };
\end{tikzpicture}
```

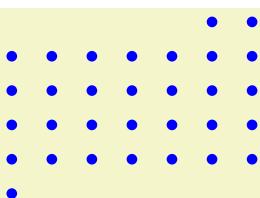
Unfortunately, the matrix-base positions, which is the cleanest way, isn't as portable as the other approaches (it currently does not work with the SVG backend for instance).

Changing the appearance of days. As mentioned before, each day in the above calendar is produced by an execution of the `\tikzdaycode`. Each time this code is executed, the coordinate system will have been set up appropriately to place the day of the month correctly. You can change both the code and its appearance using the following options.

`/tikz/day code=<code>`

(no default, initially see below)

This option allows you to change the code that is executed for each day. The default is to create a node with an appropriate name, but you can change this:



```
\usetikzlibrary {calendar}
\begin{tikzpicture}
\draw [help lines] (0,0) grid (3,2);
\matrix [anchor=cal-2000-01-20.center] at (2,2)
{ \calendar (cal) [dates=2000-01-01 to 2000-01-31,week list,
day code={\fill[blue] (0,0) circle (2pt)}];
\node {\scriptsize \texttt{\$}}; };
\end{tikzpicture}
```

The default code is the following:

```
\node [name=\pgfcalendar suggested name, every day]{\tikzdaytext};
```

The first part causes the day nodes to be accessible via the following names: If `<name>` is the name given to the calendar via a `name=` option or via the specification element `(<name>)`, then `\pgfcalendar suggested name` will expand to `<name>-<date>`, where `<date>` is the date of the day that is currently being processed in ISO format.

For example, if January 1, 2006 is being processed and the calendar has been named `mycal`, then the node containing the 1 for this date will be named `mycal-2006-01-01`. You can later reference this node.

	1	2
3	4	5
10	11	12
17	18	19
24	25	26
31	1	2
3	4	5
10	11	12
17	18	19
24	25	26
	21	22
	28	29
	30	

```
\usetikzlibrary {calendar}
\begin{tikzpicture}
  \calendar [mycal] [dates=2000-01-01 to 2000-01-31, week list];
  \draw[red] (mycal-2000-01-20) circle (4pt);
\end{tikzpicture}
```

`/tikz/day text=<text>`

(no default)

This option changes the setting of the `\tikzdaytext`. By default, this macro simply yields the current day of month, but you can change it arbitrarily. Here is a silly example:

	x	x
x	x	x
x	x	x
x	x	x
x	x	x
x	x	x
x	x	x
x		

```
\usetikzlibrary {calendar}
\tikz \calendar [dates=2000-01-01 to 2000-01-31, week list,
  day text=x];
```

More useful examples are based on using the `\%` command. This command is redefined inside a `\pgfcalendar` to mean the same as `\pgfcalendar shorthand`. (The original meaning of `\%` is lost inside the calendar, you need to save it before the calendar if you really need it.)

The `\%` inserts the current day/month/year/day of week in a certain format into the text. The first letter following the `\%` selects the type (permissible values are `d`, `m`, `y`, `w`), the second letter specifies how the value should be displayed (- means numerically, = means numerically with leading space, 0 means numerically with leading zeros, t means textual, and . means textual, abbreviated). For example `\%d0` gives the day with a leading zero (for more details see the description of `\pgfcalendar shorthand` on page ??).

Let us redefine the `day text` so that it yields the day with a leading zero:

	01	02
03	04	05
10	11	12
17	18	19
24	25	26
31	1	2
3	4	5
10	11	12
17	18	19
24	25	26
	21	22
	28	29
	30	

```
\usetikzlibrary {calendar}
\tikz \calendar [dates=2000-01-01 to 2000-01-31, week list,
  day text=\%d0];
```

`/tikz/every day (initially anchor=base east)`

(no default)

This style is executed by the default node code for each day. The `every day` style is useful for changing the way days look. For example, let us make all days red:

	1	2
3	4	5
6	7	8
9	10	11
12	13	14
15	16	
17	18	19
20	21	22
23	24	25
26	27	28
29	30	
31		

```
\usetikzlibrary {calendar}
\tikz[every day/.style=red]
\calendar[dates=2000-01-01 to 2000-01-31,week list];
```

Changing the appearance of month and year labels. In addition to the days of a calendar, labels for the months and even years (for really long calendars) can be added. These labels are only added once per month or year and this is not done by default. Rather, special styles starting with `month` label place these labels and make them visible:

January	1	2
3	4	5
6	7	8
9	10	11
12	13	14
15	16	17
18	19	20
21	22	23
24	25	26
27	28	29
30	31	

February	1	2	3	4	5	6
7	8	9	10	11	12	13
14	15	16	17	18	19	20
21	22	23	24	25	26	27
28	29					

```
\usetikzlibrary {calendar}
\tikz \calendar[dates=2000-01-01 to 2000-02-last,week list,
month label above centered];
```

The following options change the appearance of the month and year label:

/tikz/month code=(code) (no default, initially see below)

This option allows you to specify what the macro `\tikzmonthcode` should expand to.

By default, the `\tikzmonthcode` it is set to

```
\node[every month]{\tikzmonthtext};
```

Note that this node is not named by default.

/tikz/month text=(text) (no default)

This option allows you to change the macro `\tikzmonthtext`. By default, the month text is a long textual presentation of the current month being typeset.

January	2000	1	2
3	4	5	6
7	8	9	
10	11	12	13
14	15	16	17
18	19	20	21
22	23	24	25
26	27	28	29
30	31		

```
\usetikzlibrary {calendar}
\tikz \calendar[dates=2000-01-01 to 2000-01-31,week list,
month label above centered,
month text=\textcolor{red}{\%mt} \%y-];
```

/tikz/every month (style, initially empty)

This style can be used to change the appearance of month labels.

<code>/tikz/year code=(code)</code>	(no default)
Works like <code>month code</code> , only for years.	
<code>/tikz/year text=(text)</code>	(no default)
Works like <code>month text</code> , only for years.	
<code>/tikz/every year</code>	(no value)
Works like <code>every month</code> , only for years.	

Date ifs. Much of the power of the `\calendar` command comes from the use of conditionals. There are two equivalent ways of specifying such a conditional. First, you can add the text `if (<conditions> <code or options>)` to your `<calendar specification>`, possibly followed by `else<else code or options>`. You can have multiple such conditionals (but you cannot nest them in this simple manner). The second way is to use the following option:

<code>/tikz/if=(<conditions>)<code or options>else<else code or options></code>	(no default)
---	--------------

This option has the same effect as giving a corresponding `if` in the `<calendar specification>`. The option is mostly useful for use in the `every calendar` style, where you cannot provide if conditionals otherwise.

Now, regardless of how you specify a conditional, it has the following effect (individually and independently for each date in the calendar):

1. It is checked whether the current date is one of the possibilities listed in `<conditions>`. An example of such a condition is `Sunday`. Thus, when you write `if (Saturday,Sunday) {foo}`, then `foo` will be executed for every day in the calendar that is a Saturday or a Sunday.

The command `\ifdate` and, thereby, `\pgfcalendarifdate` are used to evaluate the `<conditions>`, see page ?? for a complete list of possible tests. The most useful tests are: Tests like `Monday` and so on, `workday` for the days Monday to Friday, `weekend` for Saturday and Sunday, `equals` for testing whether the current date equals a given date, `at least` and `at least` for comparing the current date with a given date.

2. If the date passes the check, the `<code or options>` is evaluated in a manner to be described in a moment; if the date fails, the `<else code or options>` is evaluated, if present.

The `<code or options>` can either be some code. This is indicated by surrounding the code with curly braces. It can also be a list of TikZ options. This is indicated by surrounding the options with square brackets. For example in the date test `if (Sunday) {\draw...} else {\fill...}` there are two pieces of code involved. By comparison, `if (Sunday) [red] else [green]` involves two options.

If `<code or options>` is code, it is simply executed (for the current day). If it is a list of options, these options are passed to a scope surrounding the current date.

Let us now have a look at some examples. First, we use a conditional to make all Sundays red.

```

1 2
3 4 5 6 7 8 9
10 11 12 13 14 15 16
17 18 19 20 21 22 23
24 25 26 27 28 29 30
31
\usetikzlibrary {calendar}
\tikz
  \calendar
    [dates=2000-01-01 to 2000-01-31,week list]
    if (Sunday) [red];

```

Next, let us do something on a specific date:

	1	2
3	4	5
6	7	8
9	10	11
12	13	14
15	16	
17	18	19
20	21	22
23		
24	25	26
27	28	29
30		
31		

```
\usetikzlibrary {calendar}
\begin{tikzpicture}
\calendar
[dates=2000-01-01 to 2000-01-31,week list]
if (Sunday) [red]
if (equals=2000-01-20) {\draw (0,0) circle (8pt);}

```

You might wonder why the circle seems to be “off” the date. Actually, it is centered on the date, it is just that the date label uses the `base east` anchor, which shifts the label up and right. To overcome this problem we can change the anchor:

	1	2
3	4	5
6	7	8
9	10	11
12	13	14
15	16	
17	18	19
20	21	22
23		
24	25	26
27	28	29
30		
31		

```
\usetikzlibrary {calendar}
\begin{tikzpicture}[every day/.style={anchor=mid}]
\calendar
[dates=2000-01-01 to 2000-01-31,week list]
if (Sunday) [red]
if (equals=2000-01-20) {\draw (0,0) circle (8pt);}

```

However, the single day dates are now no longer aligned correctly. For this, we can change the day text to `\%d`, which adds a space at the beginning of single day text.

In the following, more technical information is covered. Most readers may wish to skip it.

The current date code. As mentioned earlier, for each date in the calendar the current date code is executed. It is the job of this code to shift around date nodes, to render the date nodes, to draw the month labels and to do all other stuff that is necessary to draw a calendar.

The current date code consists of the following parts, in this order:

1. The before-scope code.
2. A scope is opened.
3. The at-begin-scope code.
4. All date-ifs from the *(calendar specification)* are executed.
5. The at-end-scope code.
6. The scope is closed.
7. The after-scope code.

All of the codes mentioned above can be changed using appropriate options, see below. In case you wonder why so many are needed, the reason is that the current date code as a whole is not surrounded by a scope or TeX group. This means that code executed in the before-scope code and in the after-scope code has an effect on all following days. For example, if the after-scope code modifies the transformation matrix by shifting everything downward, all following days will be shifted downward. If each day does this, you get a list of days, one below the other.

However, you do not always want code to have an effect on everything that follows. For instance, if a day has the date-if `if (Sunday) [red]`, we only want this Sunday to red, not all following days also. Similarly, sometimes it is easier to compute the position of a day relative to a fixed origin and we do not want any modifications of the transformation matrix to have an effect outside the scope.

By cleverly adjusting the different codes, all sorts of different day arrangements are possible.

/tikz/execute before day scope=<code> (no default)

The *<code>* is executed before everything else for each date. Multiple calls of this option have an accumulative effect. Thus, if you use this option twice, the code from the first use is used first for each day, followed by the code given the second time.

/tikz/execute at begin day scope=<code> (no default)

This code is execute before everything else inside the scope of the current date. Again, the effect is accumulative.

```
/tikz/execute at end day scope=<code>
```

(no default)

This code is executed just before the day scope is closed. The effect is also accumulative, however, in reverse order. This is useful to pair, say, `\scope` and `\endscope` commands in at-begin- and at-end-code.

```
/tikz/execute after day scope=<code>
```

(no default)

This is executed at the very end of the current date, outside the scope. The accumulation is also in reverse.

In the rest of the following subsections we have a look at how the different scope codes can be used to create different calendar arrangements.

48.1.1 Creating a Simple List of Days

We start with a list of the days of the calendar, one day below the other. For this, we simply shift the coordinate system downward at the end of the code for each day. This shift must be *outside* the day scope as we want day shifts to accumulate. Thus, we use the following code:

```
1 \usetikzlibrary {calendar}
2 \tikz
3   \calendar [dates=2000-01-01 to 2000-01-08,
4             execute after day scope=
5               {\pgftransformyshift{-1em}}];
6
7
8
```

Clearly, we can use this approach to create day lists going up, down, right, left, or even diagonally.

48.1.2 Adding a Month Label

We now want to add a month label to the left of the beginning of each month. The idea is to do two things:

1. We add code that is executed only on the first of each month.
2. The code is executed before the actual day is rendered. This ensures that options applying to the days do not affect the month rendering.

We have two options where we should add the month code: Either we add it at the beginning of the day scope or before. Either will work fine, but it might be safer to put the code inside the scope to ensure that settings to not inadvertently “leak outside”.

```
1 January    1 \usetikzlibrary {calendar}
2                   \tikz
3                   \calendar
4                     [dates=2000-01-01 to 2000-01-08,
5                      execute after day scope={\pgftransformyshift{-1em}},
6                      execute at begin day scope=
7                        {\ifdate{day of month=1}{\tikzmonthcode}{}},
8                      every month/.append style={anchor=base east,xshift=-2em}];
```

In the above code we used the `\ifdate{<condition>}{{<then code>}}{{<else code>}}` command, which is described on page ?? in detail and which has much the same effect as `if (<condition>){<then code>} else <else code>}`, but works in normal code.

48.1.3 Creating a Week List Arrangement

Let us now address a more complicated arrangement: A week list. In this arrangement there is line for each week. The horizontal placement of the days is thus that all Mondays lie below each other, likewise for all Tuesdays, and so on.

In order to typeset this arrangement, we can use the following approach: The origin of the coordinate system rests at the anchor for the Monday of each week. That means that at the end of each week the origin is moved downward one line. On all other days, the origin at the end of the day code is the same as at the beginning. To position each day correctly, we use code inside and at the beginning of the day scope to horizontally shift the day according to its day of week.

1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20	<pre>\usetikzlibrary {calendar} \tikz \calendar [dates=2000-01-01 to 2000-01-20, % each day is shifted right according to the day of week execute at begin day scope= {\pgftransformxshift{\pgfcalendarcurrentweekday em}}, % after each week, the origin is shifted downward: execute after day scope= {\ifdate{Sunday}{\pgftransformyshift{-1em}}{}};</pre>
---	---

48.1.4 Creating a Month List Arrangement

For another example, let us create an arrangement that contains one line for each month. This is easy enough to do as for weeks, unless we add the following requirement: Again, we want all days in a column to have the same day of week. Since months start on different days of week, this means that each row has to have an individual offset.

One possible way is to use the following approach: After each month (or at the beginning of each month) we advance the vertical position of the offset by one line. For horizontal placement, inside the day scope we locally shift the day by its day of month. Furthermore, we must additionally shift the day to ensure that the first day of the month lies on the correct day of week column. For this, we remember this day of week the first time we see it.

1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29
--

```
\usetikzlibrary {calendar}
\newcount\mycount
\tikz
\calendar
[dates=2000-01-01 to 2000-02-last,
execute before day scope=
{
\ifdate{day of month=1} {
% Remember the weekday of first day of month
\mycount=\pgfcalendarcurrentweekday
% Shift downward
\pgftransformyshift{-1em}
}
},
execute at begin day scope=
{
% each day is shifted right according to the day of month
\pgftransformxshift{\pgfcalendarcurrentday em}
% and additionally according to the weekday of the first
\pgftransformxshift{\the\mycount em}
};
```

48.2 Arrangements

An *arrangement* specifies how the days of calendar are arranged on the page. The `calendar` library defines a number of predefined arrangements.

We start with arrangements in which the days are listed in a long line.

`/tikz/day list downward`

(style, no value)

This style causes the days of a month to be typeset one below the other. The shift between days is given by `day yshift`. Between month an additional shift of `month yshift` is added.

```

28 \usetikzlibrary {calendar}
29 \tikz
30   \calendar [dates=2000-01-28 to 2000-02-03,
31             day list downward,month yshift=1em];

```

1
2
3

/tikz/day list upward

(style, no value)

Works as above, only the list grows upward instead of downward.

```

3 \usetikzlibrary {calendar}
2 \tikz
1   \calendar [dates=2000-01-28 to 2000-02-03,
31             day list upward,month yshift=1em];
30
29
28

```

/tikz/day list right

(style, no value)

This style also works as before, but the list of days grows to the right. Instead of `day yshift` and `month yshift`, the values of `day xshift` and `month xshift` are used.

28 29 30 31 1 2 3

```

\usetikzlibrary {calendar}
\tikz
  \calendar [dates=2000-01-28 to 2000-02-03,
            day list right,month xshift=1em];

```

/tikz/day list left

(style, no value)

As above, but the list grows left.

The next arrangement lists days by the week.

/tikz/week list

(style, no value)

This style creates one row for each week in the range. The value of `day xshift` is used for the distance between days in each week row, the value of `day yshift` is used for the distance between rows. In both cases, “distance” refers to the distance between the anchors of the nodes of the days (or, more generally, the distance between the origins of the little pictures created for each day).

The days inside each week are shifted such that Monday is always at the first position (to change this, you need to copy and then modify the code appropriately). If the date range does not start on a Monday, the first line will not start in the first column, but rather in the column appropriate for the first date in the range.

At the beginning of each month (except for the first month in the range) an additional vertical space of `month yshift` is added. If this is set to `0pt` you get a continuous list of days.

```

1 2
3 4 5 6 7 8 9
10 11 12 13 14 15 16
17 18 19 20 21 22 23
24 25 26 27 28 29 30
31

```

```

1 2 3 4 5 6
7 8 9 10 11 12 13
14 15 16 17 18 19 20
21 22 23 24 25 26 27
28 29

```

```

1 2
3 4 5 6 7 8 9
10 11 12 13 14 15 16
17 18 19 20 21 22 23
24 25 26 27 28 29 30
31 1 2 3 4 5 6
7 8 9 10 11 12 13
14 15 16 17 18 19 20
21 22 23 24 25 26 27
28 29

```

```

\usetikzlibrary {calendar}
\tikz
  \calendar [dates=2000-01-01 to 2000-02-last,week list];

```

```

\usetikzlibrary {calendar}
\tikz
  \calendar [dates=2000-01-01 to 2000-02-last,week list,
    month yshift=0pt];

```

The following arrangement gives a very compact view of a whole year.

/tikz/month list

(style, no value)

In this arrangement there is a row for each month. As for the `week list`, the `day xshift` is used for the horizontal distance. For the vertical shift, `month yshift` is used.

In each row, all days of the month are listed alongside each other. However, it is once more ensured that days in each column lie on the same day of week. Thus, the very first column contains only Mondays. If a month does not start with a Monday, its days are shifted to the right such that the days lie on the correct columns.

January	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
February	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29		
March	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
April	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
May	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
June	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	
July	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
August	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
September	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	
October	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
November	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	
December	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31

```

\usetikzlibrary {calendar}
\sffamily\scriptsize
\tikz
  \calendar [dates=2000-01-01 to 2000-12-31,
    month list,month label left,month yshift=1.25em]
    if (Sunday) [black!50];

```

48.3 Month Labels

For many calendars you may wish to add a label to each month. We have already covered how month nodes are created and rendered in the description of the `\calendar` command: use `month text`, `every month`, and also `month code` (if necessary) to change the appearance of the month labels.

What we have not yet covered is where these labels are placed. By default, they are not placed at all as there is no good “default position” for them. Instead, you can use one of the following options to specify a position for the labels:

`/tikz/month label left` (style, no value)

Places the month label to the left of the first day of the month. (For `week list` and `month list` where a month does not start on a Monday, the position is chosen “as if” the month had started on a Monday – which is usually exactly what you want.)

28	
29	
30	
31	
February	1
	2
	3

`/tikz/month label left vertical` (style, no value)

This style works like the above style, only the label is rotated counterclockwise by 90 degrees.

28	
29	
30	
31	
February	1
	2
	3

`/tikz/month label right` (style, no value)

This style places the month label to the right of the row in which the first day of the month lies. This means that for a day list the label is to the right of the first day, for a week list it is to the right of the first week, and for a month list it is to the right of the whole month.

28	
29	
30	
31	
1	February
2	
3	

`/tikz/month label right vertical` (style, no value)

Works as above, only the label is rotated clockwise by 90 degrees.

```

28
29
30
31

```

1	February
2	
3	

```
\usetikzlibrary {calendar}
\begin{tikzpicture}
\calendar [dates=2000-01-28 to 2000-02-03,
day list downward,month yshift=1em,
month label right vertical];

```

/tikz/month label above left

(style, no value)

This style places the month label above of the row of the first day, flushed left to the leftmost column. The amount by which the label is raised is fixed to 1.25em; use the `yshift` option with the month node to modify this.

February					
28	29	30	31	1	2

```
\usetikzlibrary {calendar}
\begin{tikzpicture}
\calendar [dates=2000-01-28 to 2000-02-03,
day list right,month xshift=1em,
month label above left];

```

20	21	22	23			
24	25	26	27	28	29	30
31						

```
\usetikzlibrary {calendar}
\begin{tikzpicture}
\calendar [dates=2000-01-20 to 2000-02-10,
week list,month label above left];

```

February					
1	2	3	4	5	6

/tikz/month label above centered

(style, no value)

Works as above, only the label is centered above the row containing the first day.

February																												
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29

```
\usetikzlibrary {calendar}
\begin{tikzpicture}
\calendar [dates=2000-02-01 to 2000-02-last,
day list right,month label above centered];

```

20	21	22	23			
24	25	26	27	28	29	30
31						

```
\usetikzlibrary {calendar}
\begin{tikzpicture}
\calendar [dates=2000-01-20 to 2000-02-10,
week list,month label above centered];

```

February					
1	2	3	4	5	6

`/tikz/month label above right`

(style, no value)

Works as above, but flushed right

20	21	22	23
24	25	26	27
28	29	30	
31			
February			
1	2	3	4
5	6	7	8
9	10	11	12
13	14	15	16
17	18	19	20
21	22	23	24
25	26	27	28
29			

```
\usetikzlibrary {calendar}
\begin{tikzpicture}
\calendar [dates=2000-01-20 to 2000-02-10,
week list,month label above right];

```

`/tikz/month label below left`

(style, no value)

Works like `month label above left`, only the label is placed below the row. This placement is not really useful with the `week list` arrangement, but rather with the `day list right` or `month list` arrangement.

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29
February																												

```
\usetikzlibrary {calendar}
\begin{tikzpicture}
\calendar [dates=2000-02-01 to 2000-02-last,
day list right,month label below left];

```

`/tikz/month label below centered`

(style, no value)

Works like `month label above centered`, only below.

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29
February																												

```
\usetikzlibrary {calendar}
\begin{tikzpicture}
\calendar [dates=2000-02-01 to 2000-02-last,
day list right,month label below centered];

```

48.4 Examples

In the following, some example calendars are shown that come either from real applications or are just nice to look at.

Let us start with a year-2100-countdown, in which we cross out dates as we approach the big celebration. For this, we set the shape to `strike out` for these dates.

December 2099						
X	2	3	4	5	6	
7	8	9	10	11	12	13
14	15	16	17	18	19	20
21	22	23	24	25	26	27
28	29	30	31			
January 2100						
	1	2	3			
4	5	6	7	8	9	10
11	12	13	14	15	16	17
18	19	20	21	22	23	24
25	26	27	28	29	30	31

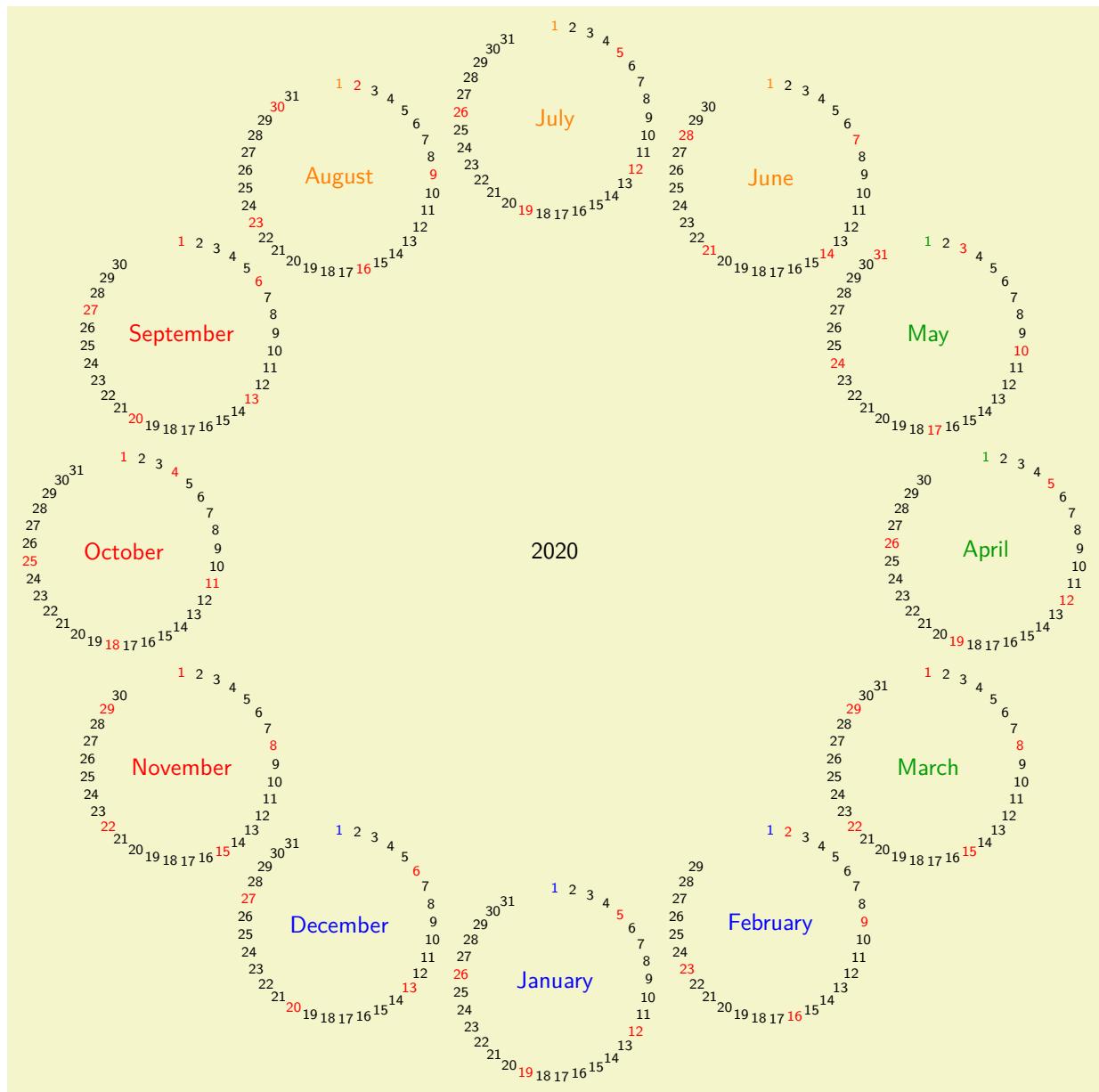
```
\usetikzlibrary {calendar,shapes.mis}
\begin{tikzpicture}
\calendar [
  dates=2099-12-01 to 2100-01-last,
  week list,inner sep=2pt,month label above centered,
  month text=\%mt \%y0
]
if (at most=2099-12-29) [nodes={strike out,draw}]
if (weekend) [black!50,nodes={draw=none}]
;
\end{tikzpicture}
```

The next calendar shows a deadline, which is 10 days in the future from the current date. The last three days before the deadline are in red, because we really should be done by then. All days on which we can no longer work on the project are crossed out.

31
June 2020
X 2 3 4 5 6 7
8 9 10 11 12 13 14
15 16 17 18 19 20 21
22 23 24 25 26 27 28
29 30
July 2020
1 2 3 4 5
6 7 8 9 10 11 12
13 14 15 16 17 18 19
20

```
\usetikzlibrary {calendar,shapes.mis}
\begin{tikzpicture}
\calendar [
  dates=\year-\month-\day+-25 to \year-\month-\day+25,
  week list,inner sep=2pt,month label above centered,
  month text=\textit{\%mt \%y0}
]
if (at least=\year-\month-\day) {}
else [nodes={strike out,draw}]
if (at most=\year-\month-\day+7)
  [green!50!black]
if (between=\year-\month-\day+8 and \year-\month-\day+10)
  [red]
if (Sunday)
  [gray,nodes={draw=none}]
;
\end{tikzpicture}
```

The following example is a futuristic calendar that is all about circles:



```

\usetikzlibrary {calendar}
\sffamily

\colorlet{winter}{blue}
\colorlet{spring}{green!60!black}
\colorlet{summer}{orange}
\colorlet{fall}{red}

% A counter, since TikZ is not clever enough (yet) to handle
% arbitrary angle systems.
\newcount\mycount

\begin{tikzpicture}
[transform shape,
 every day/.style={anchor=mid,font=\footnotesize{6}{6}\selectfont}]
\node[\normalsize{\the\year}];
\foreach \month/\monthcolor in
{1/winter,2/winter,3/spring,4/spring,5/spring,6/summer,
 7/summer,8/summer,9/fall,10/fall,11/fall,12/winter}
{
  % Compute angle:
  \mycount=\month
  \advance\mycount by -1
  \multiply\mycount by 30
  \advance\mycount by -90

  % The actual calendar
  \calendar at (\the\mycount:6.4cm)
  [
    dates=\the\year-\month-01 to \the\year-\month-last,
  ]
  if (day of month=1) {\color{\monthcolor}\tikzmonthcode}
  if (Sunday) [red]
  if (all)
  {
    % Again, compute angle
    \mycount=1
    \advance\mycount by -\pgfcalendarcurrentday
    \multiply\mycount by 11
    \advance\mycount by 90
    \pgftransformshift{\pgfpointpolar{\mycount}{1.4cm}}
  };
}
\end{tikzpicture}

```

Next, let's us have a whole year in a tight column:

```

01      1 2 3 4 5
       6 7 8 9 10 11 12
      13 14 15 16 17 18 19
      20 21 22 23 24 25 26
02 27 28 29 30 31 1 2
      3 4 5 6 7 8 9
      10 11 12 13 14 15 16
      17 18 19 20 21 22 23
03 24 25 26 27 28 29 1
      2 3 4 5 6 7 8
      9 10 11 12 13 14 15
      16 17 18 19 20 21 22
      23 24 25 26 27 28 29
04 30 31 1 2 3 4 5
      6 7 8 9 10 11 12
      13 14 15 16 17 18 19
      20 21 22 23 24 25 26
05 27 28 29 30 1 2 3
      4 5 6 7 8 9 10
      11 12 13 14 15 16 17
      18 19 20 21 22 23 24
      25 26 27 28 29 30 31
06 1 2 3 4 5 6 7
      8 9 10 11 12 13 14
      15 16 17 18 19 20 21
      22 23 24 25 26 27 28
07 29 30 1 2 3 4 5
      6 7 8 9 10 11 12
      13 14 15 16 17 18 19
      20 21 22 23 24 25 26
08 27 28 29 30 31 1 2
      3 4 5 6 7 8 9
      10 11 12 13 14 15 16
      17 18 19 20 21 22 23
      24 25 26 27 28 29 30
09 31 1 2 3 4 5 6
      7 8 9 10 11 12 13
      14 15 16 17 18 19 20
      21 22 23 24 25 26 27
10 28 29 30 1 2 3 4
      5 6 7 8 9 10 11
      12 13 14 15 16 17 18
      19 20 21 22 23 24 25
11 26 27 28 29 30 31 1
      2 3 4 5 6 7 8
      9 10 11 12 13 14 15
      16 17 18 19 20 21 22
      23 24 25 26 27 28 29
12 30 1 2 3 4 5 6
      7 8 9 10 11 12 13
      14 15 16 17 18 19 20
      21 22 23 24 25 26 27
      28 29 30 31

```

```

\usetikzlibrary {calendar}
\begin{tikzpicture}
  \small\sffamily
  \colorlet{darkgreen}{green!50!black}
  \calendar[dates=\year-01-01 to \year-12-31,week list,
    month label left,month yshift=0pt,
    month text=\textcolor{darkgreen}{\%m0}]
    if (Sunday) [black!50];
\end{tikzpicture}

```

49 Chains

TikZ Library `chains`

```
\usetikzlibrary{chains} % LATEX and plain TEX  
\usetikzlibrary[chains] % ConTEXt
```

This library defines options for creating chains.

49.1 Overview

Chains are sequences of nodes that are – typically – arranged in a row or a column and that are – typically – connected by edges. More generally, they can be used to position nodes of a branching network in a systematic manner. For the positioning of nodes in rows and columns you can also use matrices, see Section 20, but chains can also be used to describe the connections between nodes that have already been connected using, say, matrices. Thus, it often makes sense to use matrices for the positioning of elements and chains to describe the connections.

49.2 Starting and Continuing a Chain

Typically, you construct one chain at a time, but it is permissible to construct multiple chains simultaneously. In this case, the chains must be named differently and you must specify for each node which chain it belongs to.

The first step toward creating a chain is to use the `start chain` option.

```
/tikz/start chain=<chain name><direction>
```

(no default)

This key should, but need not, be given as an option to a scope enclosing all nodes of the chain. Typically, this will be a `scope` or the whole `tikzpicture`, but it might just be a path on which all nodes of the chain are found. If no `<chain name>` is given, the default value `chain` will be used instead.

The key starts a chain named `<chain name>` and makes it *active*, which means that it is currently being constructed. The `start chain` can be issued only once to activate a chain, inside a scope in which a chain is active you cannot use this option once more (for the same chain name). The chain stops being active at the end of the scope in which the `start chain` command was given.

Although chains are only locally active (that is, active inside the scope the `start chain` command was issued), the information concerning the chains is stored globally and it is possible to *continue* a chain after a scope has ended. For this, the `continue chain` option can be used, which allows you to reactivate an existing chain in another scope.

The `<direction>` is used to determine the placement rule for nodes on the chain. If it is omitted, the current value of the following key is used:

```
/tikz/chain default direction=<direction>
```

(no default, initially `going right`)

This `<direction>` is used in a `chain` option, if no other `<direction>` is specified.

The `<direction>` can have two different forms: `going <options>` or `placed <options>`. The effect of these rules will be explained in the description of the `on chain` option. Right now, just remember that the `<direction>` you provide with the `chain` option applies to the whole chain.

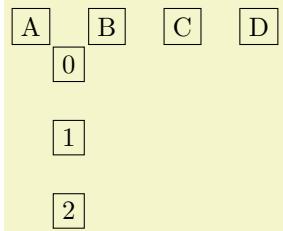
Other than this, this key has no further effect. In particular, to place nodes on the chain, you must use the `on chain` option, described next.

A B C

```
\usetikzlibrary {chains}  
\begin{tikzpicture} [start chain]  
  % The chain is called just "chain"  
  \node [on chain] {A};  
  \node [on chain] {B};  
  \node [on chain] {C};  
\end{tikzpicture}
```

A B C

```
\usetikzlibrary {chains,scopes}
\begin{tikzpicture}
    % Same as above, using the scope shorthand
    \begin{[start chain]
        \node [on chain] {A};
        \node [on chain] {B};
        \node [on chain] {C};
    }
\end{tikzpicture}
```



```
\usetikzlibrary {chains}
\begin{tikzpicture}[start chain=1 going right,
                  start chain=2 going below,
                  node distance=5mm,
                  every node/.style=draw]
\node [on chain=1] {A};
\node [on chain=1] {B};
\node [on chain=1] {C};

\node [on chain=2] at (0.5,-.5) {0};
\node [on chain=2] {1};
\node [on chain=2] {2};

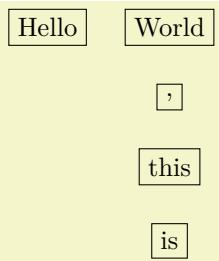
\node [on chain=1] {D};
\end{tikzpicture}
```

`/tikz/continue chain=<chain name><direction>`

(no default)

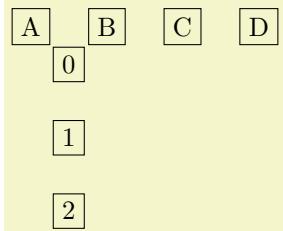
This option allows you to (re)activate an existing chain and to possibly change the default direction. If the `chain name` is missing, the name of the innermost activated chain is used. If no chain is activated, `chain` is used.

Let us have a look at the two different applications of this option. The first is to change the direction of a chain as it is being constructed. For this, just give this option somewhere inside the scope of the chain.



```
\usetikzlibrary {chains}
\begin{tikzpicture}[start chain=going right,node distance=5mm]
    \node [draw, on chain] {Hello};
    \node [draw, on chain] {World};
    \node [draw, continue chain=going below, on chain] {,};
    \node [draw, on chain] {this};
    \node [draw, on chain] {is};
\end{tikzpicture}
```

The second application is to reactivate a chain after it “has already been closed down”.



```
\usetikzlibrary {chains,scopes}
\begin{tikzpicture}[node distance=5mm,
                  every node/.style=draw]
\begin{[start chain=1]
    \node [on chain] {A};
    \node [on chain] {B};
    \node [on chain] {C};
\}
\begin{[start chain=2 going below]
    \node [on chain=2] at (0.5,-.5) {0};
    \node [on chain=2] {1};
    \node [on chain=2] {2};
\}
\begin{[continue chain=1]
    \node [on chain] {D};
\}
\end{tikzpicture}
```

49.3 Nodes on a Chain

`/tikz/on chain=(chain name)<direction>`

(no default)

This key should be given as an option to a node. When the option is used, the *<chain name>* must be the name of a chain that has been started using the `start chain` option. If *<chain name>* is the empty string, the current value of the innermost activated chain is used. If this option is used several times for a node, only the last invocation “wins”. (To place a node on several chains, use the `\chainin` command repeatedly.)

The *<direction>* part is optional. If present, it sets the direction used for this node, otherwise the *<direction>* that was given to the original `start chain` option is used (or of the last `continue chain` option, which allows you to change this default).

The effects of this option are the following:

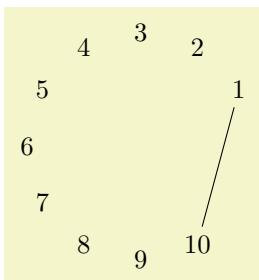
1. An internal counter (there is one local counter for each chain) is increased. This counter reflects the current number of the node in the chain, where the first node is node 1, the second is node 2, and so on.
The value of this internal counter is globally stored in the macro `\tikzchaincount`.
2. If the node does not yet have a name, (having been given using the `name` option or the name-syntax), the name of the node is set to *<chain name>-<value of the internal chain counter>*. For instance, if the chain is called `nums`, the first node would be named `nums-1`, the second `nums-2`, and so on. For the default chain name `chain`, the first node is named `chain-1`, the second `chain-2`, and so on.
3. Independently of whether the name has been provided automatically or via the `name` option, the name of the node is globally stored in the macro `\tikzchaincurrent`.
4. Except for the first node, the macro `\tikzchainprevious` is now globally set to the name of the node of the previous node on the chain. For the first node of the chain, this macro is globally set to the empty string.
5. Except possibly for the first node of the chain, the placement rule is now executed. The placement rule is just a TikZ option that is applied automatically to each node on the chain. Depending on the form of the *<direction>* parameter (either the locally given one or the one given to the `start chain` option), different things happen.

First, it makes a difference whether the *<direction>* starts with `going` or with `placed`. The difference is that in the first case, the placement rule is not applied to the first node of the chain, while in the second case the placement rule is applied also to this first node. The idea is that a `going`-direction indicates that we are “going somewhere relative to the previous node” whereas a `placed` indicates that we are “placing nodes according to their number”.

Independently of which form is used, the *<text>* inside *<direction>* that follows `going` or `placed` (separated by a compulsory space) can have two different effects:

- (a) If it contains an equal sign, then this *<text>* is used as the placement rule, that is, it is simply executed.
- (b) If it does not contain an equal sign, then *<text>=of \tikzchainprevious* is used as the placement rule.

Note that in the first case, inside the *<text>* you have access to `\tikzchainprevious` and `\tikzchaincount` for doing your positioning calculations.



```
\usetikzlibrary {chains}
\begin{tikzpicture} [start chain=circle placed at=(\tikzchaincount*30:1.5)]
\foreach \i in {1,...,10}
\node [on chain] {\i};

\draw (circle-1) -- (circle-10);
\end{tikzpicture}
```

6. The following style is executed:

/tikz/every on chain

(style, no value)

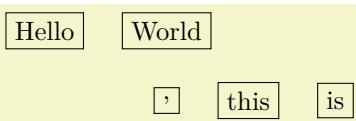
This key is executed for every node on a chain, including the first one.

Recall that the standard placement rule has a form like `right=of (\tikzchainprevious)`. This means that each new node is placed to the right of the previous one, spaced by the current value of `node distance`.



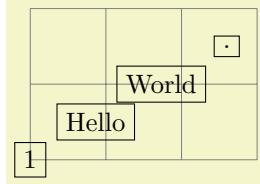
```
\usetikzlibrary {chains}
\begin{tikzpicture}[start chain,node distance=5mm]
\node [draw,on chain] {};
\node [draw,on chain] {Hallo};
\node [draw,on chain] {Welt};
\end{tikzpicture}
```

The optional `<direction>` allows us to temporarily change the direction in the middle of a chain:



```
\usetikzlibrary {chains}
\begin{tikzpicture}[start chain,node distance=5mm]
\node [draw,on chain] {Hello};
\node [draw,on chain] {World};
\node [draw,on chain=going below] {,};
\node [draw,on chain] {this};
\node [draw,on chain] {is};
\end{tikzpicture}
```

You can also use more complicated computations in the `<direction>`:



```
\usetikzlibrary {chains}
\begin{tikzpicture}[start chain=going {at=(\tikzchainprevious),shift=(30:1)}]
\draw [help lines] (0,0) grid (3,2);
\node [draw,on chain] {1};
\node [draw,on chain] {Hello};
\node [draw,on chain] {World};
\node [draw,on chain] {.};
\end{tikzpicture}
```

For each chain, two special “pseudo nodes” are created.

Predefined node `<chain name>-begin`

This node is the same as the first node on the chain. It is only defined after a first node has been defined.

Predefined node `<chain name>-end`

This node is the same as the (currently) last node on the chain. As the chain is extended, this node changes.

The `on chain` option can also be used, in conjunction with `late options`, to add an already existing node to a chain. The following command, which is only defined inside scopes where a `start chain` option is present, simplifies this process.

`\chainin(<existing name>) [<options>]`

This command makes it easy to add a node to chain that has already been constructed. This node may even be part of a another chain.

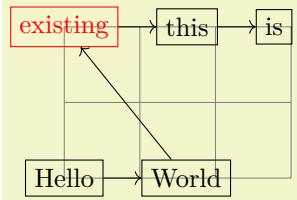
When you say `\chainin (some node);`, the node `some node` must already exist. It will then be made part of the current chain. This does not mean that the node can be changed (it is already constructed, after all), but the `join` option can be used to join `some node` to the previous last node on the chain and subsequent nodes will be placed relative to `some node`.

It is permissible to give the `on chain` option inside the `<options>` in order to specify on which chain the node should be put.

This command is just a shortcut for

```
\path (<existing name>) [late options={on chain,every chain in,<options>}]
```

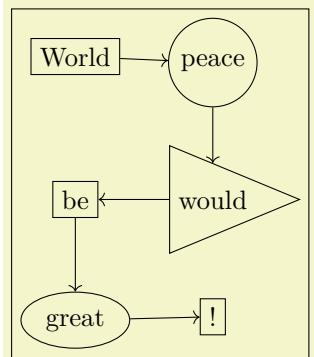
In particular, it is possible to continue to path after a `\chainin` command, though that does not seem very useful.



```
\usetikzlibrary {chains}
\begin{tikzpicture} [node distance=5mm,
                    every node/.style=draw,every join/.style=>]
\draw [help lines] (0,0) grid (3,2);

\node[red] (existing) at (0,2) {existing};
\begin{scope}[start chain]
\node [draw,on chain,join] {Hello};
\node [draw,on chain,join] {World};
\chainin (existing) [join];
\node [draw,on chain,join] {this};
\node [draw,on chain,join] {is};
\end{scope}
\end{tikzpicture}
```

Here is an example where nodes are positioned using a matrix and then connected using a chain



```
\usetikzlibrary {chains,matrix,scopes,shapes.geometric}
\begin{tikzpicture} [every node/.style=draw]
\matrix [matrix of nodes,column sep=5mm,row sep=5mm]
{
|(a)|      World & |(b)| [circle]|   peace \\
|(c)|      be    & |(d)| [isosceles triangle]| would \\
|(e)| [ellipse]| great & |(f)|           ! \\
};

% (the 'scopes' library needs to be loaded to make the following work)
\begin{scope}[start chain,every on chain/.style={join=by ->}]
\chainin (a);
\chainin (b);
\chainin (d);
\chainin (c);
\chainin (e);
\chainin (f);
\end{scope}
\end{tikzpicture}
```

49.4 Joining Nodes on a Chain

`/tikz/join=with<with> by<options>`

(no default)

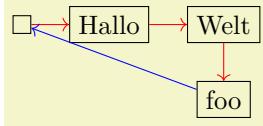
When this key is given to any node on a chain (except possibly for the first node), an `edge` command is added after the node. The `with` part specifies which node should be used for the start point of the edge; if the `with` part is omitted, the `\tikzchainprevious` is used. This `edge` command gets the `<options>` as parameter and the current node as its target. If there is no previous node and no `with` is given, no `edge` command gets executed.

`/tikz/every join`

(style, no value)

This style is executed each time this command is used.

Note that it makes sense to call this option several times for a node, in order to connect it to several nodes. This is especially useful for joining in branches, see the next section.



```
\usetikzlibrary {chains}
\begin{tikzpicture} [start chain, node distance=5mm,
                     every join/.style={->, red}]
\node [draw, on chain, join] {};
\node [draw, on chain, join] {Hallo};
\node [draw, on chain, join] {Welt};
\node [draw, on chain, going below,
       join, join=with chain-1 by {blue, <-}] {foo};
\end{tikzpicture}
```

49.5 Branches

A *branch* is a chain that (typically only temporarily) extends an existing chain. The idea is the following: Suppose we are constructing a chain and at some node x there is a fork. In this case, one (or even more) branches starts at this fork. For each branch a chain is created, but the first node on this chain should be x . For this, it is useful to use `\chainin` on the node x to make it part of the different branch chains and to name the branch chains in some way that reflects the name of the main chain.

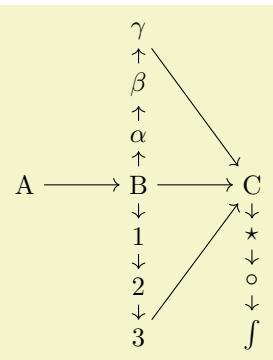
The `start` branch option provides a shorthand for doing exactly what was just described.

`/tikz/start branch=<branch name><direction>`

(no default)

This key is used in the same manner as the `start chain` command, however, the effect is slightly different:

- This option may only be used if some chain is already active and there is a (last) node on this chain. Let us call this node the *<fork node>*.
- The chain is not just called *<branch name>*, but *<current chain>/<branch name>*. For instance, if the *<fork node>* is part of the chain called `trunk` and the *<branch name>* is set to `left`, the complete chain name of the branch is `trunk/left`. The *<branch name>* must be given, there is no default value.
- The *<fork node>* is automatically “chained into” the branch chain as its first node. Thus, for the first node on the branch that you provide, the `join` option will cause it to be connected to the fork node.

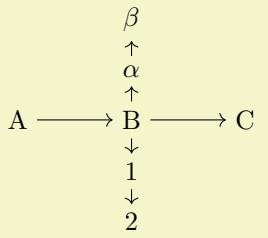


```
\usetikzlibrary {chains, scopes}
\begin{tikzpicture} [every on chain/.style=join, every join/.style=->,
                     node distance=2mm and 1cm]
{ [start chain=trunk]
  \node [on chain] {A};
  \node [on chain] {B};
{ [start branch=numbers going below]
  \node [on chain] {1};
  \node [on chain] {2};
  \node [on chain] {3};
}
{ [start branch=greek going above]
  \node [on chain] {$\alpha$};
  \node [on chain] {$\beta$};
  \node [on chain] {$\gamma$};
}
\end{tikzpicture}
\begin{tikzpicture} [on chain, join=with trunk/numbers-end, join=with trunk/greek-end] {C};
{ [start branch=symbols going below]
  \node [on chain] {$\star$};
  \node [on chain] {$\circ$};
  \node [on chain] {$\int$};
}
\end{tikzpicture}
```

`/tikz/continue branch=<branch name><direction>`

(no default)

This option works like the `continue chain` option, only *<current chain>/<branch name>* is used as the chain name, rather than just *<branch name>*.



```

\usetikzlibrary {chains,scopes}
\begin{tikzpicture}[every on chain/.style=join,every join/.style=->,
node distance=2mm and 1cm]
{ [start chain=trunk]
 \node [on chain] {A};
 \node [on chain] {B};
 { [start branch=numbers going below] } % just a declaration,
 { [start branch=greek going above] } % we will come back later
 \node [on chain] {C};

% Now come the branches...
{ [continue branch=numbers]
 \node [on chain] {1};
 \node [on chain] {2};
}
{ [continue branch=greek]
 \node [on chain] {$\alpha$};
 \node [on chain] {$\beta$};
}
}
\end{tikzpicture}

```

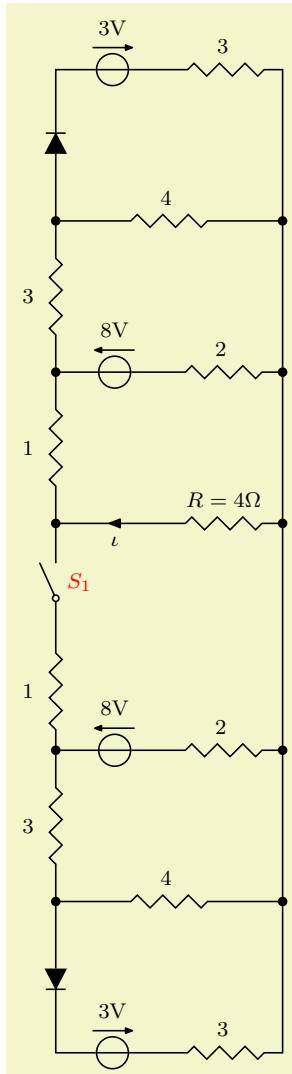
50 Circuit Libraries

Written and documented by Till Tantau, and Mark Wibrow. Inspired by the work of Massimo Redaelli.

50.1 Introduction

The circuit libraries can be used to draw different kinds of electrical or logical circuits. There is not a single library for this, but a whole hierarchy of libraries that work in concert. The main design goal was to create a balance between ease-of-use and ease-of-extending, while creating high-quality graphical representations of circuits.

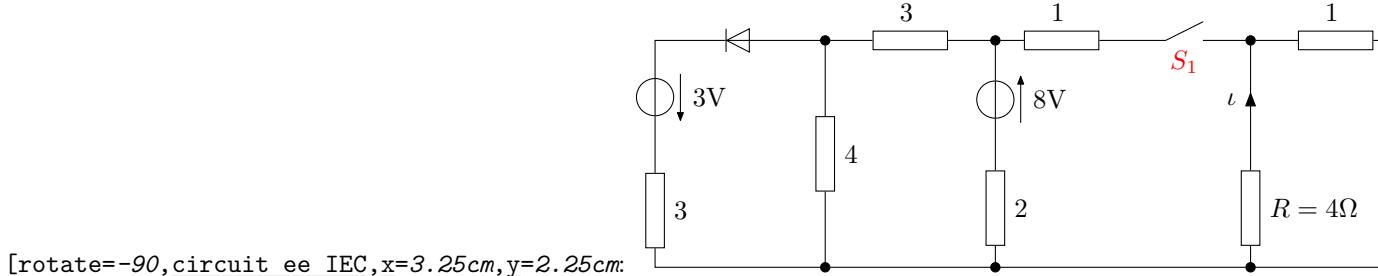
50.1.1 A First Example



```
\usetikzlibrary {circuits.ee.IEC}
\begin{tikzpicture}[circuit ee IEC, x=3cm, y=2cm, semithick,
  every info/.style={font=\footnotesize},
  small circuit symbols,
  set resistor graphic=var resistor IEC graphic,
  set diode graphic=var diode IEC graphic,
  set make contact graphic= var make contact IEC graphic]
% Let us start with some contacts:
\foreach \contact/\y in {1/1,2/2,3/3.5,4/4.5,5/5.5}
{
  \node [contact] (left contact \contact) at (0,\y) {};
  \node [contact] (right contact \contact) at (1,\y) {};
}
\draw (right contact 1) -- (right contact 2) -- (right contact 3)
  -- (right contact 4) -- (right contact 5);

\draw (left contact 1) to [diode] +(down:1)
  to [voltage source={near start,
    direction info={volt=3}}, 
    resistor={near end,ohm=3}] +(right:1)
  to (right contact 1);
\draw (left contact 1) to [resistor={ohm=4}] (right contact 1);
\draw (left contact 1) to [resistor={ohm=3}] (left contact 2);
\draw (left contact 2) to [voltage source={near start,
  direction info={<-,volt=8}}, 
  resistor={ohm=2,near end}] (right contact 2);
\draw (left contact 2) to [resistor={near start,ohm=1},
  make contact={near end,info'={[red]$S\_1$}}]
  (left contact 3);
\draw (left contact 3) to [current direction'={near start,info=$\iota$},
  resistor={near end,info={$R=4\Omega$}}]
  (right contact 3);
\draw (left contact 4) to [voltage source={near start,
  direction info={<-,volt=8}}, 
  resistor={ohm=2,near end}] (right contact 4);
\draw (left contact 3) to [resistor={ohm=1}] (left contact 4);
\draw (left contact 4) to [resistor={ohm=3}] (left contact 5);
\draw (left contact 5) to [resistor={ohm=4}] (right contact 5);
\draw (left contact 5) to [diode] +(up:1)
  to [voltage source={near start,
    direction info={volt=3}}, 
    resistor={near end,ohm=3}] +(right:1)
  to (right contact 5);
\end{tikzpicture}
```

An important feature of the `circuits` library is that the appearance of a circuit can be configured in general ways and that the labels are placed automatically by default. Here is the graphic once more, generated from *exactly the same source code*, with only the options of the `{tikzpicture}` environment replaced by



50.1.2 Symbols

A circuit typically consists of numerous electronic elements like logical gates or resistors or diodes that are connected by wires. In PGF/TikZ, we use nodes for the electronic elements and normal lines for the wires. TikZ offers a large number of different ways of positioning and connecting nodes in general, all of which can be used here. Additionally, the `circuits` library defines an additional useful `to-path` that is particularly useful for elements like a resistor on a line.

There are many different names that are used to refer to electrical “elements”, so a bit of terminology standardization is useful: We will call such elements *symbols*. A *symbol shape* is a PGF shape declared using the `\pgfdeclareshape` command. A *symbol node* is a node whose shape is a symbol shape.

50.1.3 Symbol Graphics

Symbols can be created by `\node[shape=some symbol shape]`. However, in order to represent some symbols correctly, just using standard PGF shapes is not sufficient. For instance, most symbols have a visually appealing “default size”, but the size of a symbol shape depends only on the current values of parameters like `minimum height` or `inner xsep`.

For these reasons, the circuit libraries introduce the concept of a *symbol graphic*. This is a style that causes a `\node` to not only have the correct shape, but also the correct size and the correct path usage. More generally, this style may set up things in any way so that the “symbol looks correct”. When you write, for instance, `\node[diode]`, then the style called `diode graphic` is used, which in turn is set to something like `shape=diode IEC,draw,minimum height=....`

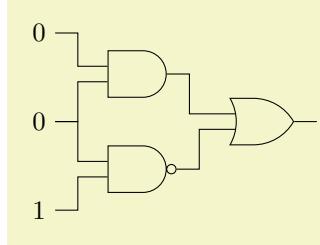
Here is an overview of the different kinds of circuit libraries:

- The TikZ-library `circuits` defines general keys for creating circuits. Mostly, these keys are useful for defining more specialized libraries.
You normally do not use this library directly since it does not define any symbol graphics.
- The TikZ-library `circuits.logic` defines keys for creating logical gates like and-gates or xor-gates. However, this library also does not actually define any symbol graphics; this is done by two sublibraries:
 - The library `circuits.logic.US` defines symbol graphics that cause the logical gates to be rendered in the “US-style”. It includes all of the above libraries and you can use this library directly.
 - The library `circuits.logic.IEC` also defines symbol graphics for logical gates, but it uses rectangular gates rather than the round US-gates. This library can coexist peacefully with the above library, you can change which symbol graphics are used “on the fly”.
- The TikZ-library `circuits.ee` defines keys for symbols from electrical engineering like resistors or capacitors. Again, sublibraries define the actual symbol graphics.
 - The library `circuits.ee.IEC` defines symbol shapes that follow the IEC norm.
- The PGF-libraries `shapes.gates.*` define (circuit) symbol shapes. However, you normally do not use these shapes directly, rather you use a style that uses an appropriate symbol graphic, which in turn uses one of these shapes.

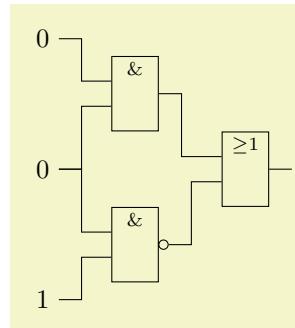
Let us have a look at a simple example. Suppose we wish to create a logical circuit. Then we first have to decide which symbol graphics we would like to use. Suppose we wish to use the US-style, then we would include the library `circuits.logic.US`. If you wish to use IEC-style symbols, use `circuits.logic.IEC`. If you cannot decide, include both:

```
\usetikzlibrary{circuits.logic.US,circuits.logic.IEC}
```

To create a picture that contains a US-style circuit you can now use the option `circuit logic US`. This will set up keys like `and gate` to create use an appropriate symbol graphic for rendering an `and` gate. Using the `circuit logic IEC` instead will set up `and gate` to use another symbol graphic.



```
\usetikzlibrary {circuits.logic.US}
\begin{tikzpicture}[circuit logic US]
 \matrix[column sep=7mm]
 {
 \node (i0) {0}; & & & & \\
 & \node [and gate] (a1) {}; & & & \\
 \node (i1) {0}; & & \node [or gate] (o) {};& & \\
 & \node [nand gate] (a2) {};& & & \\
 \node (i2) {1}; & & & & \\
 };
 \draw (i0.east) -- +(right:3mm) |- (a1.input 1);
 \draw (i1.east) -- +(right:3mm) |- (a1.input 2);
 \draw (i1.east) -- +(right:3mm) |- (a2.input 1);
 \draw (i2.east) -- +(right:3mm) |- (a2.input 2);
 \draw (a1.output) -- +(right:3mm) |- (o.input 1);
 \draw (a2.output) -- +(right:3mm) |- (o.input 2);
 \draw (o.output) -- +(right:3mm);
\end{tikzpicture}
```

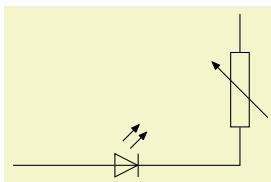


```
\usetikzlibrary {circuits.logic.IEC}
\begin{tikzpicture}[circuit logic IEC]
 \matrix[column sep=7mm]
 {
 \node (i0) {0}; & & & & \\
 & \node [and gate] (a1) {}; & & & \\
 \node (i1) {0}; & & \node [or gate] (o) {};& & \\
 & \node [nand gate] (a2) {};& & & \\
 \node (i2) {1}; & & & & \\
 };
 \draw (i0.east) -- +(right:3mm) |- (a1.input 1);
 \draw (i1.east) -- +(right:3mm) |- (a1.input 2);
 \draw (i1.east) -- +(right:3mm) |- (a2.input 1);
 \draw (i2.east) -- +(right:3mm) |- (a2.input 2);
 \draw (a1.output) -- +(right:3mm) |- (o.input 1);
 \draw (a2.output) -- +(right:3mm) |- (o.input 2);
 \draw (o.output) -- +(right:3mm);
\end{tikzpicture}
```

50.1.4 Annotations

An *annotation* is a little extra drawing that can be added to a symbol. For instance, when you add two little parallel arrows pointing away from some electrical element, this usually means that the element is light emitting.

Instead of having one symbol for “diode” and another for “light emitting diode”, there is just one `diode` symbol, but you can add the `light emitting` annotation to it. This is done by passing the annotation as a parameter to the symbol as in the following example:



```
\usetikzlibrary {circuits.ee.IEC}
\tikz [circuit ee IEC]
 \draw (0,0) to [diode={light emitting}] (3,0)
 to [resistor={adjustable}] (3,2);
```

50.2 The Base Circuit Library

TikZ Library `circuits`

```
\usetikzlibrary{circuits} % LATEX and plain TEX
\usetikzlibrary[circuits] % ConTeXt
```

This library is a base library that is included by other circuit libraries. You do not include it directly, but you will typically use some of the general keys, described below.

/tikz/circuits

(no value)

This key should be passed as an option to a picture or a scope that contains a circuit. It will do some internal setups. This key is normally called by more specialized keys like `circuit ee IEC`.

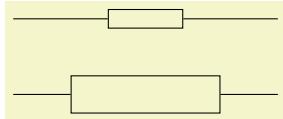
50.2.1 Symbol Size

/tikz/circuit symbol unit=<dimension>

(no default, initially 7pt)

This dimension is a “unit” for the size of symbols. The libraries generally define the sizes of symbols relative to this dimension. For instance, the longer side of an inductor is, by default, in the IEC library equal to five times this `<dimension>`. When you change this `<dimension>`, the size of all symbols will automatically change accordingly.

Note, that it is still possible to overwrite the size of any particular symbol. These settings apply only to the default sizes.



```
\usetikzlibrary {circuits.ee.IEC}
\begin{tikzpicture}[circuit ee IEC]
  \draw (0,1) to [resistor] (3.5,1);
  \draw[circuit symbol unit=14pt]
    (0,0) to [resistor] (3.5,0);
\end{tikzpicture}
```

/tikz/huge circuit symbols

(style, no value)

This style sets the default circuit symbol unit to 10pt.

/tikz/large circuit symbols

(style, no value)

This style sets the default circuit symbol unit to 8pt.

/tikz/medium circuit symbols

(style, no value)

This style sets the default circuit symbol unit to 7pt.

/tikz/small circuit symbols

(style, no value)

This style sets the default circuit symbol unit to 6pt.

/tikz/tiny circuit symbols

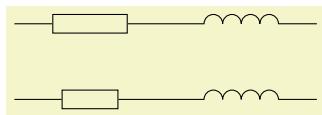
(style, no value)

This style sets the default circuit symbol unit to 5pt.

/tikz/circuit symbol size=width <width> height <height>

(no default)

This key sets `minimum height` to `<height>` times the current value of the circuit symbol unit and the `minimum width` to `<width>` times this value. Thus, this option can be used with a node command to set the size of the node as a multiple of the circuit symbol unit.



```
\usetikzlibrary {circuits.ee.IEC}
\begin{tikzpicture}[circuit ee IEC]
  \draw (0,1) to [resistor] (2,1) to [inductor] (4,1);

  \begin{scope}
    [every resistor/.style={circuit symbol size=width 3 height 1}]
    \draw (0,0) to [resistor] (2,0) to [inductor] (4,0);
  \end{scope}
\end{tikzpicture}
```

50.2.2 Declaring New Symbols

/tikz/circuit declare symbol=<name>

(no default)

This key is used to declare a symbol. It does not cause this symbol to be shown nor does it set a graphic to be used for the symbol, it simply “prepares” several keys that can later be used to draw a symbol and to configure it.

In detail, the first key that is defined is just called `<name>`. This key should be given as an option to a `node` or on a `to` path, as explained below. The key will take options, which can be used to influence the way the symbol graphic is rendered.

Let us have a look at an example. Suppose we want to define a symbol called `foo`, which just looks like a simple rectangle. We could then say

```
\tikzset{circuit declare symbol=foo}
```

The symbol could now be used like this:

```
\node [foo]      at (1,1) {};
\node [foo={red}] at (2,1) {};
```

However, in the above example we would not actually see anything since we have not yet set up the graphic to be used by `foo`. For this, we must use a key called `set foo graphic` or, generally, `set <name> graphic`. This key gets graphic options as parameter that will be set when a symbol `foo` should be shown:



```
\usetikzlibrary {circuits}
\begin{tikzpicture}
[circuit declare symbol=foo,
 set foo graphic={draw,shape=rectangle,minimum size=5mm}]

\node [foo]      at (1,1) {};
\node [foo={red}] at (2,1) {};
\end{tikzpicture}
```

In detail, when you use the key `<name>=<options>` with a node, the following happens:

1. The `inner sep` is set to 0.5pt.
2. The following style is executed:

`/tikz/every circuit symbol` (style, no value)

Use this style to set up things in general.

3. The graphic options that have been set using `set <name> graphic` are set.
4. The style `every <name>` is executed. You can use it to configure the symbol further.
5. The `<options>` are executed.

The key `<name>` will have a different effect when it is used on a `to` path command inside a `circuit` environment (the `circuit` environment sets up `to` paths in such a way that the use of a key declared using `circuit declare symbol` is automatically detected). When `<name>` is used on a `to` path, the above actions also happen (setting the inner separation, using the symbol graphic, and so on), but they are passed to the key `circuit handle symbol`, which is explained next.

`/tikz/circuit handle symbol=<options>` (no default)

This key is mostly used internally. Its purpose is to render a symbol. The effect of this key differs, depending on whether it is used as the optional argument of a `to` path command or elsewhere.

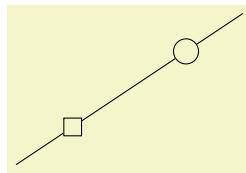
If the key is not used as an argument of a `to` path command, the `<options>` are simply executed.

The more interesting case happens when the key is given on a `to` path command. In this case, several things happen:

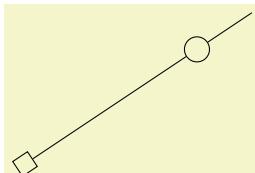
1. The `to` path is locally changed and set to an internal path (which you should not try to change) that consists mostly of a single straight line.
2. The `<options>` are tentatively executed with filtering switched on. Everything is filtered out, except for the key `pos` and also the styles `at start`, `very near start`, `near start`, `midway`, `near end`, `very near end`, and `at end`. If none of them is found, `midway` is used.
3. The filtered option is used to determine a position for the symbol on the path. At the given position (with `pos=0` representing the start and `pos=1` representing the end), a node will be added to the path (in a manner to be described presently).
4. This node gets `<options>` as its option list.
5. The node is added by virtue of a special `markings` decoration. This means that a `mark` command is executed that causes the node to be placed as a mark on the path.

6. The marking decoration will automatically subdivide the path and cause a line to be drawn from the start of the path to the node's border (at the position that lies on a line from the node's center to the start of the path) and then from the node's border (at a position on the other side of the node) to the end of the path.
7. The marking decoration will also take care of the case that multiple marks are present on a path, in this case the lines from and to the borders of the nodes are only between consecutive nodes.
8. The marking decoration will also rotate the coordinate system in such a way that the *x*-axis points along the path. Thus, if you use the `transform shape` option, the node will “point along” the path.
9. In case a node is at `pos=0` or at `pos=1` some special code will suppress the superfluous lines to the start or end of the path.

The net effect of all of the above is that a node will be placed “on the path” and the path will have a “gap” just large enough to encompass the node. Another effect is that you can use this key multiple times on a path to add several node to a path, provided they do not overlap.



```
\usetikzlibrary {circuits}
\begin{tikzpicture}[circuit]
  \draw (0,0) to [circuit handle symbol={draw,shape=rectangle,near start},
                  circuit handle symbol={draw,shape=circle,near end}] (3,2);
\end{tikzpicture}
```

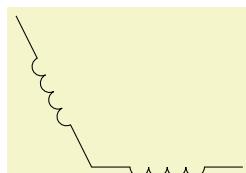


```
\usetikzlibrary {circuits}
\begin{tikzpicture}[transform shape,circuit]
  \draw (0,0) to [circuit handle symbol={draw,shape=rectangle,at start},
                  circuit handle symbol={draw,shape=circle,near end}] (3,2);
\end{tikzpicture}
```

50.2.3 Pointing Symbols in the Right Direction

Unlike normal nodes, which generally should not be rotated since this will make their text hard to read, symbols often need to be rotated. There are two ways of achieving such rotations:

1. When you place a symbol on a `to` path, the graphic symbol is automatically rotated such that it “points along the path”. Here is an examples that shows how the inductor shape (which looks, unrotated, like this: $\sim\sim\sim$) is automatically rotated around:



```
\usetikzlibrary {circuits.ee.IEC}
\tikz [circuit ee IEC]
  \draw (3,0) to[inductor] (1,0) to[inductor] (0,2);
```

2. Many shapes cannot be placed “on” a path in this way, namely whenever there are more than two possible inputs. Also, you may wish to place the nodes first, possibly using a matrix, and connect them afterwards. In this case, you can simply add rotations like `rotate=90` to the shapes to rotate them. The following four keys make this slightly more convenient:

`/tikz/point up` (no value)

This is the same as `rotate=90`.



```
\usetikzlibrary {circuits.ee.IEC}
\tikz [circuit ee IEC] \node [diode,point up] {};
```

/tikz/point down

(no value)

This is the same as `rotate=-90`.



```
\usetikzlibrary {circuits.ee.IEC}
\tikz [circuit ee IEC] \node [diode,point down] {};
```

/tikz/point left

(no value)

This is the same as `rotate=-180`.



```
\usetikzlibrary {circuits.ee.IEC}
\tikz [circuit ee IEC] \node [diode,point left] {};
```

/tikz/point right

(no value)

This key has no effect.



```
\usetikzlibrary {circuits.ee.IEC}
\tikz [circuit ee IEC] \node [diode,point right] {};
```

50.2.4 Info Labels

Info labels are used to add text to a circuit symbol. Unlike normal nodes like a rectangle, circuit symbols typically do not have text “on” them, but the text is placed next to them (like the text “ 3Ω ” next to a resistor).

TikZ already provides the `label` option for this purpose. The `info` option is built on top of this option, but it comes in some predefined variants that are especially useful in conjunction with circuits.

/tikz/info=[⟨options⟩]⟨angle⟩:⟨text⟩

(no default)

This key has nearly the same effect as the `label` key, only the following style is used additionally automatically:

/tikz/every info

(style, no value)

Set this style to configure the styling of info labels. Since this key is *not* used with normal labels, it provides an easy way of changing the way info labels look without changing other labels.

The ⟨options⟩ and ⟨angle⟩ are passed directly to the `label` command.



```
\usetikzlibrary {circuits.ee.IEC}
\begin{tikzpicture}[circuit ee IEC,every info/.style=red]
\node [resistor,info=$3\Omega$];
\end{tikzpicture}
```

You will find a detailed discussion of the `label` option on page 228.

Hint: To place some text *on* the main node, use `center` as the ⟨angle⟩:

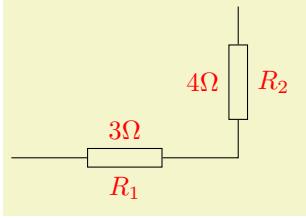


```
\usetikzlibrary {circuits.ee.IEC}
\begin{tikzpicture}[circuit ee IEC,every info/.style=red]
\node [resistor,info=center:$3\Omega$];
\node [resistor,point up,info=center:$R_1$ at (2,0)];
\end{tikzpicture}
```

/tikz/info'=[⟨options⟩]⟨angle⟩:⟨text⟩

(no default)

This key works exactly like the `info` key, only in case the ⟨angle⟩ is missing, it defaults to `below` instead of the current value of `label position`, which is usually `above`. This means that when you use `info`, you get a label above the node, while when you use the `info'` key you get a label below the node. In case the node has been rotated, the positions of the info nodes are rotated accordingly.

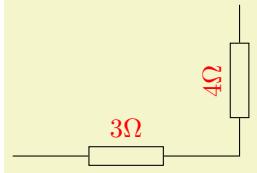


```
\usetikzlibrary {circuits.ee.IEC}
\begin{tikzpicture}[circuit ee IEC,every info/.style=red]
  \draw (0,0) to[resistor={info={$3\Omega$},info'={$R_1$}}] (3,0)
    to[resistor={info={$4\Omega$},info'={$R_2$}}] (3,2);
\end{tikzpicture}
```

`/tikz/info sloped=[<options>]<angle>:<text>`

(no default)

This key works like `info`, only the `transform shape` option is set when the label is drawn, causing it to follow the sloping of the main node.

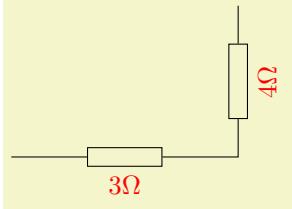


```
\usetikzlibrary {circuits.ee.IEC}
\begin{tikzpicture}[circuit ee IEC,every info/.style=red]
  \draw (0,0) to[resistor={info sloped={$3\Omega$}}] (3,0)
    to[resistor={info sloped={$4\Omega$}}] (3,2);
\end{tikzpicture}
```

`/tikz/info' sloped=`

(no default)

This is a combination of `info'` and `info sloped`.



```
\usetikzlibrary {circuits.ee.IEC}
\begin{tikzpicture}[circuit ee IEC,every info/.style=red]
  \draw (0,0) to[resistor={info' sloped={$3\Omega$}}] (3,0)
    to[resistor={info' sloped={$4\Omega$}}] (3,2);
\end{tikzpicture}
```

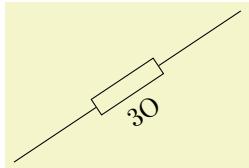
`/tikz/circuit declare unit={<name>}{<unit>}`

(no default)

This key is used to declare keys that make it easy to attach physical units to nodes. The idea is that instead of `info=3Ω` you can write `ohm=3` or instead of `info'=$5\mathit{ohm}$` you can write `siemens'=5`.

In detail, four keys are defined, namely `/tikz/<name>`, `/tikz/<name>'`, `/tikz/<name> sloped`, and `/tikz/<name>' sloped`. The arguments of all of these keys are of the form `[<options>]<angle>:<value>` and it is passed (slightly modified) to the corresponding key `info`, `info'`, `info sloped`, or `info' sloped`. The “slight modification” is the following: The text that is passed to the, say, `info` key is not `<value>`, but rather `$\mathit{value}<unit>$`.

This means that after you said `circuit declare unit={ohm}{Ω}`, then `ohm=5k` will have the same effect as `info={[every ohm]}$5\mathit{k}\Omega$`. Here, `every ohm` is a style that allows you to configure the appearance of this unit. Since the `info` key is used internally, by changing the `every info` style, you can change the appearance of all units infos.



```
\usetikzlibrary {circuits.ee.IEC}
\begin{tikzpicture}[circuit ee IEC,circuit declare unit={my ohm}{\Omega}]
  \draw (0,0) to[resistor={my ohm' sloped=3}] (3,2);
\end{tikzpicture}
```

50.2.5 Declaring and Using Annotations

Annotations are quite similar to `info` labels. The main difference is that they generally cause something to be drawn by default rather than some text to be added (although an annotation might also add some text).

Annotations can be declared using the following key:

```
/tikz/circuit declare annotation={⟨name⟩}{⟨distance⟩}{⟨path⟩} (no default)
```

This key is used to declare an annotation named ⟨name⟩. Once declared, it can be used as an argument of a symbol and will add the drawing in ⟨path⟩ to the symbol. In detail, the following happens:

The Main Keys. Two keys called ⟨name⟩ and ⟨name⟩' are defined. The second causes the annotation to be “mirrored and placed on the other side” of the symbol. Both of these keys may also take further keys as parameter like `info` keys. Whenever the ⟨name⟩ key is used, a local scope is opened and in this scope the following things are done:

1. The style `every ⟨name⟩` is executed.
2. The following style is executed and then `arrows=>`:

```
/tikz/annotation arrow (style, no value)
```

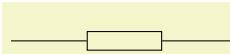
This style should set the `>` key to some desirable arrow tip.

3. The coordinate system is shifted such that the origin is at the north anchor of the symbol. (For the ⟨name⟩' key the coordinate system is flipped and shifted such that the origin is at the south anchor of the symbol.)
4. The `label distance` is locally set to ⟨distance⟩.
5. The parameter options given to the ⟨name⟩ key are executed.
6. The ⟨path⟩ is executed.

Usage. What all of the above amounts to is best explained by an example. Suppose we wish to create an annotation that looks like a little circular arrow (like \circlearrowright). We could then say:

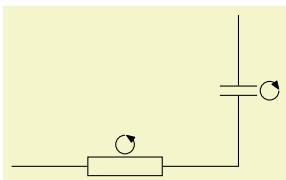
```
\tikzset{circuit declare annotation=
  {circular annotation}
  {9pt}
  {(0pt,8pt) arc (-270:80:3.5pt)}}
}
```

We can then use it like this:



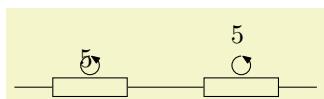
```
{\usetikzlibrary {circuits.ee.IEC}} pre
\tikz[circuit ee IEC]
\draw (0,0) to [resistor={circular annotation}] (3,0);
```

Well, not very impressive since we do not see anything. This is due to the fact that the ⟨path⟩ becomes part of a path that contains the symbol node an nothing else. This path is not drawn or filled, so we do not see anything. What we must do is to use an `edge` path operation:



```
{\usetikzlibrary {circuits.ee.IEC}}
\tikzset{circuit declare annotation={circular annotation}{9pt}
  {(0pt,8pt) edge[to path={arc(-270:80:3.5pt)}] ()}}
}
\tikz[circuit ee IEC]
\draw (0,0) to [resistor={circular annotation}] (3,0)
  to [capacitor={circular annotation'}] (3,2);
```

The ⟨distance⟩ is important for the correct placement of additional `info` labels. When an annotation is present, the info labels may need to be moved further away from the symbol, but not always. For this reason, an annotation defines an additional ⟨distance⟩ that is applied to all info labels given as parameters to the annotation. Here is an example, that shows the difference:

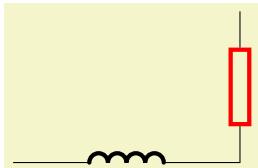


```
{\usetikzlibrary {circuits.ee.IEC}} pre
\tikz[circuit ee IEC]
\draw (0,0) to [resistor={circular annotation,ohm=5}] (2,0)
  to [resistor={circular annotation={ohm=5}}] (4,0);
```

50.2.6 Theming Symbols

For each symbol, a certain graphical representation is chosen to actually show the symbol. You can modify this graphical representation in several ways:

- You can select a different library and use a different `circuit ...` key. This will change all graphics used for the symbols.
- You can generally change the size of graphic symbols by setting `circuit size unit` to a different value or using a key like `small circuit symbols`.
- You can add options to the graphics used by symbols either globally by setting the `every circuit symbol` style or locally by setting the `every <name>` style, where `<name>` is the name of a symbol. For instance, in the following picture the symbols are ridiculously thick and resistors are red.



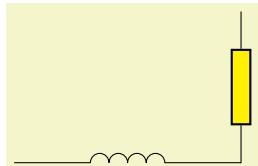
```
\usetikzlibrary {circuits.ee.IEC}
\begin{tikzpicture}
[circuit ee IEC,
 every circuit symbol/.style={ultra thick},
 every resistor/.style={red}]
\draw (0,0) to [inductor] ++(right:3) to [resistor] ++(up:2);
\end{tikzpicture}
```

- You can selectively change the graphic used for a symbol by saying `set resistor graphic=`.
- You can change one or more of the following styles:

`/tikz/circuit symbol open`

(style, initially `draw`)

This style is used with symbols that consist of lines that surround some area. For instance, the IEC version of a resistor is an open symbol.



```
\usetikzlibrary {circuits.ee.IEC}
\tikz [circuit ee IEC,
      circuit symbol open/.style={thick,draw,fill=yellow}]
\draw (0,0) to [inductor] ++(right:3) to [resistor] ++(up:2);
```

`/tikz/circuit symbol filled`

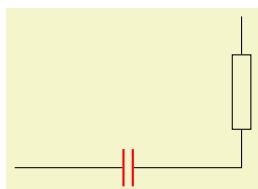
(style, initially `draw,fill=black`)

This style is used with symbols that are completely filled. For instance, the variant IEC version of an inductor is a filled, black rectangle.

`/tikz/circuit symbol lines`

(style, initially `draw`)

This style is used with symbols that consist only of lines that do not surround anything. Examples are a capacitor.



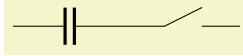
```
\usetikzlibrary {circuits.ee.IEC}
\tikz [circuit ee IEC,
      circuit symbol lines/.style={thick,draw=red}]
\draw (0,0) to [capacitor] ++(right:3) to [resistor] ++(up:2);
```

`/tikz/circuit symbol wires`

(style, initially `draw`)

This style is used for symbols that consist only of “wires”. The difference to the previous style is that a symbol consisting of wires will look strange when the lines are thicker than the lines of normal wires, while for symbols consisting of lines (but not wires) it may look nice to make them thicker. An example is the `make contact` symbol.

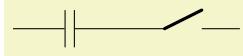
Compare



```
\usetikzlibrary {circuits.ee.IEC}
\begin{tikz} [circuit ee IEC,circuit symbol lines/.style={draw,very thick}]
\draw (0,0) to [capacitor={near start}, make contact={near end}] (3,0);

```

to



```
\usetikzlibrary {circuits.ee.IEC}
\begin{tikz} [circuit ee IEC,circuit symbol wires/.style={draw,very thick}]
\draw (0,0) to [capacitor={near start}, make contact={near end}] (3,0);

```

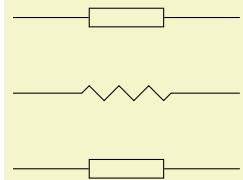
All circuit environments like `circuit logic IEC` mainly use options like `set` and `gate graphic=...` to set up the graphics used for a certain symbol. It turns out that graphic hidden in the “...” part is also always available as a separate style, whose name contains the library’s initials. For instance, the `circuit logic IEC` option actually contains the following command:

```
set and gate graphic = and gate IEC graphic,
```

The `and gate IEC graphic` style, in turn, is defined as follows:

```
\tikzset{and gate IEC graphic/.style=
{
    circuit symbol open,
    circuit symbol size=width 2.5 height 4,
    shape=and gate IEC,
    inner sep=.5ex
}}
```

Normally, you do not need to worry about this, since you will not need to access a style like `and gate IEC graphic` directly; you will only use the `and gate` key. However, sometimes libraries define *variants* of a graphic; for instance, there are two variants for the resistor graphic in the IEC library. In this case you can set the graphic for the resistor to this variant (or back to the original) by saying `set resistor graphic` yourself:



```
\usetikzlibrary {circuits.ee.IEC}
\begin{tikzpicture}[circuit ee IEC]
% Standard resistor
\draw (0,2) to [resistor] (3,2);

% Var resistor
\begin{scope}[set resistor graphic=var resistor IEC graphic]
\draw (0,1) to [resistor] (3,1);
\end{scope}

% Back to original
\draw [set resistor graphic=resistor IEC graphic]
(0,0) to [resistor] (3,0);
\end{tikzpicture}
```

50.3 Logical Circuits

50.3.1 Overview

A *logical circuit* is a circuit that contains what we call *logical gates* like an `and gate` or an `xor gate`. The logical libraries are intended to make it easy to draw such circuits.

In the following, we first have a look at the different libraries that can be used in principle and how the symbols look like. Then we have a more detailed look at how the symbols are used. Finally, we discuss the implementation details.

There are different ways of depicting logical gates, which is why there are different (sub-)libraries for drawing them. They provide the necessary graphical representations of the symbols declared in the following library:

TikZ Library `circuits.logic`

```
\usetikzlibrary{circuits.logic} % LATEX and plain TEX
\usetikzlibrary[circuits.logic] % ConTEXt
```

This library declares the logical gate symbols, but does not provide the symbol graphics. The library also defines the following key which, however, is also only used indirectly, namely by other libraries:

/tikz/circuit logic (no value)

This style calls the keys `circuit` (which internally calls `every circuit`, then it defines the `inputs` key and it calls the `every circuit logic` key.

/tikz/inputs=<inputs> (no default)

This key is defined only inside the scope of a `circuit logic`. There, it has the same effect as `logic gate inputs`, described on page 465.

/tikz/every circuit logic (style, no value)

Use this key to configure the appearance of logical circuits.

Since the `circuits.logic` library does not define any actual graphics, you need to use one of the following libraries, instead:

TikZ Library `circuits.logic.IEC`

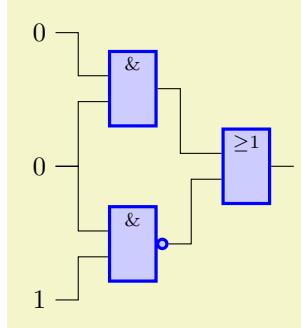
```
\usetikzlibrary{circuits.logic.IEC} % LATEX and plain TEX
\usetikzlibrary[circuits.logic.IEC] % ConTeXt
```

This library provides graphics based on gates recommended by the International Electrotechnical Commission. When you include this library, you can use the following key to set up a scope that contains a logical circuit where the gates are shown in this style.

/tikz/circuit logic IEC (no value)

This key calls `circuit logic` and installs the IEC-like graphics for the logical symbols like `and gate`.

As explained in Section 50.2.6, for each graphic symbol of the library there is also a style that stores this particular appearance. These keys are called `and gate IEC graphic`, or `gate IEC graphic`, and so on.



```
\usetikzlibrary {circuits.logic.IEC}
\begin{tikzpicture}[circuit logic IEC,
    every circuit symbol/.style={%
        logic gate IEC symbol color=black,
        fill=blue!20,draw=blue,very thick}]
\matrix[column sep=7mm]
{
    \node (i0) {0}; & \& \\
    & & \& \\
    \node (i1) {0}; & \& \& \node [or gate] (o) {≥1}; \\
    & & & \& \\
    \node (i2) {1}; & \& & & \\
};
\draw (i0.east) -- +(right:3mm) |- (a1.input 1);
\draw (i1.east) -- +(right:3mm) |- (a1.input 2);
\draw (i1.east) -- +(right:3mm) |- (a2.input 1);
\draw (i2.east) -- +(right:3mm) |- (a2.input 2);
\draw (a1.output) -- +(right:3mm) |- (o.input 1);
\draw (a2.output) -- +(right:3mm) |- (o.input 2);
\draw (o.output) -- +(right:3mm);
\end{tikzpicture}
```

TikZ Library `circuits.logic.US`

```
\usetikzlibrary{circuits.logic.US} % LATEX and plain TEX
\usetikzlibrary[circuits.logic.US] % ConTeXt
```

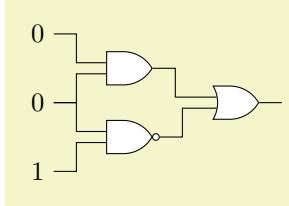
This library provides graphics showing “American” logic gates. It defines the following key:

/tikz/circuit logic US (no value)

This style calls `circuit logic` and installs US-like graphics for the logical symbols like `and gate`. For instance, it says

```
set and gate graphic = and gate US graphic
```

Here is an example:



```
\usetikzlibrary {circuits.logic.CDH}
\begin{tikzpicture}[circuit logic CDH,
    tiny circuit symbols,
    every circuit symbol/.style={
        fill=white,draw}]
\matrix[column sep=7mm]
{
\node (i0) {0}; & & \\
& \node [and gate] (a1) {};& & \\
\node (i1) {0}; & & \node [or gate] (o) {};& \\
& \node [nand gate] (a2) {};& & \\
\node (i2) {1}; & & & \\
};
\draw (i0.east) -- +(right:3mm) |- (a1.input 1);
\draw (i1.east) -- +(right:3mm) |- (a1.input 2);
\draw (i1.east) -- +(right:3mm) |- (a2.input 1);
\draw (i2.east) -- +(right:3mm) |- (a2.input 2);
\draw (a1.output) -- +(right:3mm) |- (o.input 1);
\draw (a2.output) -- +(right:3mm) |- (o.input 2);
\draw (o.output) -- +(right:3mm);
\end{tikzpicture}
```

TikZ Library `circuits.logic.CDH`

```
\usetikzlibrary{circuits.logic.CDH} % LATEX and plain TEX
\usetikzlibrary[circuits.logic.CDH] % ConTeXt
```

This library provides graphics based on the logic symbols used in A. Croft, R. Davidson, and M. Hargreaves (1992), *Engineering Mathematics*, Addison-Wesley, 82–95. They are identical to the US-style symbols, except for the and- and nand-gates.

/tikz/circuit logic CDH (no value)

This key calls `circuit logic US` and installs the two special and- and nand-gates, that is, it uses `set and gate graphic` with `and gate CDH graphic` and likewise for nand-gates.

Inside `circuit logic XYZ` scopes, you can now use the keys shown in Section 50.3.2. We have a more detailed look at one of them, all the other work the same way:

/tikz/and gate (no value)

This key should be passed to a `node` command. It will cause the node to “look like” an `and gate`, where the exact appearance of the gate is dictated by the which circuit environment is used. To further configure the appearance of the `and gate`, see Section 50.2.6.



```
\usetikzlibrary {circuits.logic.IEC}
\tikz [circuit logic IEC] \node [and gate] {$A$};
```



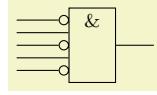
```
\usetikzlibrary {circuits.logic.US}
\tikz [circuit logic US]
{
\node [and gate,point down] {$A$};
\node [and gate,point down,info=center:$A$] at (1,0) {};
}
```

Inputs. Multiple inputs can be specified for a logic gate (provided they support multiple inputs: a not gate – also known as an inverter – does not). However, there is an upper limit for the number of inputs which has been set to 1024, which should be *way* more than would ever be needed.

The following key is used to configure the inputs. It is available only inside a `circuit logic` environment.

/tikz/inputs=(*input list*) (no default, initially `{normal,normal}`)

If a gate has n inputs, the `input list` should consist of n letters, each being `i` for “inverted” or `n` for “normal”. Inverted gates will be indicated by a little circle. In any case the anchors for the inputs will be set up appropriately, numbered from top to bottom `input 1`, `input 2`, ... and so on. If the gate only supports one input the anchor is simply called `input` with no numerical index.



```
\usetikzlibrary {circuits.logic.IEC}
\begin{tikzpicture}[circuit logic IEC]
\node[and gate,inputs={inini}] (A) {};
\foreach \a in {1,...,5}
\draw (A.input \a -| -1,0) -- (A.input \a);
\draw (A.output) -- +(right:5mm);
\end{tikzpicture}
```

(This key is just a shorthand for `logic gate inputs`, described in detail on page 465. There you will also find descriptions of how to configure the size of the inverted circles and the way the symbol size increases when there are too many inputs.)

Output. Every logic gate has one anchor called `output`.

50.3.2 Symbols: The Gates

The following table shows which symbols are declared by the main `circuits.logic` library and their appearance in the different sublibraries.

Key	Appearance inside circuit logic IEC	Appearance inside circuit logic US	Appearance inside circuit logic CDH
<code>/tikz/and gate</code>			
<code>/tikz/nand gate</code>			
<code>/tikz/or gate</code>			
<code>/tikz/nor gate</code>			
<code>/tikz/xor gate</code>			
<code>/tikz/xnor gate</code>			
<code>/tikz/not gate</code>			
<code>/tikz/buffer gate</code>			

50.3.3 Implementation: The Logic Gates Shape Library

The previous sections described the TikZ interface for creating logical circuits. In this section we take a closer look at the underlying PGF libraries.

Just as there are several TikZ circuit libraries, there are two underlying PGF shape libraries, one for creating US-style gates and one for IEC-style gates. These libraries define *shapes* only. It is the job of the circuit libraries to “theme” them so that they “look nice”. However, in principle, you can also use these shapes directly.

Let us begin with the base library that defines the handling of inputs.

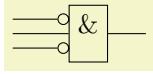
TikZ Library `shapes.gates.logic`

```
\usepgflibrary{shapes.gates.logic} % LATEX and plain TEX and pure pgf
\usepgflibrary[shapes.gates.logic] % ConTEXt and pure pgf
\usetikzlibrary{shapes.gates.logic} % LATEX and plain TEX when using TikZ
\usetikzlibrary[shapes.gates.logic] % ConTEXt when using TikZ
```

This library defines common keys used by all logical gate shapes.

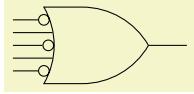
/pgf/logic gate inputs=<input list> (no default, initially {normal,normal})

Specify the inputs for the logic gate. The keyword `inverted` indicates an inverted input which will mean PGF will draw a circle attached to the main shape of the logic gate. Any keyword that is not `inverted` will be treated as a “normal” or “non-inverted” input (however, for readability, you may wish to use `normal` or `non-inverted`), and PGF will not draw the circle. In both cases the anchors for the inputs will be set up appropriately, numbered from top to bottom `input 1`, `input 2`, ... and so on. If the gate only supports one input the anchor is simply called `input` with no numerical index.



```
\usetikzlibrary {circuits.logic.IEC}
\begin{tikzpicture} [minimum height=0.75cm]
\node[and gate IEC, draw, logic gate inputs={inverted, normal, inverted}] (A) {};
\foreach \a in {1,...,3}
\draw (A.input \a |- -1,0) -- (A.input \a);
\draw (A.output) -- ([xshift=0.5cm]A.output);
\end{tikzpicture}
```

For multiple inputs it may be somewhat unwieldy to specify a long list, thus, the following “short-hand” is permitted (this is an extension of ideas due to Jürgen Werber and Christoph Bartoschek): Using `i` for inverted and `n` for normal inputs, <input list> can be specified *without the commas*. So, for example, `ini` is equivalent to `inverted, normal, inverted`.



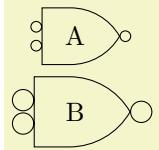
```
\usetikzlibrary {circuits.logic.US}
\begin{tikzpicture} [minimum height=0.75cm]
\node[nor gate US, draw,logic gate inputs=iniini] (A) {};
\foreach \a in {1,...,5}
\draw (A.input \a |- -1,0) -- (A.input \a);
\draw (A.output) -- ([xshift=0.5cm]A.output);
\end{tikzpicture}
```

The height of the gate may be increased to accommodate the number of inputs. In fact, it depends on three variables: n , the number of inputs, r , the radius of the circle used to indicate an inverted input and s , the distance between the centers of the inputs. The default height is then calculated according to the expression $(n + 1) \times \max(2r, s)$. This then may be increased to accommodate the node contents or any minimum size specifications.

The radius of the inverted input circle and the distance between the centers of the inputs can be customized using the following keys:

/pgf/logic gate inverted radius=<length> (no default, initially 2pt)

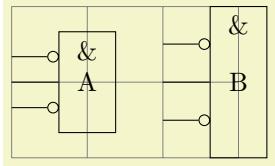
Set the radius of the circle that is used to indicate inverted inputs. This is also the radius of the circle used for the inverted output of the `nand`, `nor`, `xnor` and `not` gates.



```
\usetikzlibrary {circuits.logic.CDH}
\begin{tikzpicture} [minimum height=0.75cm]
\tikzset{every node/.style={shape=nand gate CDH, draw, logic gate inputs=ii}}
\node[logic gate inverted radius=2pt] {A};
\node[logic gate inverted radius=4pt] at (0,-1) {B};
\end{tikzpicture}
```

/pgf/logic gate input sep=<length> (no default, initially .125cm)

Set the distance between the *centers* of the inputs to the logic gate.



```
\usetikzlibrary {circuits.logic.IEC}
\begin{tikzpicture} [minimum size=0.75cm]
  \draw [help lines] grid (3,2);
  \tikzset{every node/.style={shape=and gate IEC, draw, logic gate inputs=ini}}
  \node[logic gate input sep=0.33333cm] at (1,1)(A){};
  \node[logic gate input sep=0.5cm] at (3,1)(B){};
  \foreach \a in {1,...,3}
    \draw (A.input \a |- 0,0) -- (A.input \a)
          (B.input \a |- 2,0) -- (B.input \a);
\end{tikzpicture}
```

PGF will increase the size of the logic gate to accommodate the number of inputs, and the size of the inverted radius and the separation between the inputs. However with all shapes in this library, any increase in size (including any minimum size requirements) will be applied so that the default aspect ratio is unaltered. This means that changing the height will change the width and vice versa.

50.3.4 Implementation: The US-Style Logic Gates Shape Library

TikZ Library `shapes.gates.logic.US`

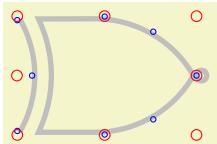
```
\usepgflibrary{shapes.gates.logic.US} % LATEX and plain TEX and pure pgf
\usepgflibrary[shapes.gates.logic.US] % ConTeXt and pure pgf
\usetikzlibrary{shapes.gates.logic.US} % LATEX and plain TEX when using TikZ
\usetikzlibrary[shapes.gates.logic.US] % ConTeXt when using TikZ
```

This library provides “American” logic gate shapes whose names are suffixed with the identifier `US`. Additionally, alternative `and` and `nand` gates are provided which are based on the logic symbols used in A. Croft, R. Davidson, and M. Hargreaves (1992), *Engineering Mathematics*, Addison-Wesley, 82–95. These two shapes are suffixed with `CDH`.

The “compass point” anchors apply to the main part of the shape and do not include any inverted inputs or outputs. This library provides an additional feature to facilitate the relative positioning of logic gates:

`/pgf/logic gate anchors use bounding box=<boolean>` (no default, initially `false`)

When set to `true` this key will ensure that the compass point anchors use the bounding rectangle of the main shape, which, ignore any inverted inputs or outputs, but includes any `outer sep`. This *only* affects the compass point anchors and is not set on a shape by shape basis: whether the bounding box is used is determined by value of this key when the anchor is accessed.



```
\usetikzlibrary {circuits.logic.US}
\begin{tikzpicture} [minimum height=1.5cm]
  \node[xnor gate US, draw, gray!50,line width=2pt] (A){};
  \foreach \x/\y/\z in {false/blue/1pt, true/red/2pt}
    \foreach \a in {north, south, east, west, north east,
      south east, north west, south west}
      \draw[logic gate anchors use bounding box=\x, color=\y]
        (A.\a) circle(\z);
\end{tikzpicture}
```

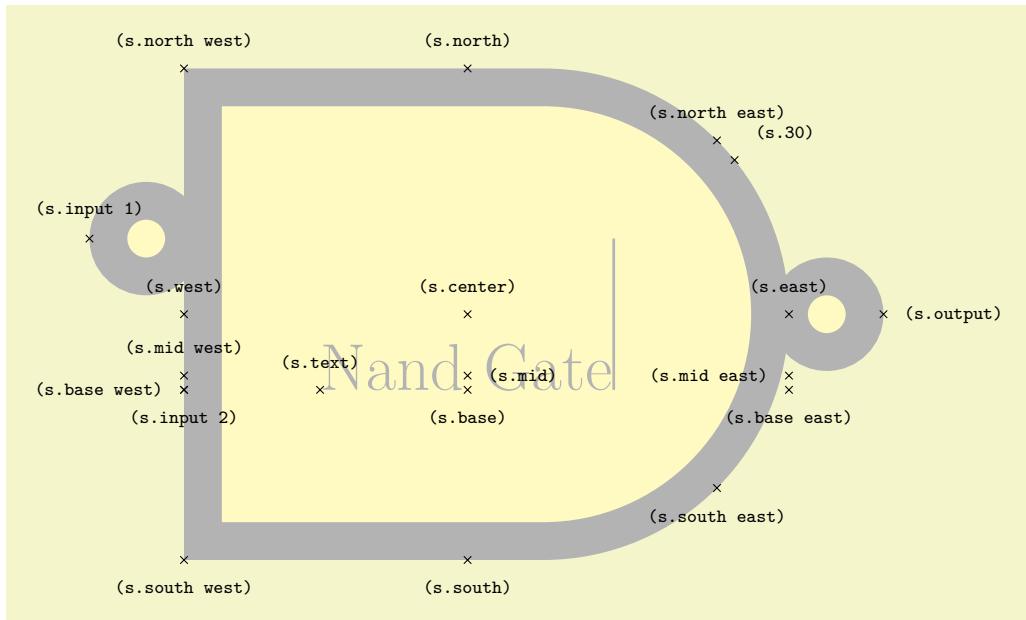
The library defines a number of shapes. For each shape the allowed number of inputs is also shown:

- `and gate US`, two or more inputs
- `and gate CDH`, two or more inputs
- `nand gate US`, two or more inputs
- `nand gate CDH`, two or more inputs
- `or gate US`, two or more inputs
- `nor gate US`, two or more Inputs
- `xor gate US`, two inputs
- `xnor gate US`, two inputs
- `not gate US`, one input
- `buffer gate US`, one input

In the following, we only have a detailed look at the anchors defined by one of them. We choose the `nand` gate US because it shows all the “interesting” anchors.

Shape nand gate US

This shape is a nand gate, which supports two or more inputs. If less than two inputs are specified an error will result. The anchors for this gate with two non-inverted inputs (using the normal compass point anchors) are shown below. Anchor 30 is an example of a border anchor.



```

\usetikzlibrary {circuits.logic.US}
\Huge
\begin{tikzpicture}
  \node [name=s,shape=nand gate US,shape example, inner sep=0cm,
  logic gate inputs={in}, 
  logic gate inverted radius=.5cm] {Nand Gate\vrule width1pt height2cm};
  \foreach \anchor/\placement in
  {center/above, text/above, 30/above right,
  mid/right, mid east/left, mid west/above,
  base/below, base east/below, base west/left,
  north/above, south/below, east/above, west/above,
  north east/above, south east/below, south west/below, north west/above,
  output/right, input 1/above, input 2/below}
  \draw [shift=(s.\anchor)] plot [mark=x] coordinates{(0,0)}
    node[\placement] {\scriptsize\sffamily\texttt{(s.\anchor)}};
\end{tikzpicture}

```

(For the definition of the `shape example` style, see Section 72.)

50.3.5 Implementation: The IEC-Style Logic Gates Shape Library

TikZ Library shapes.gates.logic.IEC

```
\usepgflibrary{shapes.gates.logic.IEC} % LATEX and plain TEX and pure pgf  
\usepgflibrary[shapes.gates.logic.IEC] % ConTeXt and pure pgf  
\usetikzlibrary{shapes.gates.logic.IEC} % LATEX and plain TEX when using TikZ  
\usetikzlibrary[shapes.gates.logic.IEC] % ConTeXt when using TikZ
```

This library provides rectangular logic gate shapes. These shapes are suffixed with IEC as they are based on gates recommended by the International Electrotechnical Commission.

By default each gate is drawn with a symbol, `&` for `and` and `nand` gates, ≥ 1 for `or` and `nor` gates, 1 for `not` and `buffer` gates, and `= 1` for `xor` and `xnor` gates. These symbols are drawn automatically (internally they are drawn using the “foreground” path), and are not strictly speaking part of the node contents. However, the gate is enlarged to make sure the symbols are within the border of the node. It is possible to change the symbols and their position within the node using the following keys:

`/pgf/and gate IEC symbol=<text>` (no default, initially `\char`&`)

Set the symbol for the `and` gate. Note that if the node is filled, this color will be used for the symbol, making it invisible, so it will be necessary set `<text>` to something like `\color{black}\char`&`. Alternatively, the `logic gate IEC symbol color` key can be used to set the color of all symbols simultaneously.

In TikZ, when the `use IEC style logic gates` key has been used, this key can be replaced by `and gate symbol`.

`/pgf/nand gate IEC symbol=<text>` (no default, initially `\char`&`)

Set the symbol for the `nand` gate. In TikZ, when the `use IEC style logic gates` key has been used, this key can be replaced by `nand gate symbol`.

`/pgf/or gate IEC symbol=<text>` (no default, initially `\geq1$`)

Set the symbol for the `or` gate. In TikZ, when the `use IEC style logic gates` key has been used, this key can be replaced by `or gate symbol`.

`/pgf/nor gate IEC symbol=<text>` (no default, initially `\geq1$`)

Set the symbol for the `nor` gate. In TikZ, when the `use IEC style logic gates` key has been used, this key can be replaced by `nor gate symbol`.

`/pgf/xor gate IEC symbol=<text>` (no default, initially `\$=1\$`)

Set the symbol for the `xor` gate. Note the necessity for braces, as the symbol contains `=`. In TikZ, when the `use IEC style logic gates` key has been used, this key can be replaced by `xor gate symbol`.

`/pgf/xnor gate IEC symbol=<text>` (no default, initially `\$=1\$`)

Set the symbol for the `xnor` gate. In TikZ, when the `use IEC style logic gates` key has been used, this key can be replaced by `xnor gate symbol`.

`/pgf/not gate IEC symbol=<text>` (no default, initially `1`)

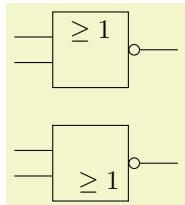
Set the symbol for the `not` gate. In TikZ, when the `use IEC style logic gates` key has been used, this key can be replaced by `not gate symbol`.

`/pgf/buffer gate IEC symbol=<text>` (no default, initially `1`)

Set the symbol for the `buffer` gate. In TikZ, when the `use IEC style logic gates` key has been used, this key can be replaced by `buffer gate symbol`.

`/pgf/logic gate IEC symbol align=<align>` (no default, initially `top`)

Set the alignment of the logic gate symbol (in TikZ, when the `use IEC style logic gates` key has been used, IEC can be omitted). The specification in `<align>` is a comma separated list from `top`, `bottom`, `left` or `right`. The distance between the border of the node and the outer edge of the symbol is determined by the values of the `inner xsep` and `inner ysep`.



```
\usetikzlibrary {shapes.gates.logic.IEC}
\begin{tikzpicture}[minimum size=1cm, use IEC style logic gates]
\tikzset{every node/.style={nor gate, draw}}
\node (A) at (0,1.5) {};
\node [logic gate symbol align={bottom, right}] (B) at (0,0) {};
\foreach \g in {A, B} {
\foreach \i in {1,2}
\draw ([xshift=-0.5cm]\g.input \i) -- (\g.input \i);
\draw (\g.output) -- ([xshift=0.5cm]\g.output);
}
```

`/pgf/logic gate IEC symbol color=<color>` (no default)

This key sets the color for all symbols simultaneously. This color can be overridden on a case by case basis by specifying a color when setting the symbol text.

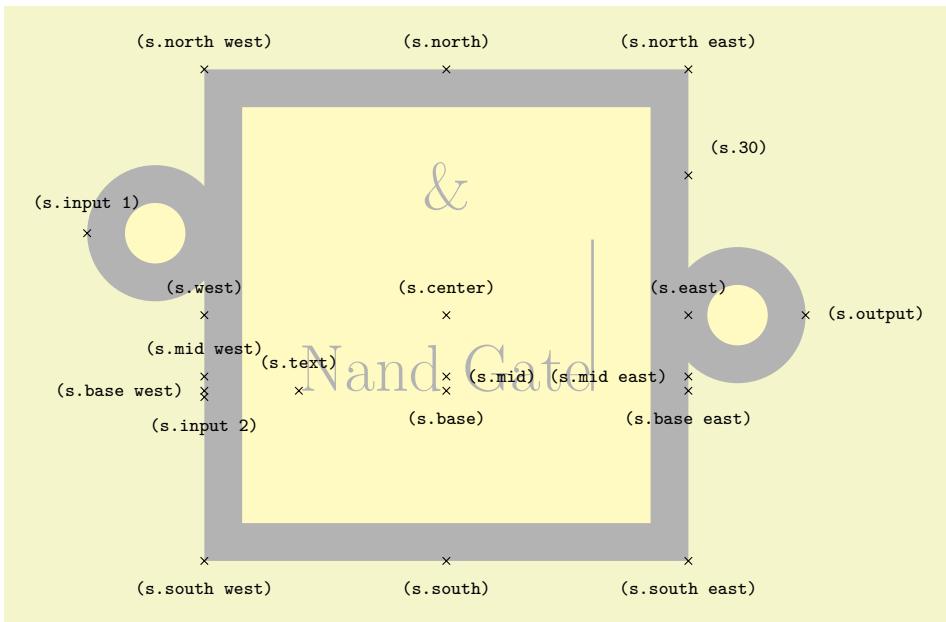
The library defines the following shapes:

- `and gate IEC`, two or more inputs
- `nand gate IEC`, two or more inputs
- `or gate IEC`, two or more inputs
- `nor gate IEC`, two or more inputs
- `xor gate IEC`, two inputs
- `xnor gate IEC`, two inputs
- `not gate IEC`, one input
- `buffer gate IEC`, one input

Again, we only have a look at the nand-gate in more detail:

Shape `nand gate IEC`

This shape is a nand gate. It supports two or more inputs. If less than two inputs are specified an error will result. The anchors for this gate with two inverted inputs are shown below. Anchor 30 is an example of a border anchor.



```
\usetikzlibrary {circuits.logic.IEC}
\Huge
\begin{tikzpicture}
\node [name=s,shape=nand gate IEC ,shape example, inner xsep=1cm, inner ysep=1cm,
minimum height=6cm, nand gate IEC symbol=\color{black!30}\char`'&,
logic gate inputs={in},
logic gate inverted radius=0.65cm]
{Nand Gate \vrule width1pt height2cm};
\foreach \anchor/\placement in
{center/above, text/above, 30/above right,
mid/right, mid east/left, mid west/above,
base/below, base east/below, base west/left,
north/above, south/below, east/above, west/above,
north east/above, south east/below, south west/below, north west/above,
output/right, input 1/above, input 2/below}
\draw[shift=(s.\anchor)] plot [mark=x] coordinates{(0,0)}
node[\placement]{\scriptsize\tt(s.\anchor)};
\end{tikzpicture}
```

50.4 Electrical Engineering Circuits

50.4.1 Overview

An *electrical engineering circuit* contains symbols like resistors or capacitors or voltage sources and annotations like the two arrows pointing toward an element whose behaviour is light dependent. The

electrical engineering libraries, abbreviated ee-libraries, provide such symbols and annotations.

Just as for logical gates, there are different ways of drawing ee-symbols. Currently, there is one main library for drawing circuits, which uses the graphics from the International Electrotechnical Commission, but you can add your own libs. This is why, just as for logical gates, there is a base library and more specific libraries.

TikZ Library `circuits.ee`

```
\usetikzlibrary{circuits.ee} % LATEX and plain TEX  
\usetikzlibrary[circuits.ee] % ConTEXt
```

This library declares the ee symbols, but (mostly) does not provide the symbol graphics, which is left to the sublibraries. Just like the logical gates library, a key is defined that is normally only used internally:

```
/tikz/circuit ee
```

(no value)

This style calls the keys `circuit` (which internally calls `every circuit` and the following style:

```
/tikz/every circuit ee
```

(style, no value)

Use this key to configure the appearance of logical circuits.

The library also declares some standard annotations and units.

As for logical circuits, to draw a circuit the first step is to include a library containing the symbols graphics. Currently, you have to include `circuits.ee.IEC`.

TikZ Library `circuits.ee.IEC`

```
\usetikzlibrary{circuits.ee.IEC} % LATEX and plain TEX  
\usetikzlibrary[circuits.ee.IEC] % ConTEXt
```

When this library is loaded, you can use the following style:

```
/tikz/circuit ee IEC
```

(no value)

This style calls `circuit ee` and installs the IEC-like graphics for the logical symbols like `resistor`.

Inside the `circuit ee IEC` scope, you can now use the keys for symbols, units, and annotations listed in the later sections. We have a more detailed look at one of each of them, all the others work the same way.

Let us start with an example of a symbol: the resistor symbol. The other predefined symbols are listed in Section 50.4.2 and later sections.

```
/tikz/resistor=<options>
```

(no default)

This key should be used with a `node` path command or with the `to` path command.

Using the Key with Normal Nodes. When used with a node, it will cause this node to “look like” a resistor (by default, in the IEC library, this is just a simple rectangle).



```
\usetikzlibrary {circuits.ee.IEC}  
\tikz [circuit ee IEC]  
  \node [resistor] {};
```

Unlike normal nodes, a resistor node generally should not take any text (as in `node [resistor] {foo}`). Instead, the labeling of resistors should be done using the `label`, `info` and `ohm` options.



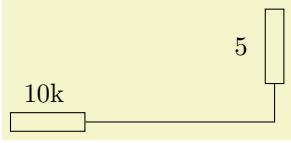
```
\usetikzlibrary {circuits.ee.IEC}  
\tikz [circuit ee IEC]  
  \node [resistor,ohm=5] {};
```

The `<options>` make no real sense when the `resistor` option is used with a normal node, you can just as well give them to the `node` itself. Thus, the following has the same effect as the above example:



```
\usetikzlibrary {circuits.ee.IEC}  
\tikz [circuit ee IEC]  
  \node [resistor={ohm=5}] {};
```

In a circuit, you will often wish to rotate elements. For this, the options `point up`, `point down`, `point left` or `point right` may be especially useful. They are just shorthands for appropriate rotations like `rotate=90`.

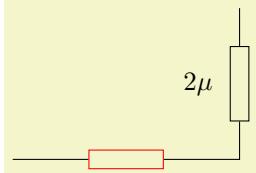


```
\usetikzlibrary {circuits.ee.IEC}
\begin{tikz} [circuit ee IEC]
\node (R1) [resistor,point up,ohm=5] at (3,1) {};
\node (R2) [resistor,ohm=10k] at (0,0) {};
\draw (R2) -| (R1);
\end{tikz}
```

Using the Key on a To Path. When the `resistor` key is used on a `to` path inside a `circuit ee IEC`, the `circuit handle symbol` key is called internally. This has a whole bunch of effects:

1. The path currently being constructed is cut up to make place for a node.
2. This node will be a `resistor` node that is rotated so that it points “along” the path (unless an option like `shift only` or an extra rotation is used to change this).
3. The `(options)` passed to the `resistor` key are passed on to the node.
4. The `(options)` are pre-parsed to identify a `pos` key or a key like `at start` or `midway`. These keys are used to determine where on the `to` path the node will lie.

Since the `(options)` of the `resistor` key are passed on to the resistor node on the path, you can use it to add labels to the node. Here is a simple example:



```
\usetikzlibrary {circuits.ee.IEC}
\begin{tikz} [circuit ee IEC]
\draw (0,0) to [resistor=red] (3,0)
to [resistor={ohm=2\mu}] (3,2);
\end{tikz}
```

You can add multiple labels to a resistor and you can have multiple resistors (or other elements) on a single path.

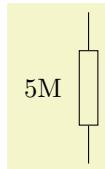
Inputs, Outputs, and Anchors. Like the logical gates, all ee-symbols have an `input` and an `output` anchor. Special-purpose-nodes may have even more anchors of this type. Furthermore, the ee-symbols-nodes also have four standard compass direction anchors.

Changing the Appearance. To configure the appearance of all `resistors`, see Section 50.2.6. You can use the `(options)` to locally change the appearance of a single resistor.

Let us now have a look at an example of a unit: the Ohm unit. The other predefined units are listed in Section 50.4.7.

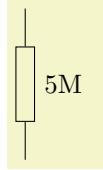
`/tikz/ohm=<value>` (no default)

This key is used to add an `info` label to a node with a special text: `$\mathsf{value}\Omega$`. In other words, the `ohm` key can only be used with the options of a node and, when used, it will cause the `(value)` to be placed next to the node, followed by Ω . Since the `(value)` is typeset inside a `\mathsf{value}` command, when you write `ohm=5k` you get 5k , `ohm=5p` yields 5p , and `ohm=5.6\cdot 10^{2}\mu` yields $5.6 \cdot 10^2 \mu$.



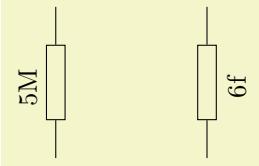
```
\usetikzlibrary {circuits.ee.IEC}
\begin{tikz} [circuit ee IEC]
\draw (0,0) to [resistor={ohm=5M}] (0,2);
\end{tikz}
```

Instead of `ohm` you can also use `ohm'`, which places the label on the other side.



```
\usetikzlibrary {circuits.ee.IEC}
\begin{tikzpicture} [circuit ee IEC]
\draw (0,0) to [resistor={ohm'=5M}] (0,2);
\end{tikzpicture}
```

Finally, there are also keys `ohm sloped` and `ohm' sloped` for having the info label rotate together with the main node.



```
\usetikzlibrary {circuits.ee.IEC}
\begin{tikzpicture} [circuit ee IEC]
\draw (0,0) to [resistor={ohm sloped=5M}] (0,2);
(2,0) to [resistor={ohm' sloped=6f}] (2,2);
\end{tikzpicture}
```

You can configure the appearance of an Ohm info label using the key `every ohm`.

Finally, let us have a look at an annotation: the `light emitting` annotation. The other predefined units are listed in Section 50.4.8.

`/tikz/light emitting=(options)`

(no default)

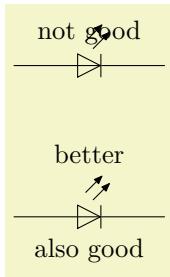
Like a unit, an annotation should be given as an additional option to a node. It causes some drawings (in this case, two parallel lines) to be placed next to the node.



```
\usetikzlibrary {circuits.ee.IEC}
\begin{tikzpicture} [circuit ee IEC]
\draw (0,0) to [diode=light emitting] (2,0);
\end{tikzpicture}
```

The `(options)` can be used for three different things:

1. You can use keys like `red` to change the appearance of this annotation, locally.
2. You can use keys like `<-` or `- latex` to change the direction and kinds of arrows used in the annotation.
3. You can use info labels like `ohm=5` or `info=foo` inside the `(options)`. These info labels will be added to the main node (not to the annotation itself), but the label distance will have been changed to accommodate for the space taken up by the annotation.



```
\usetikzlibrary {circuits.ee.IEC}
\begin{tikzpicture} [circuit ee IEC]
\{
  \draw (0,2) to [diode={light emitting,info=not good}] (2,2);
  \draw (0,0) to [diode={light emitting=info=better}, info'={also good}] (2,0);
\}
```

In addition to `light emitting` there is also a key called `light emitting'`, which simply places the annotation on the other side of the node.

You can configure the appearance of annotations in three ways:

- You can set the `every circuit annotation` style.
- You can set the `every light emitting` style.
- You can set the following key:

`/tikz/annotation arrow`

(style, no value)

This style should set the default > arrow to some nice value.

50.4.2 Symbols: Indicating Current Directions

There are two symbols for indicating current directions. These symbols are defined directly inside `circuit ee`.

Key	Appearance
<code>/tikz/current direction</code>	
<code>/tikz/current direction'</code>	

The examples have been produced by (in essence) `\draw (0,0) to[<symbol name>] (3,0);`.

50.4.3 Symbols: Basic Elements

The following table show basic symbols as they are depicted inside the `circuit ee` IEC environment. To install one of alternate graphics, you have to say `set <symbol name> graphic=var <symbol name>` IEC `graphic`.

Key	Appearance	Alternate appearance
<code>/tikz/resistor</code>		
<code>/tikz/inductor</code>		
<code>/tikz/capacitor</code>		
<code>/tikz/battery</code>		
<code>/tikz/bulb</code>		
<code>/tikz/current source</code>		
<code>/tikz/voltage source</code>		
<code>/tikz/ac source</code>		
<code>/tikz/dc source</code>		
<code>/tikz/ground</code>		

50.4.4 Symbols: Diodes

The following table shows diodes as they are depicted inside the `circuit ee` IEC environment.

Key	Appearance	Alternate appearance
<code>/tikz/diode</code>		
<code>/tikz/Zener diode</code>		
<code>/tikz/Schottky diode</code>		
<code>/tikz/tunnel diode</code>		
<code>/tikz/backward diode</code>		
<code>/tikz/breakdown diode</code>		

50.4.5 Symbols: Contacts

The following table shows contacts as they are depicted inside the `circuit ee` IEC environment.

Key	Appearance	Alternate appearance
<code>/tikz/contact</code>		
<code>/tikz/make contact</code>		
<code>/tikz/break contact</code>		

50.4.6 Symbols: Measurement devices

The following table shows measurement devices as they are depicted inside the `circuit ee` IEC environment.

Key	Appearance
<code>/tikz/amperemeter</code>	
<code>/tikz/voltmeter</code>	
<code>/tikz/ohmmeter</code>	

50.4.7 Units

The `circuits.ee` library predefines the following unit keys:

Key	Appearance of 1 unit
<code>/tikz/ampere</code>	1A
<code>/tikz/volt</code>	1V
<code>/tikz/ohm</code>	1
<code>/tikz/siemens</code>	1S
<code>/tikz/henry</code>	1H
<code>/tikz/farad</code>	1F
<code>/tikz/coulomb</code>	1C
<code>/tikz/voltampere</code>	1VA
<code>/tikz/watt</code>	1W
<code>/tikz/hertz</code>	1Hz

50.4.8 Annotations

The `circuits.ee.IEC` library defines the following annotations:

Key	Appearance
<code>/tikz/light emitting</code>	
<code>/tikz/light dependent</code>	
<code>/tikz/direction info</code>	
<code>/tikz/adjustable</code>	

The lines have been produced using, in essence,

```
\draw (0,0) to [resistor=light emitting] (2,0) to [diode=light emitting'] (4,0);
```

and similarly for the other annotations.

50.4.9 Implementation: The EE-Symbols Shape Library

The TikZ libraries depend on two shape libraries, which are included automatically. Usually, you will not need to use these shapes directly.

TikZ Library `shapes.gates.ee`

```
\usepgflibrary{shapes.gates.ee} % LATEX and plain TEX and pure pgf
\usepgflibrary[shapes.gates.ee] % ConTeXt and pure pgf
\usetikzlibrary{shapes.gates.ee} % LATEX and plain TEX when using TikZ
\usetikzlibrary[shapes.gates.ee] % ConTeXt when using TikZ
```

This library defines basic shapes that can be used by all ee-circuit libraries. Currently, it defines the following shapes:

- `rectangle ee`
- `circle ee`
- `direction ee`

Additionally, the library defines the following arrow tip: The `direction ee` arrow tip is basically the same as a `triangle 45` arrow tip with rounded joins.

`direction ee` yields thick and thin

However, unlike normal arrow tips, its size does *not* depend on the current line width. Rather, it depends on the value of its arrow options, which should be set to the desired size. Thus, you should say something like `\pgfsetarrowoptions{direction ee}{5pt}` to set the size of the arrow.

Shape `rectangle ee`

This shape is completely identical to a normal `rectangle`, only there are two additional anchors: The `input` anchor is an alias for the `west` anchor, while the `output` anchor is an alias for the `east` anchor.

Shape `circle ee`

Like the `rectangle ee` shape, only for circles.

Shape `direction ee`

This shape is rather special. It is intended to be used to “turn an arrow tip into a shape”. First, you should set the following key to the name of an arrow tip:

`/pgf/direction ee arrow=(right arrow tip name)` (no default)

The value of this key will be used for the arrow tip depicted in an `direction ee` shape.

When a node of shape `direction ee` is created, several things happen:

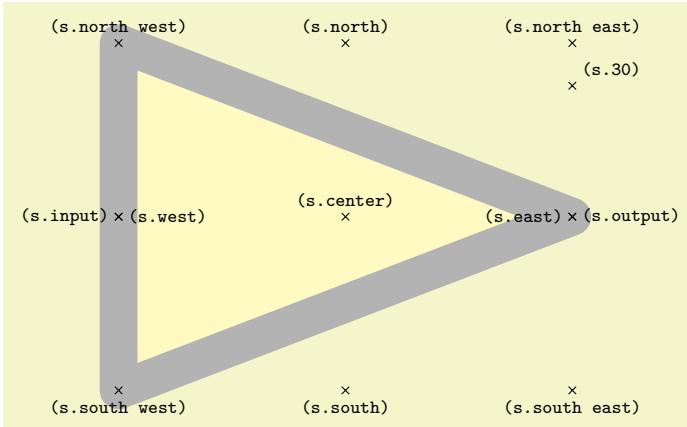
1. The size of the shape is computed according to the following rules: The width of the shape is set up so that the left border of the shape is at the left end of the arrow tip and the right border is at the right end of the arrow tip. These left and right “ends” of the arrow are the tip end and the back end specified by the arrow itself (see Section ?? for details). You usually need not worry about this width setting.

By comparison, the height of the arrow is given by the current setting of `minimum height`. Thus, this key must have been set up correctly to reflect the “real” height of the arrow tip. The reason is that the height of an arrow is not specified when arrows are declared and is, thus, not available, here.

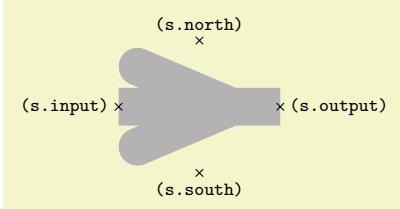
Possibly, the height computation will change in the future to reflect the real height of the arrow, so you should generally set up the `minimum height` to be the same as the real height.

2. A straight line from left to right inside the shape’s boundaries is added to the background path.
3. The arrow tip, pointing right, is drawn before the background path.

The anchors of this shape are just the compass anchors, which lie on a rectangle whose width and height are the above-computed height and width.



```
\usetikzlibrary {circuits.ee.IEC}
\begin{tikzpicture}
\pgfsetarrowoptions{direction ee}{6cm}
\node [name=s,shape=direction ee,shape example,minimun height=0.7654*6cm] {};
\foreach \anchor/\placement in
{center/above, 30/above right,
 north/above, south/below, east/left, west/right,
 north east/above, south east/below, south west/below, north west/above,
 input/left,output/right}
\draw[shift=(s.\anchor)] plot[mark=x] coordinates{(0,0)};
\node[\placement] {\scriptsize\tt(s.\anchor)};
```



```
\usetikzlibrary {circuits.ee.IEC}
\begin{tikzpicture}[direction ee arrow=angle 45]
\node[name=s,shape=direction ee,shape example,minimum height=1.75cm] {};
\foreach \anchor/\placement in {north/above, south/below,
                                output/right, input/left}
  \draw[shift=(s.\anchor)] plot[mark=x] coordinates{(0,0)}
    node[\placement]{\scriptsize\textrm{\texttt{(s.\anchor)}}};
\end{tikzpicture}
```

50.4.10 Implementation: The IEC-Style EE-Symbols Shape Library

TikZ Library `shapes.gates.ee.IEC`

```
\usepgflibrary{shapes.gates.ee.IEC} % LATEX and plain TEX and pure pgf
\usepgflibrary[shapes.gates.ee.IEC] % ConTEXt and pure pgf
\usetikzlibrary{shapes.gates.ee.IEC} % LATEX and plain TEX when using TikZ
\usetikzlibrary[shapes.gates.ee.IEC] % ConTEXt when using TikZ
```

This library defines shapes for depicting ee symbols according to the IEC recommendations. These shapes will typically be used in conjunction with the graphic mechanism detailed earlier, but you can also use them directly.

Shape `generic circle IEC`

This shape inherits from `circle ee`, which in turn is just a normal `circle` with additional `input` and `output` anchors at the left and right ends. However, additionally, this shape allows you to specify a path that should be added before the background path using the following key:

`/pgf/generic circle IEC/before background=<code>` (no default)

When a node of shape `generic circle IEC` is created, the current setting of this key is used as the “before background path”. This means that after the circle’s background has been drawn/-filled/whatever, the `<code>` is executed.

When the `<code>` is executed, the coordinate system will have been transformed in such a way that the point `(1pt,0pt)` lies at the right end of the circle and `(0pt,1pt)` lies at the top of the circle. (More precisely, these points will lie exactly on the middle of the radial line.)

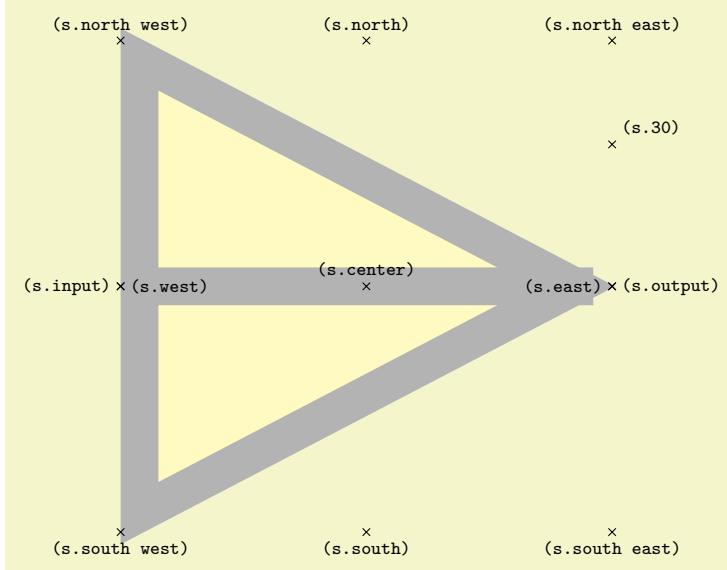
Here is an example of how to use this shape:



```
\usetikzlibrary {circuits.ee.IEC}
\tikz \node [generic circle IEC,
            /pgf/generic circle IEC/before background={%
              \pgfpathmoveto{\pgfpointorigin}%
              \pgfpathlineto{\pgfpoint{1pt}{0pt}}%
              \pgfpathlineto{\pgfpoint{0pt}{1pt}}%
              \pgfpathlineto{\pgfpoint{-0.5pt}{-0.5pt}}%
              \pgfusepathqstroke%
            },
            draw] {Hello world};
```

Shape `generic diode IEC`

This shape is used to depict diodes. The main shape is taken up by a “right pointing” triangle. The anchors are positioned on the border of a rectangle around the diode, see the below example. The diode’s size is based on the current settings of `minimum width` and `minimum height`.



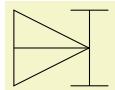
```
\usetikzlibrary {circuits.ee.IEC}
\begin{tikzpicture}
\node [name=s,shape=generic diode IEC,shape example,minimum size=6cm] {};
\foreach \anchor/\placement in
{center/above, 30/above right,
 north/above, south/below, east/left, west/right,
 north east/above, south east/below, south west/below, north west/above,
 input/left,output/right}
\draw[shift=(s.\anchor)] plot[mark=x] coordinates{(0,0)};
\node[\placement] {\scriptsize\tt(s.\anchor)};
\end{tikzpicture}
```

This shape, like the `generic circle` IEC shape, is generic in the sense that there is a special key that is used for the before background drawings:

`/pgf/generic diode IEC/before background=<code>` (no default)

Similarly to the `generic circle` IEC shape, when a node of shape `generic diode IEC` is created, the current setting of this key is used as the “before background path”. When the `<code>` is executed, the coordinate system will have been transformed in such a way that the origin is at the “tip” of the diode’s triangle, the point `(0pt, 1pt)` is exactly half the diode’s height above this origin, and the point `(1pt, 0pt)` is half the diode’s height to the right of the origin.

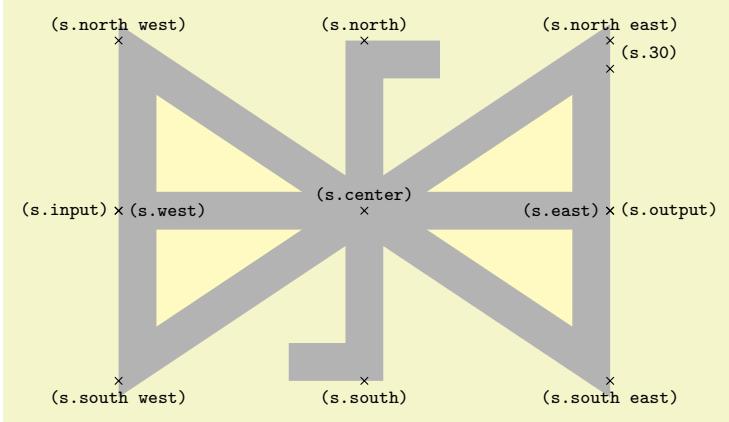
The idea is that you use this key to draw different kinds of diode endings.



```
\usetikzlibrary {circuits.ee.IEC}
\tikz \node [minimum size=1cm,generic diode IEC,
/pgf/generic diode IEC/before background={%
\pgfpathmoveto{\pgfqpoint{-5pt}{-1pt}}%
\pgfpathlineto{\pgfqpoint{.5pt}{-1pt}}%
\pgfpathmoveto{\pgfqpoint{0pt}{-1pt}}%
\pgfpathlineto{\pgfqpoint{0pt}{1pt}}%
\pgfpathmoveto{\pgfqpoint{-5pt}{1pt}}%
\pgfpathlineto{\pgfqpoint{.5pt}{1pt}}%
\pgfusepathqstroke%
},
draw] {};
```

Shape `breakdown diode IEC`

This shape is used to depict a bidirectional breakdown diode. The diode’s size is based on the current settings of `minimum width` and `minimum height`.

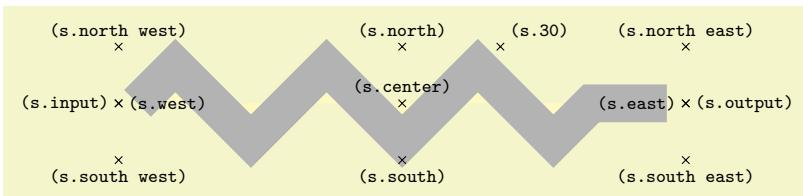


```
\usetikzlibrary {circuits.ee.IEC}
\begin{tikzpicture}
\node[name=s,shape=breakdown diode IEC,shape example,minimum width=6cm,minimum height=4cm] {};
\foreach \anchor/\placement in
{center/above, 30/above right,
 north/above, south/below, east/left, west/right,
 north east/above, south east/below, south west/below, north west/above,
 input/left,output/right}
\draw[shift=(s.\anchor)] plot[mark=x] coordinates{(0,0)}
node[\placement] {\scriptsize\textrm{\texttt{(s.\anchor)}}};
\end{tikzpicture}
```

Shape var resistor IEC

This shape is used to depict a variant version of a resistor. Its size is computed as for a rectangle (thus, its size depends things like the `minimum height`). Then, inside this rectangle, a background path is set up according to the following rule: Starting from the left end, zigzag segments are added to the path. Each segment consists of a line at a 45 degree angle going up to the top of the rectangle, then going down to the bottom, then going up to mid height of the node. As many segments as possible are put inside as possible. The last segment is then connected to the output anchor via a straight line.

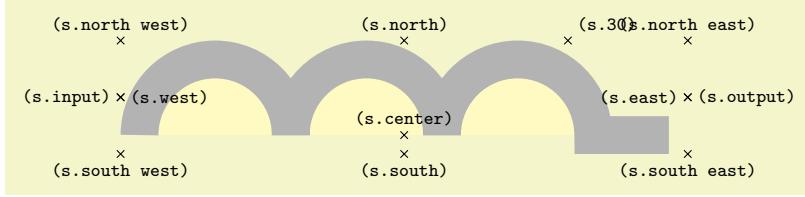
All of this means that, in general, the shape should be much wider than high.



```
\usetikzlibrary {circuits.ee.IEC}
\begin{tikzpicture}
\node[name=s,shape=var resistor IEC,shape example,minimum width=7cm,minimum height=1cm] {};
\foreach \anchor/\placement in
{center/above, 30/above right,
 north/above, south/below, east/left, west/right,
 north east/above, south east/below, south west/below, north west/above,
 input/left,output/right}
\draw[shift=(s.\anchor)] plot[mark=x] coordinates{(0,0)}
node[\placement] {\scriptsize\textrm{\texttt{(s.\anchor)}}};
\end{tikzpicture}
```

Shape inductor IEC

This shape is used to depict an inductor, using a bumpy line. Its size is computed as follows: Any text and `inner sep` are ignored (and should normally not be given). The `minimum height` plus (twice) the `outer ysep` specify the distance between the `north` and `south` anchors, similarly for the `minimum width` plus the `outer xsep` for the `east` and `west`. The bumpy line is drawn starting from the lower left corner to the lower right corner with bumps being half-circles whose height is exactly the `minimum height`. The `center` of the shape is just above the `south` anchor, at a distance of the `outer ysep`.

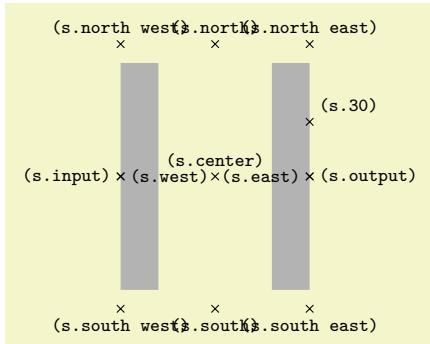


```
\usetikzlibrary {circuits.ee.IEC}
\begin{tikzpicture}
\node [name=s,shape=inductor IEC,shape example,minimum width=7cm,minimum height=1cm] {};
\foreach \anchor/\placement in
{center/above, 30/above right,
north/above, south/below, east/left, west/right,
north east/above, south east/below, south west/below, north west/above,
input/left,output/right}
\draw[shift=(s.\anchor)] plot[mark=x] coordinates{(0,0)}
node[\placement] {\scriptsize\texttt{(s.\anchor)}};
\end{tikzpicture}
```

Just as for a `var resistor IEC`, as many bumps as possible are added and the last bump is connected to the output anchor via a straight line.

Shape `capacitor IEC`

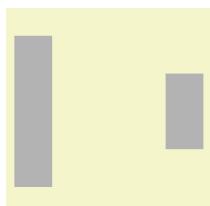
This shape is based on a `rectangle ee`. However, instead of a rectangle as the background path, only the “left and right lines” that make up the rectangle are drawn.



```
\usetikzlibrary {circuits.ee.IEC}
\begin{tikzpicture}
\node [name=s,shape=capacitor IEC,shape example,
minimum width=2cm,minimum height=3cm,inner sep=0pt] {};
\foreach \anchor/\placement in
{center/above, 30/above right,
north/above, south/below, east/left, west/right,
north east/above, south east/below, south west/below, north west/above,
input/left,output/right}
\draw[shift=(s.\anchor)] plot[mark=x] coordinates{(0,0)}
node[\placement] {\scriptsize\texttt{(s.\anchor)}};
\end{tikzpicture}
```

Shape `battery IEC`

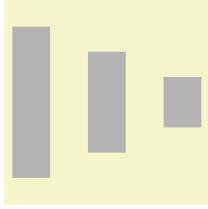
This shape is similar to a `capacitor IEC`, however, the right line is only half the height of the left line.



```
\usetikzlibrary {circuits.ee.IEC}
\tikz \node[shape=battery IEC,shape example,minimum size=2cm,
inner sep=0pt] {};
```

Shape `ground` IEC

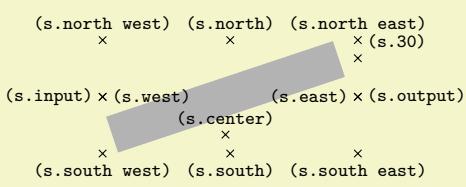
This shape is similar to a `batter` IEC, only three lines of different heights are drawn.



```
\usetikzlibrary {circuits.ee.IEC}
\begin{tikzpicture}
\node [shape=ground IEC,shape example,minimum size=2cm,
inner sep=0pt] {};
\end{tikzpicture}
```

Shape `make contact` IEC

This shape consists of a line going from the lower left corner to the upper right corner. The size and anchors of this shape are computed in the same way as for an `inductor` IEC.



```
\usetikzlibrary {circuits.ee.IEC}
\begin{tikzpicture}
\node [name=s,shape=make contact IEC,shape example,minimum width=3cm,minimum height=1cm] {};
\foreach \anchor/\placement in
{center/above, 30/above right,
 north/above, south/below, east/left, west/right,
 north east/above, south east/below, south west/below, north west/above,
 input/left,output/right}
\draw[shift=(s.\anchor)] plot[mark=x] coordinates{(0,0)};
\node[\placement] {\scriptsize\tt(s.\anchor)};
\end{tikzpicture}
```

Shape `var make contact` IEC

This shape works like `make contact` IEC, only a little circle is added to the path at the lower left corner. The radius of this circle is one twelfth of the width of the node.



```
\usetikzlibrary {circuits.ee.IEC}
\begin{tikzpicture}
\node [shape=var make contact IEC,shape example,
minimum height=1cm,minimum width=3cm,inner sep=0pt] {};
\end{tikzpicture}
```

Shape `break contact` IEC

This shape depicts a contact that can be broken. It works like `make contact` IEC.



```
\usetikzlibrary {circuits.ee.IEC}
\begin{tikzpicture}
\node [shape=break contact IEC,shape example,
minimum height=1cm,minimum width=3cm,inner sep=0pt] {};
\end{tikzpicture}
```

51 Decoration Library

51.1 Overview and Common Options

The decoration libraries define a number of (more or less useful) decorations that can be applied to paths. The usage of decorations is not covered in the present section, please consult Sections 24, which explains how decorations are used in TikZ, and ??, which explains how new decorations can be defined.

The decorations are influenced by a number of parameters that can be set using the `decoration` option. These parameters are typically shared between different decorations. In the following, the general options are documented (they are defined directly in the `decoration` module), special-purpose keys are documented with the decoration that uses it.

Since you are encouraged to use these keys to make your own decorations configurable, it is indicated for each key where the value is stored (so that you can access it). Note that some values are stored in TeX dimension registers while others are stored in macros.

`/pgf/decoration/amplitude=⟨dimension⟩` (no default, initially 2.5pt)

This key determines the “desired height” (or amplitude) of decorations for which this makes sense. For instance, the initial value of 2.5pt means that deforming decorations should deform a path by up to 2.5pt away from the original path.

This key sets the TeX-dimension `\pgfdecorationsegmentamplitude`.

`/pgf/decoration/meta-amplitude=⟨dimension⟩` (no default, initially 2.5pt)

This key determines the amplitude for a meta-decoration.

This key sets the TeX-macro (!) `\pgfmetadecorationsegmentamplitude`.

`/pgf/decoration/segment length=⟨dimension⟩` (no default, initially 10pt)

Many decorations are made up of small segments. This key determines the desired length of such segments.

This key sets the TeX-dimension `\pgfdecorationsegmentlength`.

`/pgf/decoration/meta-segment length=⟨dimension⟩` (no default, initially 1cm)

This determined the length of the meta-segments from which a meta-decoration is made up.

This key sets the TeX-macro (!) `\pgfmetadecorationsegmentlength`.

`/pgf/decoration/angle=⟨degree⟩` (no default, initially 45)

The way some decorations look like depends on a configurable angle. For instance, a `wave` decoration consists of arcs and the opening angle of these arcs is given by the `angle`.

This key sets the TeX-macro `\pgfdecorationsegmentangle`.

`/pgf/decoration/aspect=⟨factor⟩` (no default, initially 0.5)

For some decorations there is a natural aspect ratio. For instance, for a `brace` decoration the aspect ratio determines where the brace point will be.

This key sets the TeX-macro `\pgfdecorationsegmentaspect`.

`/pgf/decoration/start radius=⟨dimension⟩` (no default, initially 2.5pt)

For some decorations there is a natural start radius (of some circle, presumably).

This key stores the value directly inside the key.

`/pgf/decoration/end radius=⟨dimension⟩` (no default, initially 2.5pt)

For some decorations there is a natural end radius (of some circle, presumably).

This key stores the value directly inside the key.

`/pgf/decoration/radius=⟨dimension⟩` (style, no default)

Sets the start and end radius simultaneously.

`/pgf/decoration/path has corners=⟨boolean⟩` (no default, initially `false`)

This is a hint to the decoration code as to whether the path has corners or not. If a path has a sharp corner, setting this option to `true` may result in better rendering of the decoration because the joins of input segments are approached “more carefully” than when this key is set to false. However, if the path is, say, a smooth circle, setting this key to `true` will usually look worse. Most decorations ignore this key, anyway. Internally, it sets the `\TeX`-if `\ifpgfdecoratethascorners`.

51.2 Path Morphing Decorations

TikZ Library `decorations.pathmorphing`

```
\usepgflibrary{decorations.pathmorphing} % \TeX and plain \TeX and pure pgf
\usepgflibrary[decorations.pathmorphing] % ConTeXt and pure pgf
\usetikzlibrary{decorations.pathmorphing} % \TeX and plain \TeX when using TikZ
\usetikzlibrary[decorations.pathmorphing] % ConTeXt when using TikZ
```

A *path morphing decoration* “morphs” or “deforms” the to-be-decorated path. This means that what used to be a straight line might afterwards be a snaking curve and have bumps. However, a line is still a line and path deforming decorations do not change the number of subpaths. For instance, if the path used to consist of two circles and an open arc, the path will, after the decoration process, still consist of two closed subpaths and one open subpath.

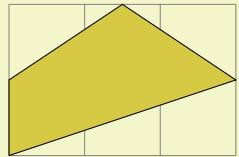
51.2.1 Decorations Producing Straight Line Paths

The following deformations use only straight lines in order to morph the paths.

Decoration `lineto`

This decoration replaces the path by straight lines. For each curve, the path simply goes directly from the start point to the end point. In the following example, the arc actually consists of two subcurves.

This decoration is actually always defined when the decoration module is loaded, but it is documented here for consistency.

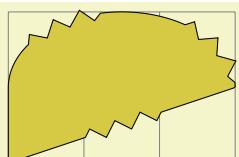


```
\usetikzlibrary {decorations}
\begin{tikzpicture}[decoration=lineto]
\draw [help lines] grid (3,2);
\draw [decorate,fill=yellow!80!black]
(0,0) -- (3,1) arc (0:180:1.5 and 1) -- cycle;
\end{tikzpicture}
```

Decoration `straight zigzag`

This (meta-)decoration decorates the path by alternating between `curveto` and `zigzag` decorations. It always finishes with the `curveto` decoration. The following parameters influence the decoration:

- `amplitude` determines how much the zigzag line raises above and falls below a straight line to the target point.
- `segment length` determines the length of a complete “up-down” cycle.
- `meta-segment length` determines the length of the `curveto` and the `zigzag` decorations.



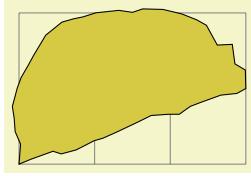
```
\usetikzlibrary {decorations.pathmorphing}
\begin{tikzpicture}[decoration={straight zigzag,meta-segment length=1.1cm}]
\draw [help lines] grid (3,2);
\draw [decorate,fill=yellow!80!black]
(0,0) -- (3,1) arc (0:180:1.5 and 1) -- cycle;
\end{tikzpicture}
```

Decoration `random steps`

This decoration consists of straight line segments. The line segments head towards the target, but each step is randomly shifted a little bit. The following parameters influence the decorations:

- `segment length` determines the basic length of each step.

- **amplitude** The end of each step is perturbed both in x - and in y -direction by two values drawn uniformly from the interval $[-d, d]$, where d is the value of **amplitude**.

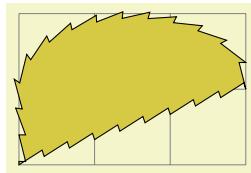


```
\usetikzlibrary {decorations.pathmorphing}
\begin{tikzpicture}
[decoration={random steps,segment length=2mm}]
\draw [help lines] grid (3,2);
\draw [decorate,fill=yellow!80!black]
(0,0) -- (3,1) arc (0:180:1.5 and 1) -- cycle;
\end{tikzpicture}
```

Decoration `saw`

This decoration looks like the blade of a saw. The following parameters influence the decoration:

- **amplitude** determines how much each spike raises above the straight line.
- **segment length** determines the length each spike.

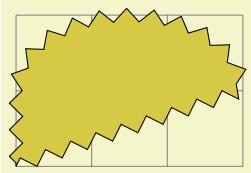


```
\usetikzlibrary {decorations.pathmorphing}
\begin{tikzpicture}[decoration=saw]
\draw [help lines] grid (3,2);
\draw [decorate,fill=yellow!80!black]
(0,0) -- (3,1) arc (0:180:1.5 and 1) -- cycle;
\end{tikzpicture}
```

Decoration `zigzag`

This decoration looks like a zigzag line. The following parameters influence the decoration:

- **amplitude** determines how much the zigzag line raises above and falls below a straight line to the target point.
- **segment length** determines the length of a complete “up-down” cycle.



```
\usetikzlibrary {decorations.pathmorphing}
\begin{tikzpicture}[decoration=zigzag]
\draw [help lines] grid (3,2);
\draw [decorate,fill=yellow!80!black]
(0,0) -- (3,1) arc (0:180:1.5 and 1) -- cycle;
\end{tikzpicture}
```

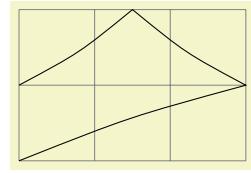
51.2.2 Decorations Producing Curved Line Paths

Decoration `bent`

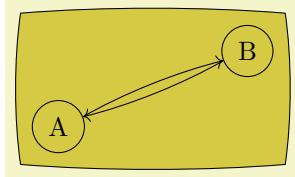
This decoration adds a slightly bent line from the start to the target. The amplitude of the bend is given **amplitude** (an amplitude of zero gives a straight line).

- **amplitude** determines the amplitude of the bend.
- **aspect** determines how tight the bend is. A good value is around 0.3.

Note that this decoration makes only little sense for curves. You should apply it only to straight lines.



```
\usetikzlibrary {decorations.pathmorphing}
\begin{tikzpicture}[decoration=bent]
\draw [help lines] grid (3,2);
\draw [decorate] (0,0) -- (3,1) -- (1.5,2) -- (0,1);
\end{tikzpicture}
```

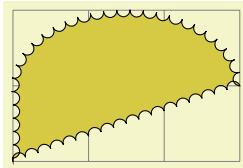


```
\usetikzlibrary {decorations.pathmorphing}
\begin{tikzpicture}[decoration={bent,aspect=.3}]
  \draw [decorate,fill=yellow!80!black] (0,0) rectangle (3.5,2);
  \node[circle,draw] (A) at (.5,.5) {A};
  \node[circle,draw] (B) at (3,1.5) {B};
  \draw[-,decorate] (A) -- (B);
  \draw[-,decorate] (B) -- (A);
\end{tikzpicture}
```

Decoration `bumps`

This decoration replaces the path by little half ellipses. The following parameters influence it.

- `amplitude` determines the height of the half ellipse.
- `segment length` determines the width of the half ellipse.

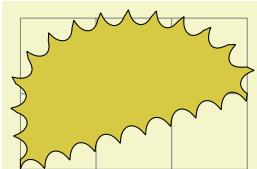


```
\usetikzlibrary {decorations.pathmorphing}
\begin{tikzpicture}[decoration=bumps]
  \draw [help lines] grid (3,2);
  \draw [decorate,fill=yellow!80!black]
    (0,0) -- (3,1) arc (0:180:1.5 and 1) -- cycle;
\end{tikzpicture}
```

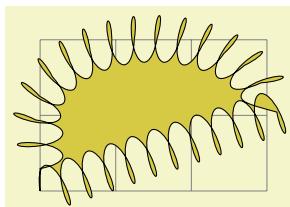
Decoration `coil`

This decoration replaces the path by a coiled line. To understand how this works, imagine a three-dimensional spring. The spring's axis points along the path toward the target. Then, we "view" the spring from a certain angle. If we look "straight from the side" we will see a perfect sine curve, if we look "more from the front" we will see a coil. The following parameters influence the decoration:

- `amplitude` determines how much the coil rises above the path and falls below it. Thus, this is the radius of the coil.
- `segment length` determines the distance between two consecutive "curls". Thus, when the spring is seen "from the side" this will be the wave length of the sine curve.
- `aspect` determines the "viewing direction". A value of 0 means "looking from the side" and a value of 0.5, which is the default, means "look more from the front".



```
\usetikzlibrary {decorations.pathmorphing}
\begin{tikzpicture}[decoration=coil]
  \draw [help lines] grid (3,2);
  \draw [decorate,fill=yellow!80!black]
    (0,0) -- (3,1) arc (0:180:1.5 and 1) -- cycle;
\end{tikzpicture}
```

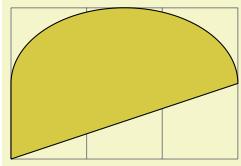


```
\usetikzlibrary {decorations.pathmorphing}
\begin{tikzpicture}
  [decoration={coil,aspect=0.3,segment length=3mm,amplitude=3mm}]
  \draw [help lines] grid (3,2);
  \draw [decorate,fill=yellow!80!black]
    (0,0) -- (3,1) arc (0:180:1.5 and 1) -- cycle;
\end{tikzpicture}
```

Decoration `curveto`

This decoration simply yields a line following the original path. This means that (ideally) it does not change the path and follows any curves in the path (hence the name). In reality, due to the internals of how decorations are implemented, this decoration actually replaces the path by numerous small straight lines.

This decoration is mostly useful in conjunction with meta-decorations. It is also actually defined in the decoration module and is always available.

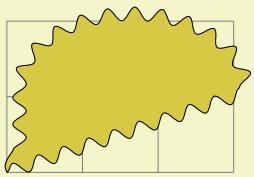


```
\usetikzlibrary {decorations.pathmorphing}
\begin{tikzpicture}[decoration=curveto]
\draw [help lines] grid (3,2);
\draw [decorate,fill=yellow!80!black]
(0,0) -- (3,1) arc (0:180:1.5 and 1) -- cycle;
\end{tikzpicture}
```

Decoration `snake`

This decoration replaces the path by a line that looks like a snake seen from above. More precisely, the snake is a sine wave with a “softened” start and ending. The following parameters influence the snake:

- `amplitude` determines the sine wave’s amplitude.
- `segment length` determines the sine wave’s wavelength.



```
\usetikzlibrary {decorations.pathmorphing}
\begin{tikzpicture}[decoration=snake]
\draw [help lines] grid (3,2);
\draw [decorate,fill=yellow!80!black]
(0,0) -- (3,1) arc (0:180:1.5 and 1) -- cycle;
\end{tikzpicture}
```

51.3 Path Replacing Decorations

TikZ Library `decorations.pathreplacing`

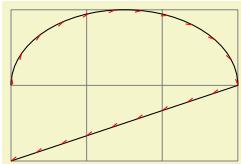
```
\usepgflibrary{decorations.pathreplacing} % LATEX and plain TEX and pure pgf
\usepgflibrary[decorations.pathreplacing] % ConTeXt and pure pgf
\usetikzlibrary{decorations.pathreplacing} % LATEX and plain TEX when using TikZ
\usetikzlibrary[decorations.pathreplacing] % ConTeXt when using TikZ
```

This library defines decorations that replace the to-be-decorated path by another path. Unlike morphing decorations, the replaced path might be quite different, for instance a straight line might be replaced by a set of circles. Note that filling a path that has been replaced using one of the decorations in this library typically does not fill the original area but, rather, the smaller area of the newly-created path segments.

Decoration `border`

This decoration adds straight lines to the path that are at a specific angle to the line toward the target. The idea is to add these little lines to indicate the “border” of an area. The following parameters influence the decoration:

- `segment length` determines the distance between consecutive ticks.
- `amplitude` determines the length of the ticks.
- `angle` determines the angle between the ticks and the line of the path.



```
\usetikzlibrary {decorations.pathreplacing}
\begin{tikzpicture}[postaction={decorate,draw,red}]
\draw [help lines] grid (3,2);
\draw [postaction={decorate,draw,red}]
(0,0) -- (3,1) arc (0:180:1.5 and 1);
\end{tikzpicture}
```

Decoration `brace`

This decoration replaces a straight line path by a long brace. The left and right end of the brace will be exactly on the start and endpoint of the decoration. The decoration really only makes sense for paths that are a straight line.

- `amplitude` determines how much the brace rises above the path.
- `aspect` determines the fraction of the total length where the “middle part” of the brace will be.

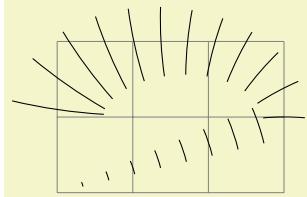


```
\usetikzlibrary {decorations.pathreplacing}
\begin{tikzpicture}[decoration=brace]
  \draw [help lines] grid (3,2);
  \draw [decorate] (0,0) -- (3,1);
\end{tikzpicture}
```

Decoration `expanding waves`

This decoration adds arcs to the path that get bigger along the line towards the target. The following parameters influence the decoration:

- `segment length` determines the distance between consecutive arcs.
- `angle` determines the opening angle below and above the path. Thus, the total opening angle is twice this angle.



```
\usetikzlibrary {decorations.pathreplacing}
\begin{tikzpicture}[decoration={expanding waves,angle=5}]
  \draw [help lines] grid (3,2);
  \draw [decorate] (0,0) -- (3,1) arc (0:180:1.5 and 1);
\end{tikzpicture}
```

Decoration `moveto`

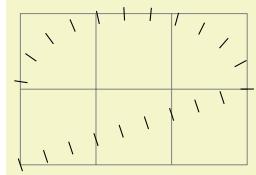
This decoration simply jumps to the end of the path using a move-to path operation. It is mainly useful as `pre=moveto` or `post=moveto` decorations.

This decoration is actually always defined when the decoration module is loaded, but it is documented here for consistency.

Decoration `ticks`

This decoration replaces the path by straight lines that are orthogonal to the path. The following parameters influence the decoration:

- `segment length` determines the distance between consecutive ticks.
- `amplitude` determines half the length of the ticks.

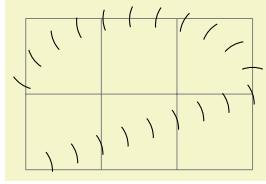


```
\usetikzlibrary {decorations.pathreplacing}
\begin{tikzpicture}[decoration=ticks]
  \draw [help lines] grid (3,2);
  \draw [decorate] (0,0) -- (3,1) arc (0:180:1.5 and 1);
\end{tikzpicture}
```

Decoration `waves`

This decoration replaces the path by arcs that have a constant size. The following parameters influence the decoration:

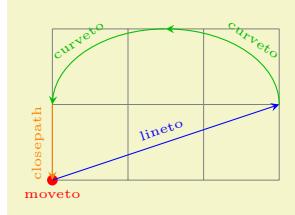
- `segment length` determines the distance between consecutive arcs.
- `angle` determines the opening angle below and above the path. Thus, the total opening angle is twice this angle.
- `radius` determines the radius of each arc.



```
\usetikzlibrary {decorations.pathreplacing}
\begin{tikzpicture}[decoration={waves, radius=4mm}]
  \draw [help lines] grid (3,2);
  \draw [decorate] (0,0) -- (3,1) arc (0:180:1.5 and 1);
\end{tikzpicture}
```

Decoration `show path construction`

This decoration allows “something different” to be done for each *type* of input segment (i.e., moveto, lineto, curveto or closepath). Typically, each segment will be replaced with another path, but this need not necessarily be the case.



```
\usetikzlibrary {decorations.pathreplacing}
\begin{tikzpicture} [>=stealth, every node/.style={midway, sloped, font=\tiny}, decoration={show path construction,
  moveto code={
    \fill [red] (\tikzinputsegmentfirst) circle (2pt)
    node [fill=none, below] {moveto};},
  lineto code={
    \draw [blue,->] (\tikzinputsegmentfirst) -- (\tikzinputsegmentlast)
    node [above] {lineto};},
  curveto code={
    \draw [green!75!black,->] (\tikzinputsegmentfirst) .. controls
      (\tikzinputsegmentsupporta) and (\tikzinputsegmentsupportb)
      ..(\tikzinputsegmentlast) node [above] {curveto};},
  closepath code={
    \draw [orange,->] (\tikzinputsegmentfirst) -- (\tikzinputsegmentlast)
    node [above] {closepath};}}
]
\draw [help lines] grid (3,2);
\path [decorate] (0,0) -- (3,1) arc (0:180:1.5 and 1) -- cycle;
\end{tikzpicture}
```

The following keys can be used to specify the code to execute for each type of input segment.

`/pgf/decoration/moveto code=<code>`

(no default, initially {})

Set the code to be executed for every moveto input segment. It is important to remember that the transformations applied by the decoration automaton are turned *off* when `<code>` is executed.

`/pgf/decoration/lineto code=<code>`

(no default, initially {})

Set the code to be executed for every lineto input segment.

`/pgf/decoration/curveto code=<code>`

(no default, initially {})

Set the code to be executed for every curveto input segment.

`/pgf/decoration/closepath code=<code>`

(no default, initially {})

Set the code to be executed for every closepath input segment.

Within `<code>` the first and last points on the current input segment can be accessed using `\pgfpointdecoratedinputsegmentfirst` and `\pgfpointdecoratedinputsegmentlast`. For curves, the control (support) points can be accessed using `\pgfpointdecoratedinputsegmentsupporta` and `\pgfpointdecoratedinputsegmentsupportb`.

In TikZ, you can use the following macros inside a TikZ coordinate.

`\tikzinputsegmentfirst`

The first point on the current input segment path.

`\tikzinputsegmentlast`

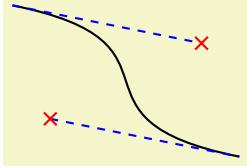
The last point on the current input segment path.

```
\tikzinputsegmentsupporta
```

The first support on the curveto input segment path.

```
\tikzinputsegmentsupportb
```

The second support on the curveto input segment path.



```
\usetikzlibrary {decorations.pathreplacing,shapes.misic}
\tikzset{
    show curve controls/.style={
        decoration={
            show path construction,
            curveto code={
                \draw [blue, dashed]
                    (\tikzinputsegmentfirst) -- (\tikzinputsegmentsupporta)
                    node [at end, cross out, draw, solid, red, inner sep=2pt]{};
                \draw [blue, dashed]
                    (\tikzinputsegmentsupportb) -- (\tikzinputsegmentlast)
                    node [at start, cross out, draw, solid, red, inner sep=2pt]{};
            }
        },decorate
    }
}

\tikzpicture
    \draw [postaction=show curve controls, thick]
        (0,2) .. controls (2.5,1.5) and (0.5,0.5) .. (3,0);
\endtikzpicture
```

51.4 Marking Decorations

51.4.1 Overview

A *marking on a path* is any kind of graphic that is placed on a specific position on a path. Markings are useful in rather diverse situations: you can use them to, say, place little “footsteps” along a path as if someone were walking along the path; to place arrow tips on the middle of a path to indicate the “direction” in which something is flowing; or you can use them to place informative information at certain positions of a path.

For historical reasons there are three different libraries for placing marks on a path. They differ in what kind of markings can be added to a path. We start with the most general and most useful of these libraries.

51.5 Arbitrary Markings

TikZ Library `decorations.markings`

```
\usepgflibrary{decorations.markings} % LATEX and plain TEX and pure pgf
\usepgflibrary[decorations.markings] % ConTeXt and pure pgf
\usetikzlibrary{decorations.markings} % LATEX and plain TEX when using TikZ
\usetikzlibrary[decorations.markings] % ConTeXt when using TikZ
```

Markings are arbitrary “marks” that can be put on a path. Marks can be arrow tips or nodes or even whole pictures.

Decoration `markings`

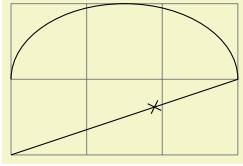
A *marking* can be thought of a “little picture” or more precisely of “some scope contents” that is placed “on” a path at a certain position. Suppose the marking should be a simple cross. We can produce this with the following code:

```
\draw (-2pt,-2pt) -- (2pt,2pt);
\draw (2pt,-2pt) -- (-2pt,2pt);
```

If we use this code as a marking at position `2cm` on a path, then the following happens: PGF determines the position on the path that is `2cm` along the path. Then it translates the coordinate system to this position and rotates it such that the positive x -axis is tangent to the path. Then a protective scope is created, inside which the above code is executed – resulting in a little cross on the path.

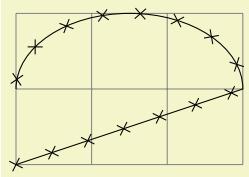
The `markings` decoration allows you to place one or more such markings on a path. The decoration destroys the input path (except in certain cases, detailed later), which means that it uses the path for determining positions on the path, but after the decoration is done this path is gone. You typically need to use a `postaction` to add markings.

Let us start with the above example in real code:



```
\usetikzlibrary {decorations.markings}
\begin{tikzpicture}[decoration={%
    markings,% switch on markings
    mark=% actually add a mark
    at position 2cm
    with
    {
        \draw (-2pt,-2pt) -- (2pt,2pt);
        \draw (2pt,-2pt) -- (-2pt,2pt);
    }
}]
\draw [help lines] grid (3,2);
\draw [postaction={decorate}] (0,0) -- (3,1) arc (0:180:1.5 and 1);
\end{tikzpicture}
```

We can also add the cross repeatedly:



```
\usetikzlibrary {decorations.markings}
\begin{tikzpicture}[decoration={%
    markings,% switch on markings
    mark=% actually add a mark
    between positions 0 and 1 step 5mm
    with
    {
        \draw (-2pt,-2pt) -- (2pt,2pt);
        \draw (2pt,-2pt) -- (-2pt,2pt);
    }
}]
\draw [help lines] grid (3,2);
\draw [postaction={decorate}] (0,0) -- (3,1) arc (0:180:1.5 and 1);
\end{tikzpicture}
```

The `mark` decoration option is used to specify a marking. It comes in two versions:

`/pgf/decoration/mark=at position <pos> with <code>` (no default)

The options specifies that when a `marking` decoration is applied, there should be a marking at position `<pos>` on the path whose code is given by `<code>`.

The `<pos>` can have four different forms:

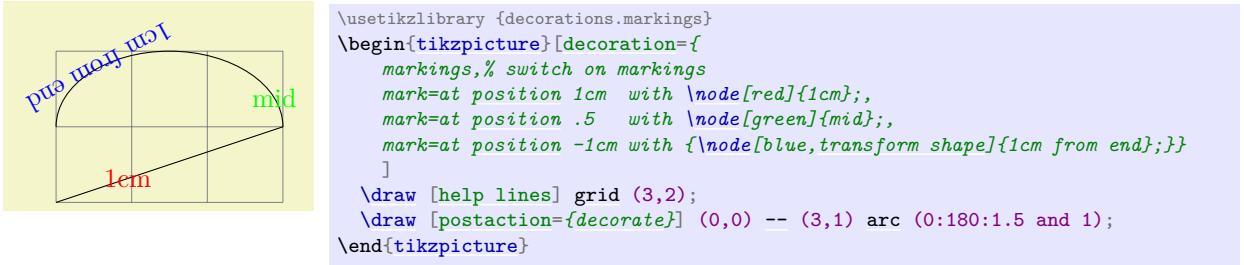
1. It can be a non-negative dimension like `0pt` or `2cm` or `5cm/2`. In this case, it refers to the position along the path that is this far displaced from the start.
2. It can be a negative dimension like `-1cm-2pt` or `-1sp`. In this case, the position is taken from the end of the path. Thus, `-1cm` is the position that is -1cm displaced from the end of the path.
3. It can be a dimensionless non-negative number like `1/2` or `0.333+2*0.1`. In this case, the `<pos>` is interpreted as a factor of the total path length. Thus, a `<pos>` or `0.5` refers to the middle of the path, `0.1` is near the start, and so on.
4. It can be a dimensionless negative number like `-0.1`. Then, again, the fraction of the path length counts “from the end”.

The `<pos>` determines a position on the path. When the marking is applied, the (high level) coordinate system will have been transformed so that the origin lies at this position and the positive x -axis points along the path. For this coordinate system, the `<code>` is executed. It can contain all sorts of graphic drawing commands, including (even named) nodes.

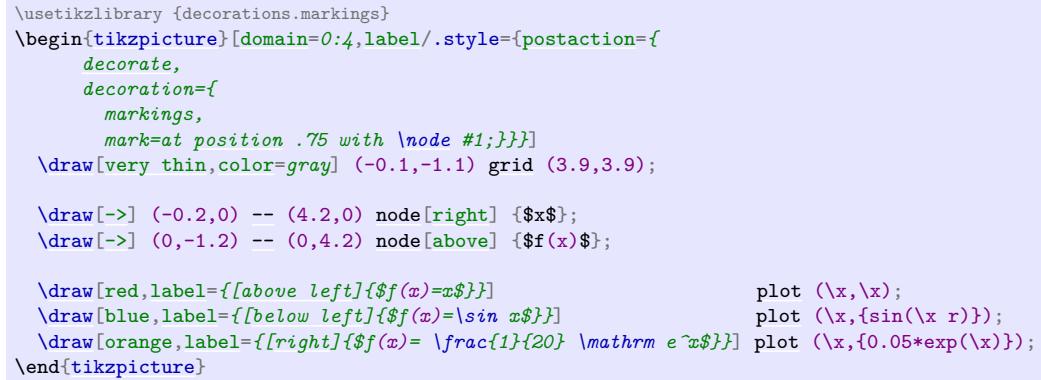
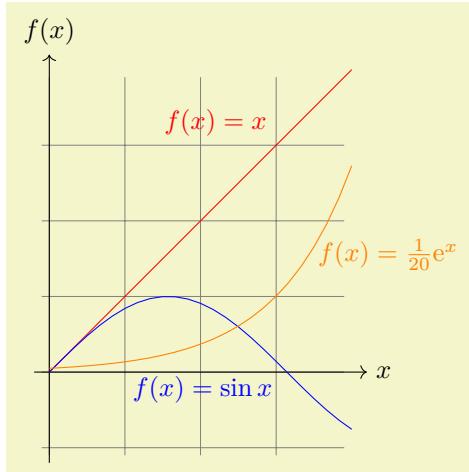
If the position lies past the end of the path (for instance if `<pos>` is set to `1.2`), the marking will not be drawn.

It is possible to give the `mark` option several times, which causes several markings to be applied. In this case, however, it is necessary that the positions on the path are in increasing order. That is,

it is not allowed (and will result in chaos) to have a marking that lies earlier on the path to follow a marking that is later on the path.



Here is an example that shows how markings can be used to place text on plots:



When the `<code>` is being executed, two special keys will have been set up, whose value may be of interest:

`/pgf/decoration/mark info/sequence number` (no value)

This key can only be read. Its value (which can be obtained using the `\pgfkeysvalueof` command) is a “sequence number” of the mark. The first mark that is added to a path has number 1, the second number 2, and so on. This key is mainly useful in conjunction with repeated markings (see below).

`/pgf/decoration/mark info/distance from start` (no value)

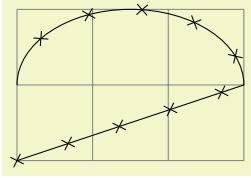
This key can only be read. Its value is the distance of the marking from the start of the path in points. For instance, if the path length is 100pt and the marking is in the middle of the path, the value of this key would be 50.0pt.

A second way to use the `mark` key is the following:

`/pgf/decoration/mark=between positions <start pos> and <end pos> step <stepping> with <code>` (no default)

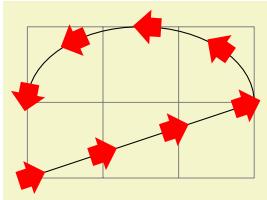
This works similarly to the `at` position version of this option, only multiple marks are placed, starting at `<start pos>` and then spaced apart by `<stepping>`. The `<start pos>`, the `<end pos>`, and also the `<stepping>` may all be specified in the same way as for the `at` position version, that is, either using units or no units and also using positive or negative values.

Let us start with a simple example in which we place ten crosses along a path starting with the beginning of the path (`<start pos> = 0`) and ending at the end (`<end pos> = 1`).



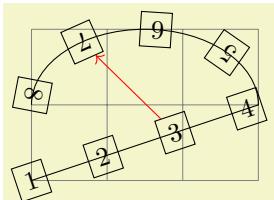
```
\usetikzlibrary {decorations.markings}
\begin{tikzpicture}[decoration={markings,
    mark=between positions 0 and 1 step 0.1
        with { \draw (-2pt,-2pt) -- (2pt,2pt);
            \draw (2pt,-2pt) -- (-2pt,2pt); } }]
\draw [help lines] grid (3,2);
\draw [postaction={decorate}] (0,0) -- (3,1) arc (0:180:1.5 and 1);
\end{tikzpicture}
```

In the next example we place arrow shapes on the path instead of crosses. Note the use of the `transform shape` option to ensure that the nodes are actually rotated.



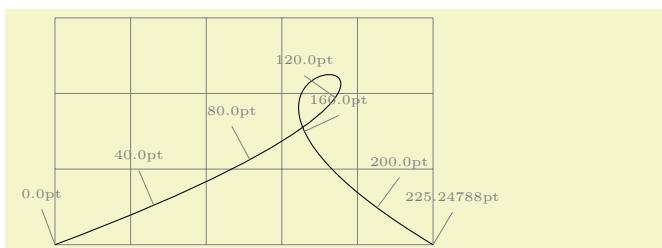
```
\usetikzlibrary {decorations.markings,shapes.arrows}
\begin{tikzpicture}[decoration={markings,
    mark=between positions 0 and 1 step 1cm
        with { \node [single arrow,fill=red,
            single arrow head extend=3pt,transform shape] {};} }]
\draw [help lines] grid (3,2);
\draw [postaction={decorate}] (0,0) -- (3,1) arc (0:180:1.5 and 1);
\end{tikzpicture}
```

Using the key `sequence number` we can also “number” the nodes and even refer to them later on.



```
\usetikzlibrary {decorations.markings}
\begin{tikzpicture}[decoration={markings,
    mark=between positions 0 and 1 step 1cm with {
        \node [draw,
            name=mark-\pgfkeysvalueof{/pgf/decoration/mark info/sequence number},
            transform shape]
            {\pgfkeysvalueof{/pgf/decoration/mark info/sequence number}}; }}]
\draw [help lines] grid (3,2);
\draw [postaction={decorate}] (0,0) -- (3,1) arc (0:180:1.5 and 1);
\draw [red,->] (mark-3) -- (mark-7);
\end{tikzpicture}
```

In the following example we use the distance info to place “length information” on a path:



```
\usetikzlibrary {decorations.markings}
\begin{tikzpicture}[decoration={markings,
    % Main marks
    mark=between positions 0 and 1 step 40pt with
        { \draw [help lines] (0,0) -- (0,0.5)
            node[above,font=\tiny]{\pgfkeysvalueof{/pgf/decoration/mark info/distance from start}}; },
    mark=at position -0.1pt with
        { \draw [help lines] (0,0) -- (0,0.5)
            node[above,font=\tiny]{\pgfkeysvalueof{/pgf/decoration/mark info/distance from start}}; }]
\draw [help lines] grid (5,3);
\draw [postaction={decorate}] (0,0) .. controls (8,3) and (0,3) .. (5,0) ;
\end{tikzpicture}
```

```
/pgf/decoration/reset marks
```

(no value)

Since `mark` options accumulate, there needs to be a way to “reset” things, so that any `mark` options set in an enclosing scope do not interfere. This option does exactly this. Note that when the `<code>` of a marking is executed, the markings are automatically reset.

As mentioned earlier, the decoration usually destroys the path. However, this is no longer the case when the following key is set:

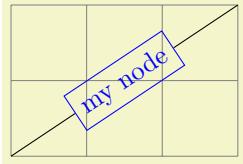
```
/pgf/decoration/mark connection node=<node name>
```

(no default, initially `empty`)

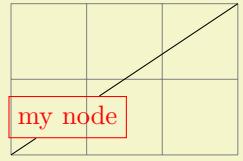
When this key is set to a non-empty `<node name>` while the decoration is being processed, the following happens: The marking code should, among possibly other things, define a node named `<node name>`. Then, the output path of this decoration will contain a line-to to “one end” of this node, followed by a moveto to the “other end” of the node. More precisely, the first end is given by the position on the border of `<node name>` that lies in the direction “from which the path heads toward the node” while the other end lies on the border “where the path heads away from the node”. Furthermore, this option causes the decoration to end with a line-to to the end instead of a move-to.

The net effect of all this is that when you decorate a straight line with one or more markings that contain just a node, the line will effectively connect these nodes.

Here are two examples that show how this works:



```
\usetikzlibrary {decorations.markings}
\begin{tikzpicture}[decoration={markings,
    mark connection node=my node,
    mark=at position .5 with
        {\node [draw,blue,transform shape] (my node) {my node};}]
    \draw [help lines] grid (3,2);
    \draw decorate { (0,0) -- (3,2) };
\end{tikzpicture}
```



```
\usetikzlibrary {decorations.markings}
\begin{tikzpicture}[decoration={markings,
    mark connection node=my node,
    mark=at position .25 with
        {\node [draw,red] (my node) {my node};}]
    \draw [help lines] grid (3,2);
    \draw decorate { (0,0) -- (3,2) };
\end{tikzpicture}
```

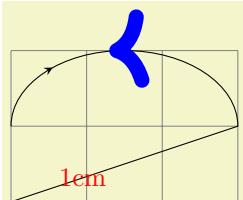
51.5.1 Arrow Tip Markings

Frequent markings that are hard to create correctly are arrow tips. For them, two special commands are available when the `<code>` of a `mark` option is executed. (They are only defined in this code):

```
\arrow[<options>]{<arrow end tip>}
```

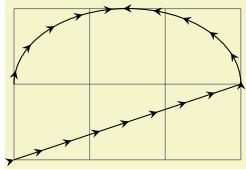
This command simply draws the `<arrow end tip>` at the origin, pointing right. This is exactly what you need when you want to draw an arrow tip as a marking.

The `<options>` can only be given when TikZ is used. In this case, they are executed in a scope that contains the arrow tip.



```
\usetikzlibrary {decorations.markings}
\begin{tikzpicture}[decoration={
    markings,% switch on markings
    mark=at position 1cm with {\node[red]{1cm};},
    mark=at position .75 with {\arrow[blue,line width=2mm]{>}},
    mark=at position -1cm with {\arrowreversed[black]{stealth}}}
]
\draw [help lines] grid (3,2);
\draw [postaction={decorate}] (0,0) -- (3,1) arc (0:180:1.5 and 1);
\end{tikzpicture}
```

Here is a more useful example:



```
\usetikzlibrary {decorations.markings}
\begin{tikzpicture}[decoration={%
  markings,% switch on markings
  mark=at position 0 and .75 step 4mm with {\arrow{stealth}},
  mark=at position .75 and 1 step 4mm with {\arrowreversed{stealth}}%
}]
\draw [help lines] grid (3,2);
\draw [postaction={decorate}] (0,0) -- (3,1) arc (0:180:1.5 and 1);
\end{tikzpicture}
```

\arrowreversed[*<options>*]{*<arrow end tip>*}

As above, only the arrow end tip is flipped and points in the other direction.

51.5.2 Footprint Markings

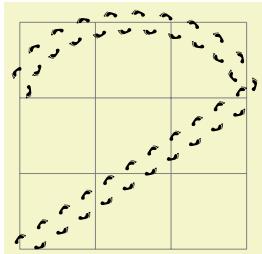
TikZ Library `decorations.footprints`

```
\usepgflibrary{decorations.footprints} % LATEX and plain TEX and pure pgf
\usepgflibrary[decorations.footprints] % ConTeXt and pure pgf
\usetikzlibrary{decorations.footprints} % LATEX and plain TEX when using TikZ
\usetikzlibrary[decorations.footprints] % ConTeXt when using TikZ
```

The decorations of this library can be used to decorate a path with little footprints, as if someone had “walked” along the path.

Decoration `footprints`

The footprint decoration adds little footprints around the path. They start with the left foot.



```
\usetikzlibrary {decorations.footprints}
\begin{tikzpicture}[decoration={footprints,foot length=5pt,stride length=10pt}]
\draw [help lines] grid (3,3);
\fill [decorate] (0,0) -- (3,2) arc (0:180:1.5 and 1);
\end{tikzpicture}
```

You can influence the way this decoration looks using the following options:

/pgf/decoration/foot length

(initially 10pt)

The length or size of the footprint itself. A larger value makes the footprint larger, but does not change the stride length.



```
\usetikzlibrary {decorations.footprints}
\begin{tikzpicture}[decoration={footprints,foot length=20pt}]
\fill [decorate] (0,0) -- (3,0);
\end{tikzpicture}
```

/pgf/decoration/stride length

(initially 30pt)

The length of strides. This is the distance between the beginnings of left footprints along the path.



```
\usetikzlibrary {decorations.footprints}
\begin{tikzpicture}[decoration={footprints,stride length=50pt}]
\fill [decorate] (0,0) -- (3,0);
\end{tikzpicture}
```

/pgf/decoration/foot sep

(initially 4pt)

The separation in the middle between the footprints. The footprints are moved away from the path by half this amount.



```
\usetikzlibrary {decorations.footprints}
\begin{tikzpicture}[decoration={footprints,foot sep=10pt}]
  \fill [decorate] (0,0) -- (3,0);
\end{tikzpicture}
```

/pgf/decoration/foot angle

(initially 10)

Footprints are rotated by this much.



```
\usetikzlibrary {decorations.footprints}
\begin{tikzpicture}[decoration={footprints,foot angle=60}]
  \fill [decorate] (0,0) -- (3,0);
\end{tikzpicture}
```

/pgf/decoration/foot of

(initially `human`)

The species whose footprints are shown. Possible values are:

Species

Result

gnome



human



bird



felis silvestris



51.5.3 Shape Background Markings

The third library for adding markings uses the background paths of certain shapes. This library is included mostly for historical reasons, using the `markings` library is usually preferable.

TikZ Library `decorations.shapes`

```
\usepgflibrary{decorations.shapes} % LETX and plain TEX and pure pgf
\usepgflibrary[decorations.shapes] % ConTEXt and pure pgf
\usetikzlibrary{decorations.shapes} % LETX and plain TEX when using TikZ
\usetikzlibrary[decorations.shapes] % ConTEXt when using TikZ
```

This library defines decorations that use shapes or shape-like drawings to decorate a path. The following options are common options used by the decorations in this library:

/pgf/decoration/shape width=(dimension)

(no default, initially 2.5pt)

The desired width of the shapes. For decorations that support varying shape sizes, this key sets both the start and end width (which can be overwritten using options like `shape start width`).

/pgf/decoration/shape height=(dimension)

(no default, initially 2.5pt)

Works like the previous key, only for the height.

/pgf/decoration/shape size=(dimension)

(no default)

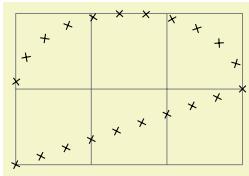
Sets the desired width and height simultaneously.

For the exact places and macros where these keys store the values, please consult the beginning of the code of the library.

Decoration `crosses`

This decoration replaces the path by (diagonal) crosses. The following parameters influence the decoration:

- `segment length` determines the distance between (the centers of) consecutive crosses.
- `shape height` determines the height of each cross.
- `shape width` determines the width of each cross.

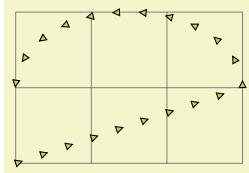


```
\usetikzlibrary {decorations.shapes}
\begin{tikzpicture}[decoration=crosses]
\draw [help lines] grid (3,2);
\draw [decorate] (0,0) -- (3,1) arc (0:180:1.5 and 1);
\end{tikzpicture}
```

Decoration `triangles`

This decoration replaces the path by triangles that point along the path. The following parameters influence the decoration:

- `segment length` determines the distance between consecutive triangles.
- `shape height` determines the height of the triangle side that is orthogonal to the path.
- `shape width` determines the width of the triangle.

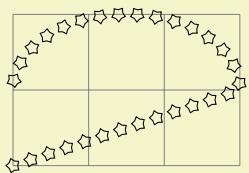


```
\usetikzlibrary {decorations.shapes}
\begin{tikzpicture}[decoration=triangles]
\draw [help lines] grid (3,2);
\draw [decorate,fill=yellow!80!black] (0,0) -- (3,1) arc (0:180:1.5 and 1);
\end{tikzpicture}
```

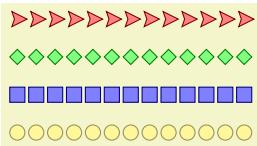
Decoration `shape backgrounds`

This is a general decoration that replaces the to-be-decorated path by repeated copies of the background path of an arbitrary shape that has previously been defined using the `\pgfdeclareshape` command (that is, you can use any shape in the shape libraries).

Please note that the background path of the shapes is used, but *no nodes are created*. This means that *you cannot have text inside the shapes of this path, you cannot name them, or refer to them*. Finally, this decoration *will not work with shapes that depend strongly on the size of the text box (like the arrow shapes)*. If any of these restrictions pose a problem, use the `markings` library instead.



```
\usetikzlibrary {decorations.shapes,shapes.geometric}
\begin{tikzpicture}[decoration={shape backgrounds,shape=star,shape size=5pt}]
\draw [help lines] grid (3,2);
\draw [decorate] (0,0) -- (3,1) arc (0:180:1.5 and 1);
\end{tikzpicture}
```



```
\usetikzlibrary {decorations.shapes,shapes.geometric}
\tikzset{paint/.style={ draw=#1!50!black, fill=#1!50 },
          decorate with/.style=
            {decorate,decoration={shape backgrounds,shape=#1,shape size=2mm}}}
\begin{tikzpicture}
\draw [decorate with=dart,    paint=red] (0,1.5) -- (3,1.5);
\draw [decorate with=diamond, paint=green] (0,1)   -- (3,1);
\draw [decorate with=rectangle, paint=blue] (0,0.5) -- (3,0.5);
\draw [decorate with=circle,   paint=yellow] (0,0)   -- (3,0);
\end{tikzpicture}
```

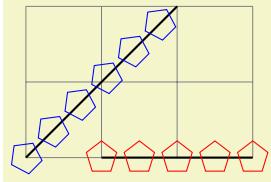
All shapes are positioned by the anchor that is specified via the `anchor` decoration option:

`/pgf/decoration/anchor=(anchor)`

(no default, initially `center`)

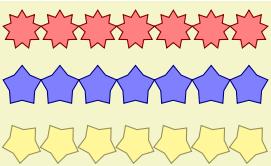
The anchor used to position the shape backgrounds.

A shape background path is added at the start point of the path and, if the distance between the shapes is appropriate, at the end point of the path.



```
\usetikzlibrary {decorations.shapes,shapes.geometric}
\begin{tikzpicture}[decoration=
    shape backgrounds,shape=regular polygon,shape size=.5mm]
\draw [help lines] grid (3,2);
\draw [thick] (0,0) -- (2,2) (1,0) -- (3,0);
\draw [red, decorate, decoration={shape sep=.5cm}] (1,0) -- (3,0);
\draw [blue, decorate, decoration={shape sep=.5cm}] (0,0) -- (2,2);
\end{tikzpicture}
```

Keys for customizing specific shapes can be specified (e.g., `star points`, `cloud puffs`, `kite angles`, and so on). The size of the shape is “enforced” using transformations. This means that the shape is typeset with an empty text box and some default size values, resulting in an initial shape. This shape is then rescaled using coordinate transformations so that it has the desired size (which may vary as we travel along the to-be-decorated path). This means that settings involving angles and distances may not appear entirely accurate. More general options such as `inner sep` and `minimum size` will be ignored, but transformations can be applied to each segment as described below.



```
\usetikzlibrary {decorations.shapes,shapes.geometric}
\tikzset{
    paint/.style={draw=#1!50!black, fill=#1!50},
    my star/.style={decorate,decoration={shape backgrounds,shape=star},
        star points=#1}
}
\begin{tikzpicture}[decoration={shape sep=.5cm, shape size=.5cm}]
\draw [my star=9, paint=red] (0,.15) -- (3,.15);
\draw [my star=5, paint=blue] (0,.75) -- (3,.75);
\draw [my star=5, paint=yellow, shape border rotate=30] (0,0) -- (3,0);
\end{tikzpicture}
```

There are various keys to control the drawing of the shape decoration.

`/pgf/decoration/shape=(shape name)`

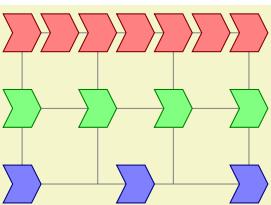
(no default, initially `circle`)

The shape whose background path is used.

`/pgf/decoration/shape sep=(spacing)`

(no default, initially `.25cm`, between centers)

Set the spacing between the shapes on the decorations path. This can be just a distance on its own, but the additional keywords `between centers`, and `between borders` (which must be preceded by a comma), specify that the distance is between the center anchors of the shapes or between the edges of the *boundaries* of the shape borders.



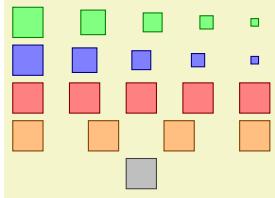
```
\usetikzlibrary {decorations.shapes,shapes.symbols}
\begin{tikzpicture}[
    decoration={shape backgrounds,shape size=.5cm,shape=signal},
    signal from=west, signal to=east,
    paint/.style={decorate, draw=#1!50!black, fill=#1!50}
]
\draw [help lines] grid (3,2);
\draw [paint=red, decoration={shape sep=.5cm}]
(0,2) -- (3,2);
\draw [paint=green, decoration={shape sep={1cm, between centers}}]
(0,1) -- (3,1);
\draw [paint=blue, decoration={shape sep={1cm, between borders}}]
(0,0) -- (3,0);
\end{tikzpicture}
```

`/pgf/decoration/shape evenly spread=(number)`

(no default)

This key overrides the `shape sep` key and forces the decoration to fit `(number)` shapes evenly across the path. If `(number)` is less than 1, then no shapes will be used. If `(number)` equals 1, then one shape is put in the middle of the path. The additional keywords `by centers` (the default, if no keyword is specified) and `by borders` can be used (both preceded by a comma), to specify how

the distance between shapes is determined. These keywords will only have a noticeable effect if the shapes sizes differ over time.



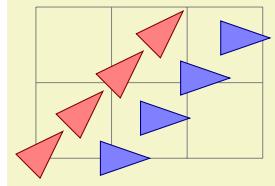
```
\usetikzlibrary {decorations.shapes}
\begin{tikzset{
  paint/.style={draw=#1!50!black, fill=#1!50},
  spreading/.style={
    decorate,decoration={shape backgrounds, shape=rectangle,
    shape start size=4mm,shape end size=1mm,shape evenly spread={#1}}}
}

\begin{tikzpicture}
  \fill [paint=green,spreading={5, by borders},
    decoration={shape scaled}] (0,2) -- (3,2);
  \fill [paint=blue,spreading={5, by centers},
    decoration={shape scaled}] (0,1.5) -- (3,1.5);
  \fill [paint=red, spreading=5] (0,1) -- (3,1);
  \fill [paint=orange, spreading=4] (0,.5) -- (3,.5);
  \fill [paint=gray, spreading=1] (0,0) -- (3,0);
\end{tikzpicture}
```

`/pgf/decoration/shape sloped=(boolean)`

(no default, initially `true`)

By default, shapes are rotated to the slope of the decorations path. If `<boolean>` is the value `false`, then this rotation is turned off. Internally this sets the T_EX-if `\ifpgfshapedecorationsloped` accordingly.



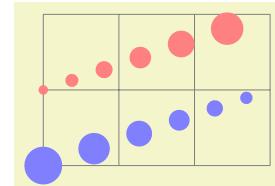
```
\usetikzlibrary {decorations.shapes,shapes.geometric}
\begin{tikzset{
  paint/.style={draw=#1!50!black, fill=#1!50}
}

\begin{tikzpicture}[decoration={
  shape width=.65cm, shape height=.45cm,
  shape=isosceles triangle, shape sep=.75cm,
  shape backgrounds}]
  \draw [help lines] grid (3,2);
  \draw [paint=red,decorate] (0,0) -- (2,2);
  \draw [paint=blue,decorate,decoration={shape sloped=false}]
  (1,0) -- (3,2);
\end{tikzpicture}
```

It is possible to scale the width and height of the shapes along the length of the decorations path. The shapes are scaled between the starting size and the ending size. The following keys customize the way the decoration shapes are scaled:

`/pgf/decoration/shape scaled=(boolean)`

(no default, initially `false`)



```
\usetikzlibrary {decorations.shapes}
\begin{tikzset{
  bigger/.style={decoration={shape start size=.125cm, shape end size=.5cm}},
  smaller/.style={decoration={shape start size=.5cm, shape end size=.125cm},
  decoration={shape backgrounds,
  shape sep=.25cm, between borders},shape scaled}
}

\begin{tikzpicture}
  \draw [help lines] grid (3,2);
  \fill [decorate, bigger, red!50] (0,1) -- (3,2);
  \fill [decorate, smaller, blue!50] (0,0) -- (3,1);
\end{tikzpicture}
```

If this key is set to false (which is the default), then only the start width and height are used. Note that the keys `shape width` and `shape height` set the start and end height simultaneously.

`/pgf/decoration/shape start width=(length)`

(no default, initially 2.5pt)

The starting width of the shape.

`/pgf/decoration/shape start height=(length)`

(no default, initially 2.5pt)

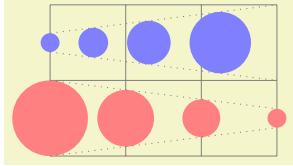
The starting height of the shape.

`/pgf/decoration/shape start size=<length>` (style, no default)

Sets both the start height and start width simultaneously.

`/pgf/decoration/shape end width=<length>` (no default, initially 2.5pt)

The recommended ending width of the shape. Note that this is the width that a shape will take only if it is drawn exactly at the end of the path.



```
\usetikzlibrary {decorations.shapes}
\tikzset{
    bigger/.style={decoration={shape start size=.25cm, shape end size=1cm}},
    smaller/.style={decoration={shape start size=1cm, shape end size=.25cm}},
    decoration={shape backgrounds,
        shape sep={.25cm, between borders},shape scaled}
}
\begin{tikzpicture}
    \draw [help lines] grid (3,2);
    \fill [decorate,bigger,
        decoration={shape sep={.25cm, between borders}}, blue!50]
    (0,1.5) -- (3,1.5);
    \fill [decorate,smaller,
        decoration={shape sep={1cm, between centers}}, red!50]
    (0,.5) -- (3,.5);
    \draw [gray, dotted] (0,1.625) -- (3,2) (0,1.375) -- (3,1)
    (0,1) -- (3,.625) (0,0) -- (3,.375);
\end{tikzpicture}
```

`/pgf/decoration/shape end height=<length>` (no default)

The recommended ending height of the shape.

`/pgf/decoration/shape end size=<length>` (style, no default)

Set both the end height and end width simultaneously.

51.6 Text Decorations

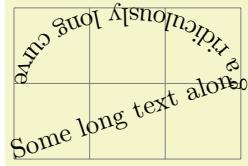
TikZ Library `decorations.text`

```
\usepgflibrary{decorations.text} % LATEX and plain TEX and pure pgf
\usepgflibrary[decorations.text] % ConTeXt and pure pgf
\usetikzlibrary{decorations.text} % LATEX and plain TEX when using TikZ
\usetikzlibrary[decorations.text] % ConTeXt when using TikZ
```

The decoration in this library decorates the path with some text. This can be used to draw text that follows a curve.

Decoration `text along path`

This decoration decorates the path with text. This drawing of the text is a “side effect” of the decoration. The to-be-decorated path is only used to determine where the characters should be put and it is thrown away after the decoration is done. This is why no line is shown in the following example.



```
\usetikzlibrary {decorations.text}
\catcode`\|12
\begin{tikzpicture}[decoration={text along path,
    text={Some long text along a ridiculously long curve that}}]
    \draw [help lines] grid (3,2);
    \draw [decorate] (0,0) -- (3,1) arc (0:180:1.5 and 1);
\end{tikzpicture}
```

PGF “does its best” to typeset the text, however you should note the following points:

- Each character in the text is typeset in a separate `\hbox`. This means that if you want fancy things like kerning or ligatures you will have to manually annotate the characters in the decoration text within a group, for example, `W{\kern-1ptA}TER`.
- Each character is positioned using the center of its baseline. To move the text vertically (relative to the path), the additional transform key should be used.

- No attempt is made to ensure characters do not overlap when the angle between segments is considerably less than 180° (this is tricky to do in TeX without a huge processing overhead). In general this should not be too much of a problem, but, once again, kerning can be used in most cases to overcome any undesirable effects.
- It is only possible to typeset text in math mode under considerable restrictions. Math mode is entered and exited using any character of category code 3 (e.g., in plain TeX this is \$). Math subscripts and superscripts need to be contained within braces (e.g., \hat{y}_i) as do commands like \times or \cdot . However, even modestly complex mathematical typesetting is unlikely to be successful along a path (or even desirable).
- Some inaccuracies in positioning may be particularly apparent at input segment boundaries. This can (unfortunately) only be solved on a case-by-case basis by individually kerning the offending characters within a group.

The following keys are used by the `text` decoration:

`/pgf/decoration/text=(text)` (no default, initially empty)

Sets the text to typeset along the curve. Consecutive spaces are ignored, so \ (or \space in L^AT_EX) should be used to insert multiple spaces. It is possible to format the text using normal formatting commands, such as `\it`, `\bf` and `\color`, within customizable delimiters. Initially these delimiters are both | (however, care will be needed regarding the category codes of delimiters – see below).



```
\usetikzlibrary {decorations.text}
\catcode`\|12
\begin{tikzpicture}
  \draw [help lines] grid (3,2);
  \path [decorate,decoration={text along path,
    text={a big |\color{green}|green|| juicy apple.}}]
  (0,0) .. controls (0,2) and (3,0) .. (3,2);
\end{tikzpicture}
```

By following the first delimiter with +, the formatting commands are added to any existing formatting.



```
\usetikzlibrary {decorations.text}
\begin{tikzpicture}
  \draw [help lines] grid (3,2);
  \path [decorate,decoration={text along path,
    text={a |\large/big|+|\bf|color{red}|red|| juicy apple.}}]
  (0,0) .. controls (0,2) and (3,0) .. (3,2);
\end{tikzpicture}
```

Internally, the text is stored in the macro `\pgfdecorationtext`. Any characters that have not been typeset when the end of the path has been reached will be stored in `\pgfdecorationrestoftext`.

`/pgf/decoration/text format delimiters={⟨before⟩}{⟨after⟩}` (no default, initially {}{})

Set the characters that the text decoration will use to parse formatting commands. If `⟨after⟩` is empty, then `⟨before⟩` will be used for both delimiters. In general you should stick to characters whose category codes are 11 or 12. As + is used to indicate that the specified format commands are added to any existing ones, you should avoid using + as a delimiter.



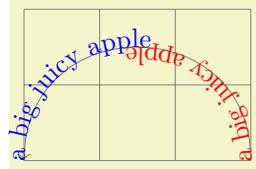
```
\usetikzlibrary {decorations.text}
\begin{tikzpicture}
  \draw [help lines] grid (3,2);
  \path [decorate, decoration={text along path, text format delimiters={[]{}},
    text={A big [\color{red}]red[] and [\color{green}]green[] apple.}}]
  (0,0) .. controls (0,2) and (3,0) .. (3,2);
\end{tikzpicture}
```

`/pgf/decoration/text color=⟨color⟩` (no default, initially black)

The color of the text.

`/pgf/decoration/reverse path=⟨boolean⟩` (no default, initially false)

This key reverses the path. This is especially useful for typesetting text along different sides of curves.



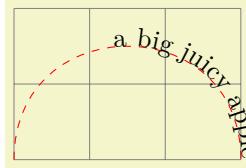
```
\usetikzlibrary {decorations.text}
\begin{tikzpicture}
\draw [help lines] grid (3,2);
\draw [gray, ->]
[postaction={decoration={text along path,
    text={a big juicy apple}, text color=red}, decorate}]
[postaction={decoration={text along path,
    text={a big juicy apple}, text color=blue, reverse path}, decorate}]
(3,0) .. controls (3,2) and (0,2) .. (0,0);
\end{tikzpicture}
```

`/pgf/decoration/text align={⟨alignment options⟩}` (no default)

This changes the key path to `/pgf/decoration/text align` and executes `⟨alignment options⟩`.

`/pgf/decoration/text align/align=⟨alignment⟩` (no default, initially `left`)

Aligns the text according to `⟨alignment⟩`, which should be one of `left`, `right`, or `center`.



```
\usetikzlibrary {decorations.text}
\begin{tikzpicture}
\draw [help lines] grid (3,2);
\draw [red, dashed]
[postaction={decoration={text along path, text={a big juicy apple},
    text align=⟨align=right⟩}, decorate}]
(0,0) .. controls (0,2) and (3,2) .. (3,0);
\end{tikzpicture}
```

`/pgf/decoration/text align/left` (style, no value)

Aligns the text to the left end of the path.

`/pgf/decoration/text align/right` (style, no value)

Aligns the text to the right end of the path.

`/pgf/decoration/text align/center` (style, no value)

Aligns the text to the center of the path.

`/pgf/decoration/text align/left indent=⟨length⟩` (no default, initially `0pt`)

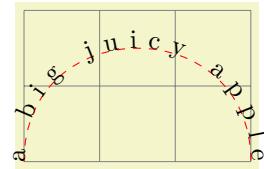
Specifies a distance which the automaton should move along before it starts typesetting the text.

`/pgf/decoration/text align/right indent=⟨length⟩` (no default, initially `0pt`)

Specifies a distance before the end of the path, where the automaton should stop typesetting the text.

`/pgf/decoration/text align/fit to path=⟨boolean⟩` (no default, initially `false`)

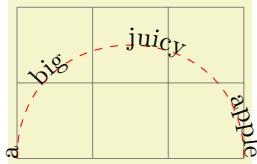
This key makes the decoration automaton try to fit the text to the length of the path. The automaton shifts forward by a small amount between each character in order to fit the text to the path. If, however, the length of the text is longer than the length of the path (i.e., the automaton would have to shift *backwards* between characters) this key will have no effect.



```
\usetikzlibrary {decorations.text}
\begin{tikzpicture}
\draw [help lines] grid (3,2);
\draw [red, dashed]
[postaction={decoration={text along path, text={a big juicy apple},
    text align=fit to path}, decorate}]
(0,0) .. controls (0,2) and (3,2) .. (3,0);
\end{tikzpicture}
```

`/pgf/decoration/text align/fit to path stretching spaces=⟨boolean⟩` (no default, initially `false`)

This key works like the previous key except the automaton shifts forward only for space characters (including `\space`, but *excluding* `\`).



```
\usetikzlibrary {decorations.text}
\begin{tikzpicture}
\draw [help lines] grid (3,2);
\draw [red, dashed]
[postaction={decoration={text along path, text={a big juicy apple}, text align={fit to path stretching spaces}}, decorate}]
(0,0) .. controls (0,2) and (3,2) .. (3,0);
\end{tikzpicture}
```

Decoration `text effects along path`

This decoration is similar to the `text along path` decoration except that each character is inserted into the picture as a TikZ node, and node options (such as `text`, `scale` and `opacity`) can be used to create ‘text effects’.



```
\usetikzlibrary {decorations.text,math}
\bfseries\large
\begin{tikzpicture}[decoration={text effects along path,
text={text effects along path!}, text align=center,
text effects/.cd,
character count=\i, character total=\n,
characters={evaluate={\i=\i/\n*100;}, text along path, text=red!\c!orange},
character widths={text along path, xscale=0, yscale=1}}]

\path [postaction={decorate}, preaction={decorate,
text effects={characters/.append={yscale=-1.5, opacity=0.5,
text=gray, xscale=(\i/\n-0.5)*3}}}]
(0,0) .. controls ++(2,1) and ++(-2,-1) .. (3,4);
\end{tikzpicture}
```

There are some important differences between this decoration and the `text along path` decoration:

- formatting (e.g., font and color) cannot be specified in the decoration text. They can only be specified using the keys described below.
- as a consequence of using the TikZ node options, this decoration is only available in TikZ.
- due to the number of computations involved, this is quite a slow decoration.

The following keys are shared with the `text along path` decoration:

`/pgf/decoration/text={<text>}` (no default)

Set the text this decoration will use. Braces can be used to group multiple characters together, or commands that should not be expanded until they are typeset, for example `gr{\\"o}{\ss}eren`. You should *not* use the formatting delimiters or math mode characters that the `text along path` decoration supports.

`/pgf/decoration/text align=<align>` (no default)

This sets the alignment of the text along the path. The `<align>` argument should be `left`, `right` or `center`. Spreading the text out, or stretching the spaces between words is *not* supported.

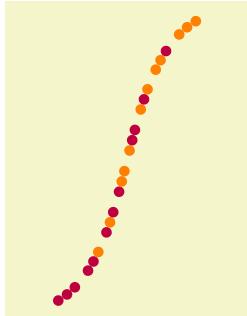
The decoration text can be thought of as consisting of *characters* arranged in to sequences of *letters* to make *words* which are separated by a *word separator*. This, however, does not mean that you are limited to using only natural language as the decoration text.

0	0	0	-	0	0	1	-	0	1	0	-	0	1	1	-	1	0	0	-	1	0	1	-	1	1	0	-	1	1	1
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

```
\usetikzlibrary {decorations.text}
\begin{tikzpicture}[decoration={text effects along path,
  text={000-001-010-011-100-101-110-111},
  text effects/.cd,
  path from text,
  word separator=-,
  every letter/.style={shape=rectangle, fill=blue!20, draw=blue!40}]

\path [decorate] (0,0);
\end{tikzpicture}
```

In addition, it is possible to replace characters with TikZ code:



```
\usetikzlibrary {decorations.text}
\begin{tikzpicture}[decoration={text effects along path,
  text={000-001-010-011-100-101-110-111}, text align=center,
  text effects/.cd,
  word separator=-,
  replace characters=0 with {\fill [purple] circle [radius=2pt]; },
  replace characters=1 with {\fill [orange] circle [radius=2pt]; },
  replace characters=- with {\path circle [radius=2pt]; },
  every letter/.style={shape=rectangle, fill=blue!20, draw=blue!40}]

\path [decorate] (0,0) .. controls ++(2,0) and ++(-2,0) .. (3,4);
\end{tikzpicture}
```

There are many keys and styles that can be used to add effects to the decoration text. Many of these keys have the parent path `/pgf/decoration/text effects/`, but for convenience, these keys can be accessed using the following key:

`/tikz/text effects={⟨options⟩}` (no default)

Execute every option in `{⟨options⟩}` with the key path for each option temporarily set to `/pgf/decoration/text effects/`.

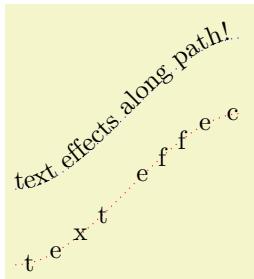
The following keys can be used to customise the appearance of text in the `text effects along path` decoration.

`/pgf/decoration/text effects/every character` (style, no value)

Set the effects that will be applied to every character in the decoration text. The effects will typically be TikZ node options. Initially, this style is empty so the decoration simply positions nodes at the appropriate position along the path. In order to make the text ‘follow the path’ like the `text along path` decoration the following key can be added to the `every character` style.

`/pgf/decoration/text effects/text along path` (style, no value)

This style automatically sets the TikZ keys `transform shape` (to make the character slope with the path), `anchor=baseline` (to make the baseline of the characters ‘sit’ on the path) and `inner xsep=0pt` (to horizontally fit each node to the character it contains, reducing the spacing between characters).



```
\usetikzlibrary {decorations.text}
\begin{tikzpicture}[decoration={text effects along path,
  text={text effects along path!}}]

\path [draw=red, dotted, postaction={decorate}]
  (0,0) .. controls ++(1,0) and ++(-1,0) .. (3,2);
\path [draw=blue, dotted, yshift=1cm, postaction={decorate},
  text effects={text along path}]
  (0,0) .. controls ++(1,0) and ++(-1,0) .. (3,2);
\end{tikzpicture}
```

`/pgf/decoration/text effects/characters={⟨effects⟩}` (no default)

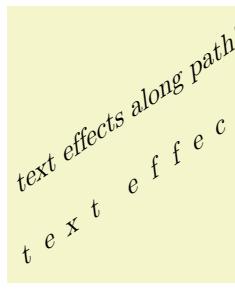
Shorthand for the `every character`.

<code>/pgf/decoration/text effects/character <number></code>	(style, no value)
Specify additional effects for the character <code><number></code> .	
<code>/pgf/decoration/text effects/every letter</code>	(style, no value)
Specify additional effects for every letter (i.e., every character that isn't the word separator) in the decoration text.	
<code>/pgf/decoration/text effects/letter <number></code>	(style, no value)
Specify the effects for letter <code><number></code> in <i>every</i> word.	
<code>/pgf/decoration/text effects/every first letter</code>	(style, no value)
Specify additional effects for the first letter in <i>every</i> word.	
<code>/pgf/decoration/text effects/every last letter</code>	(style, no value)
Specify additional effects for the last letter in <i>every</i> word.	
<code>/pgf/decoration/text effects/every word</code>	(style, no value)
Specify additional effects for every word in the decoration text.	
<code>/pgf/decoration/text effects/word <number></code>	(style, no value)
Specify additional effects for word <code><number></code> in the decoration text.	
<code>/pgf/decoration/text effects/word <m> letter <n></code>	(style, no value)
Specify additional effects for letter <code><n></code> in word <code><m></code> in the decoration text.	
<code>/pgf/decoration/text effects/every word separator</code>	(style, no value)
Specify additional effects for every character that is a word separator.	
<code>/pgf/decoration/text effects/word separator=<character></code>	(no default, initially space)
Specify the character that is to be used as the word separator. This <i>must</i> be a single character such as <code>a</code> or <code>-</code> or the special value <code>space</code> (which should be used to indicate that spaces should be used as the separator).	

By default, the width for each character is calculated according to the bounding box of the node in which it is contained. However, if the node is rotated or slanted, or has a substantial `inner sep`, this bounding box will be quite big. The following key enables different effects to be applied to the node that is used to calculate the width.

<code>/pgf/decoration/text effects/every character width</code>	(style, no value)
This style is applied to the (invisible) nodes used for calculating the width of a character node.	

<code>/pgf/decoration/text effects/character widths={<effects>}</code>	(no default)
Shorthand for the <code>every character width</code> style.	



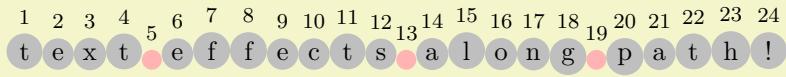
```
\usetikzlibrary {decorations.text}
\begin{tikzpicture}[decoration={text effects along path,
    text={text effects along path!}, text align=center,
    text effects/.cd,
    character count=\i,
    characters={xslant=0.5, text along path, name=c-\i}}]

\path [decorate] (0,0) -- (3,2);
\path [decorate,
    text effects={character widths={inner xsep=0pt, xslant=0}}]
(0,1) -- (3,3);
\end{tikzpicture}
```

It is possible to parameterize effects, perhaps for doing calculations, or labelling nodes based on the number of the character in the decoration text. To access the number of the character, and the total number of characters the following keys can be used. However, these keys should *not* be used inside the style keys given above.

/pgf/decoration/text effects/character count=*macro* (no default)

Store the number of the character being typeset in *macro*.



```
\usetikzlibrary {decorations.text}
\begin{tikzpicture}[decoration={text effects along path,
    text={text effects along path!},
    text effects/.cd,
    path from text,
    character count=\i, every word separator/.style={fill=red!30},
    characters={text along path, shape=circle, fill=gray!50}}]

\path [decorate, text effects={characters/.append={label=above:\footnotesize\i}}] (0,0) ;
\end{tikzpicture}
```

/pgf/decoration/text effects/character total=*macro* (no default)

Store the total number of the characters in the decoration text in *macro*. This key can be used with the **character count** key to produce some quite pleasing effects:



```
\usetikzlibrary {decorations.text,math}
\begin{tikzpicture}[decoration={text effects along path,
    text={text effects along path!},
    text effects/.cd,
    character count=\i, character total=\n,
    characters={text along path, evaluate={\c=\i/\n*100;},
    text=orange!\c!blue, scale=\i/\n+0.5}]

\path [decorate]
(0,0) .. controls ++(1,0) and ++(-1,0) .. (3,2);
\end{tikzpicture}
```

/pgf/decoration/text effects/letter count=*macro* (no default)

Store the number of letter being typeset (i.e., the position of the character in the word) in *macro*. Numbering starts at 1 and the character acting as a word separator is numbered 0.



```
\usetikzlibrary {decorations.text}
\begin{tikzpicture}[decoration={text effects along path,
    text={text effects along path!},
    text effects/.cd,
    path from text, letter count=\i, every word separator/.style={fill=red!30},
    characters={text along path, shape=circle, fill=gray!50}}]

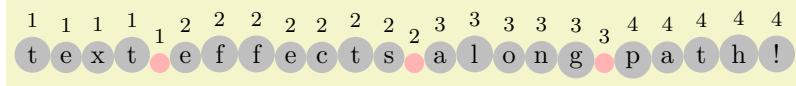
\path [decorate, text effects={characters/.append={label=above:\footnotesize\i}}] (0,0) ;
\end{tikzpicture}
```

/pgf/decoration/text effects/letter total=*macro* (no default)

Store the number of letters in the current word in *macro*. When the character is the word separator, this value is 0.

/pgf/decoration/text effects/word count=*macro* (no default)

Store the number of words in the decoration text in *macro*. Numbering starts at 1. When the character is the word separator, *macro* takes the number of the previous word. If the decoration text starts with a word separator *macro* will be 0.



```
\usetikzlibrary {decorations.text}
\begin{tikzpicture}[decoration={text effects along path,
    text={text effects along path!},
    text effects/.cd,
    path from text, word count=\i, every word separator/.style={fill=red!30},
    characters={text along path, shape=circle, fill=gray!50}}]

\path [decorate, text effects={characters/.append={label=above:\footnotesize\i}}] (0,0);
\end{tikzpicture}
```

/pgf/decoration/text effects/word total=⟨macro⟩

(no default)

Store the total number of words in the decoration text in ⟨macro⟩.

It is also possible to apply effects to specific characters such as coloring every instance of the character a, or changing the font of every T in the decoration text:

/pgf/decoration/text effects/style characters=⟨characters⟩ with {⟨effects⟩} (no default)

This key enables ⟨effects⟩ to be applied to every character in the decoration text that is specified in ⟨characters⟩.

Falsches Üben von Xylophonmusik quält jeden größeren Zwerg

```
\usetikzlibrary {decorations.text}
\begin{tikzpicture}[decoration={text effects along path,
    text={Falsches {"Ü}ben von Xylophonmusik qu{"a}lt jeden gr{"o}ßen Zwerg},
    text effects/.cd,
    path from text,
    style characters=aeiou{"U}{a}{o} with {text=blue},
    characters={text along path}}]

\path [decorate] (0,0);
\end{tikzpicture}
```

/pgf/decoration/text effects/path from text=⟨true or false⟩

(default true)

When this key is set to true and the decorated path consists only of a single point, the decoration will calculate the width of the decoration text using all the specified parameters as if the decorated path was actually a straight line starting from the given point. This ‘virtual’ straight line is then decorated with the text.

text effects along path!

```
\usetikzlibrary {decorations.text}
\begin{tikzpicture}[decoration={text effects along path,
    text={text effects along path!},
    text effects/.cd,
    path from text,
    character count=\i, character total=\n,
    characters={text along path, scale=\i/\n+0.5}}]

\path [decorate] (0,0);
\end{tikzpicture}
```

/pgf/decoration/text effects/path from text angle=⟨angle⟩

(no default)

When used in conjunction with the path from text key, the straight line that is used as the decorated path is rotated by ⟨angle⟩ around the starting point.

text effects along path!

```
\usetikzlibrary {decorations.text}
\begin{tikzpicture}[decoration={text effects along path,
  text={text effects along path!},
  text effects/.cd,
  path from text, path from text angle=60,
  character count=\i, character total=\n,
  characters={text along path, scale=\i/\n+0.5}}]

\path [decorate] (0,0);
\end{tikzpicture}
```

/pgf/decoration/text effects/fit text to path=<true or false>

(default true)

This key will make the decoration increase the space between characters so that the entire path is used by the decoration.

text effects along path!

```
\usetikzlibrary {decorations.text}
\begin{tikzpicture}[decoration={text effects along path,
  text={text effects along path!},
  text effects/every character/.style={text along path}}]

\path [draw=gray, postaction={decorate}, rotate=90]
  (0,0) .. controls ++(2,0) and ++(-1,0) .. (5,-1);
\path [draw=gray, postaction={decorate}, rotate=90, yshift=-1cm,
  text effects={fit text to path}]
  (0,0) .. controls ++(2,0) and ++(-1,0) .. (5,-1);
\end{tikzpicture}
```

/pgf/decoration/text effects/scale text to path=<true or false>

(default true)

This key will make the decoration scale the text so that the entire path is used by the decoration.

text effects along path!

```
\usetikzlibrary {decorations.text}
\begin{tikzpicture}[decoration={text effects along path,
  text={text effects along path!},
  text effects/every character/.style={text along path}}]

\path [draw=gray, postaction={decorate}, rotate=90]
  (0,0) .. controls ++(2,0) and ++(-1,0) .. (5,-1);
\path [draw=gray, postaction={decorate}, rotate=90, yshift=-1cm,
  text effects={scale text to path}]
  (0,0) .. controls ++(2,0) and ++(-1,0) .. (5,-1);
\end{tikzpicture}
```

/pgf/decoration/text effects/reverse text

(no value)

Reverse the order of the characters in the decoration text. This may be useful if using ‘right-to-left’ languages. Unfortunately, any leading ‘soft’ spaces in the original text will be lost.



```
\usetikzlibrary {decorations.text}
\begin{tikzpicture}[decoration={text effects along path,
    text={text effects along path!},
    text effects/.cd,
    path from text, path from text angle=60,
    reverse text,
    character count=\i, character total=\n,
    characters={text along path, scale=\i/\n+0.5}]]

\path [decorate] (0,0) .. controls ++(1,0) and ++(-1,0) .. (3,2);
\end{tikzpicture}
```

It is important to note that the `reverse text` key reverses the text *before* doing anything else. This means that the numbering of characters, letters and words will still be in the normal order, so any parameterized effects will have to take this into account. Alternatively, to get the numbering to follow the reversed text, it is possible to reverse the path and then invert the scale:



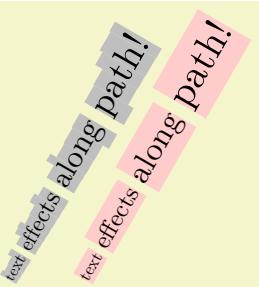
```
\usetikzlibrary {decorations.text}
\begin{tikzpicture}[decoration={text effects along path,
    text={text effects along path!},
    text effects/.cd,
    path from text, path from text angle=60,
    character count=\i, character total=\n,
    characters={text along path, scale=\i/\n+0.5}]]

\path [decorate, text effects={reverse text}] (0,0);
\path [blue, decorate, decoration={reverse path},
    text effects={characters/.append={scale=-1}}] (1,0);
\end{tikzpicture}
```

/pgf/decoration/text effects/group letters

(no value)

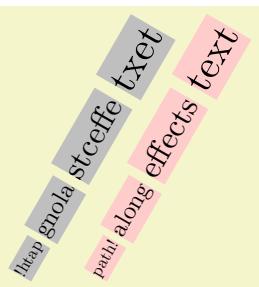
Group sequences of letters together so they are treated as a single ‘character’.



```
\usetikzlibrary {decorations.text}
\begin{tikzpicture}[decoration={text effects along path,
    text={text effects along path!},
    text effects/.cd,
    path from text, path from text angle=60,
    every word separator/.style={fill=none},
    character count=\i, character total=\n,
    characters={text along path, fill=gray!50, scale=\i/\n+0.5}]]

\path [decorate] (0,0);
\path [decorate, text effects={group letters,
    characters/.append={fill=red!20}}]
(1,0);
\end{tikzpicture}
```

The order in which the `reverse text` and `group letters` keys are applied is important:



```
\usetikzlibrary {decorations.text}
\begin{tikzpicture}[decoration={text effects along path,
    text={text effects along path!},
    text effects/.cd,
    path from text, path from text angle=60,
    every word separator/.style={fill=none},
    character count=\i, character total=\n,
    characters={text along path, fill=gray!50, scale=\i/\n+0.5}]]

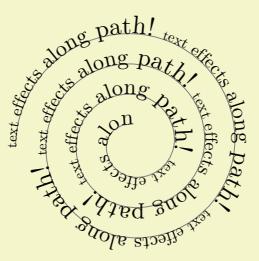
\path [decorate, text effects={reverse text, group letters}] (0,0);
\path [decorate, text effects={group letters, reverse text,
    characters/.append={fill=red!20}}] (1,0);
\end{tikzpicture}
```

/pgf/decoration/text effects/repeat text=<times>

(no default)

Usually, when the decoration runs out of text, it simply stops. This key will make the decoration repeat the decoration text for the specified number of `<times>`. If no value is given the text will be repeated until the path is finished. There are two points to remember however. Firstly the

numbering of characters, letters and words will be restarted each time the text is repeated. Secondly, the options for alignment, scaling or fitting the text to the path, fitting the path to the text, and so on, are computed using the decoration text before the decoration starts. If any of these options are given the behaviour of the `repeat text` key is undefined, but typically it will be ignored.



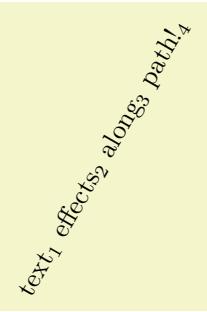
```
\usetikzlibrary {decorations.text}
\begin{tikzpicture}[decoration={text effects along path,
    text={text effects along path!\ },
    text effects/.cd,
    repeat text,
    character count=\m, character total=\n,
    characters={text along path, scale=0.5+\m/\n/2}}]

\path [draw=gray, ultra thin, postaction=decorate]
    (180:2) \foreach \a in {0,...,12}{ arc (180-\a*90:90-\a*90:1.5-\a/10) };
\end{tikzpicture}
```

`/pgf/decoration/text effects/character command=<macro>`

(no default)

This key specifies a command that is executed when each character is placed in the node. The `<macro>` should be an ordinary TeX macro which takes one argument. The argument will be a macro which when expanded will contain the current character.



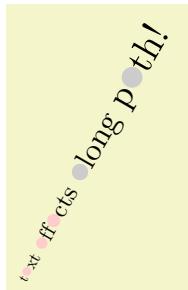
```
\usetikzlibrary {decorations.text}
\def\mycommand{\#1\$\n\$}
\begin{tikzpicture}[decoration={text effects along path,
    text={text effects along path!\ },
    text effects/.cd,
    path from text, path from text angle=60, group letters,
    word count=\n,
    every word/.style={character command=\mycommand},
    characters={text along path}}]

\path [decorate] (0,0);
\end{tikzpicture}
```

`/pgf/decoration/text effects/replace characters=<characters> with {<code>}`

(no default)

Replace the node for each character in `<characters>` with `<code>`. The `<code>` can be thought of as describing a little picture or marking which will be used instead of the character node. The origin will be the current point along the decoration path. Any transformations associated with the `<characters>` (e.g., applied with the `every character` or `every letter` styles) will also be applied to `<code>`.



```
\usetikzlibrary {decorations.text}
\begin{tikzpicture}[decoration={text effects along path,
    text={text effects along path!\ },
    text effects/.cd,
    path from text, path from text angle=60,
    replace characters=e with {\fill [red!20] (0,1mm) circle [radius=1mm];},
    replace characters=a with {\fill [black!20] (0,1mm) circle [radius=1mm];},
    character count=\i, character total=\n,
    characters={text along path, scale=\i/\n+0.5}}]

\path [decorate] (0,0);
\end{tikzpicture}
```

51.7 Fractal Decorations

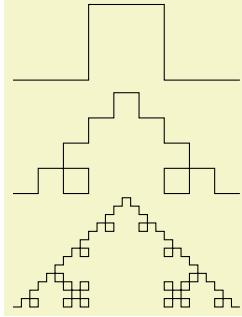
TikZ Library `decorations.fractals`

```
\usepgflibrary{decorations.fractals} % LATEX and plain TEX and pure pgf
\usepgflibrary[decorations.fractals] % ConTEXt and pure pgf
\usetikzlibrary{decorations.fractals} % LATEX and plain TEX when using TikZ
\usetikzlibrary[decorations.fractals] % ConTEXt when using TikZ
```

The decorations of this library can be used to create fractal lines. To use them, you typically have to apply the decoration repeatedly to an originally straight path.

Decoration Koch curve type 1

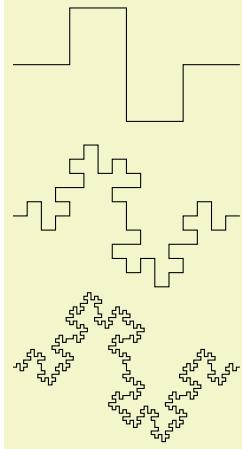
This decoration replaces a straight line by a “rectangular bump”. By repeatedly applying this replacement, different levels of the Koch curve fractal can be created. Its Hausdorff dimension is $\log 5 / \log 3$.



```
\usetikzlibrary {decorations.fractals}
\begin{tikzpicture}[decoration=Koch curve type 1]
  \draw decorate{ (0,0) -- (3,0) };
  \draw decorate{ decorate{ (0,-1.5) -- (3,-1.5) }};
  \draw decorate{ decorate{ decorate{ (0,-3) -- (3,-3) }}};
\end{tikzpicture}
```

Decoration Koch curve type 2

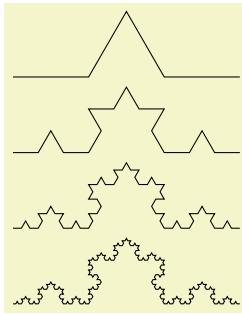
This decoration replaces a straight line by a “rectangular sine”. Its Hausdorff dimension is $3/2$.



```
\usetikzlibrary {decorations.fractals}
\begin{tikzpicture}[decoration=Koch curve type 2]
  \draw decorate{ (0,0) -- (3,0) };
  \draw decorate{ decorate{ (0,-2) -- (3,-2) }};
  \draw decorate{ decorate{ decorate{ (0,-4) -- (3,-4) }}};
\end{tikzpicture}
```

Decoration Koch snowflake

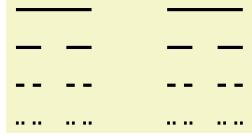
This decoration replaces a straight line by a “line with a spike”. The Hausdorff dimension of Koch’s snowflake’s is $\log 4 / \log 3$.



```
\usetikzlibrary {decorations.fractals}
\begin{tikzpicture}[decoration=Koch snowflake]
  \draw decorate{ (0,0) -- (3,0) };
  \draw decorate{ decorate{ (0,-1) -- (3,-1) }};
  \draw decorate{ decorate{ decorate{ (0,-2) -- (3,-2) }}};
  \draw decorate{ decorate{ decorate{ decorate{ (0,-3) -- (3,-3) }}}};
\end{tikzpicture}
```

Decoration Cantor set

This decoration replaces a straight line by a “line with a gap in the middle”. The Hausdorff dimension of the Cantor set is $\log 2 / \log 3$.



```
\usetikzlibrary {decorations.fractals}
\begin{tikzpicture}[decoration=Cantor set,very thick]
  \draw decorate{ (0,0) -- (3,0)};
  \draw decorate{ decorate{ (0,-.5) -- (3,-.5)}};
  \draw decorate{ decorate{ decorate{ (0,-1) -- (3,-1)}}};
  \draw decorate{ decorate{ decorate{ decorate{ (0,-1.5) -- (3,-1.5)}}}};
\end{tikzpicture}
```

52 Entity-Relationship Diagram Drawing Library

TikZ Library `er`

```
\usetikzlibrary{er} % LATEX and plain TEX  
\usetikzlibrary[er] % ConTEX
```

This package provides styles for drawing entity-relationship diagrams.

This library is intended to help you in creating E/R-diagrams. It defines only few new styles, but using the style `entity` instead of saying `rectangle,draw` makes the code more expressive.

52.1 Entities

The package defines a simple style for drawing entities:

/tikz/entity

(style, no value)

This style is to be used with nodes that represent entity types. It causes the node's shape to be set to a rectangle that is drawn and whose minimum size and width are set to sensible values.

Note that this style is called `entity` despite the fact that it is to be used for nodes representing entity *types* (the difference between an entity and an entity type is the same as the difference between an object and a class in object-oriented programming). If this bothers you, feel free to define a style `entity type` instead.

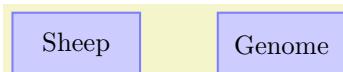


```
\usetikzlibrary {er,positioning}  
\begin{tikzpicture}  
  \node[entity] (sheep) [Sheep];  
  \node[entity] (genome) [right=of sheep] [Genome];  
\end{tikzpicture}
```

/tikz/every entity

(style, no value)

This style is evoked by the style `entity`. To change the appearance of entities, you can change this style.



```
\usetikzlibrary {er,positioning}  
\begin{tikzpicture}  
  [every entity/.style={draw=blue!50,fill=blue!20,thick}]  
  \node[entity] (sheep) [Sheep];  
  \node[entity] (genome) [right=of sheep] [Genome];  
\end{tikzpicture}
```

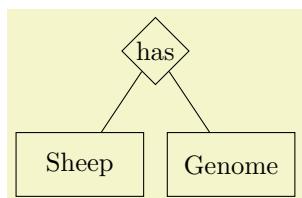
52.2 Relationships

Relationships are drawn using styles that are very similar to the styles for entities.

/tikz/relationship

(style, no value)

This style works like `entity`, only it is to be used for relationships. Again, `relationships` are actually relationship types.

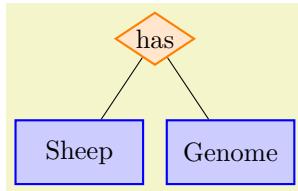


```
\usetikzlibrary {er}  
\begin{tikzpicture}  
  \node[entity] (sheep) at (0,0) [Sheep];  
  \node[entity] (genome) at (2,0) [Genome];  
  \node[relationship] at (1,1.5) [has]  
    edge (sheep)  
    edge (genome);  
\end{tikzpicture}
```

`/tikz/every relationship`

(style, no value)

Works like `every entity`.



```

\usetikzlibrary {er}
\begin{tikzpicture}
[every entity/.style={fill=blue!20,draw=blue,thick},
 every relationship/.style={fill=orange!20,draw=orange,thick,aspect=1.5}]
\node[entity] (sheep) at (0,0) {Sheep};
\node[entity] (genome) at (2,0) {Genome};
\node[relationship] at (1,1.5) {has}
edge (sheep)
edge (genome);
\end{tikzpicture}

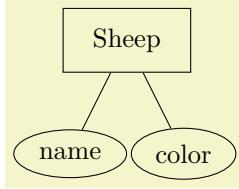
```

52.3 Attributes

`/tikz/attribute`

(style, no value)

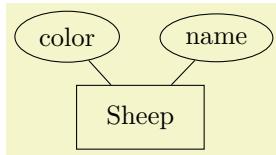
This style is used to indicate that a node is an attribute. To connect an attribute to its entity, you can use, for example, the `child` command or the `pin` option.



```

\usetikzlibrary {er}
\begin{tikzpicture}
\node[entity] (sheep) {Sheep};
\node[attribute] (name) {name};
\node[attribute] (color) {color};
\end{tikzpicture}

```



```

\usetikzlibrary {er}
\begin{tikzpicture}
[every pin edge/.style=draw]
\node[entity,pin={[attribute]60:name},pin={[attribute]120:color}] {Sheep};
\end{tikzpicture}

```

`/tikz/key attribute`

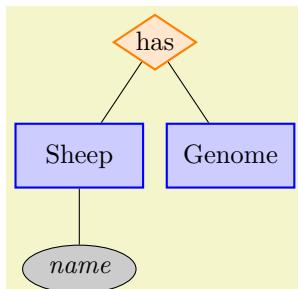
(style, no value)

This style is intended for key attributes. By default, the will cause the attribute to be typeset in italics. Typically, underlining is used instead, but that looks ugly and it is difficult to implement in T_EX.

`/tikz/every attribute`

(style, no value)

This style is used with every attribute, and therefore also for every key attribute.



```

\usetikzlibrary {er}
\begin{tikzpicture}
[text depth=1pt,
every attribute/.style={fill=black!20,draw=black},
every entity/.style={fill=blue!20,draw=blue,thick},
every relationship/.style={fill=orange!20,draw=orange,thick,aspect=1.5}]
\node[entity] (sheep) at (0,0) {Sheep};
\node[entity] (genome) at (2,0) {Genome};
\node[relationship] at (1,1.5) {has}
edge (sheep)
edge (genome);
\node[key attribute] (name) at (0,-1) {name};
\end{tikzpicture}

```

53 Externalization Library

by Christian Feuersänger

TikZ Library `external`

```
\usetikzlibrary{external} % LATEX and plain TEX  
\usetikzlibrary[external] % ConTEXt
```

This library provides a high-level automatic or semi-automatic export feature for TikZ pictures. Its purpose is to convert each picture to a separate PDF without changing the document as such.

It also externalizes `\label` information (and other aux file related stuff) using auxiliary files.

53.1 Overview

There are several reasons why external images for at least some pictures are of interest:

1. Larger picture require a considerable amount of time, which is necessary for every compilation. However, only few images will change from run to run. It can simply save time to export finished images and include them as final graphics.
2. It may be desirable to have final images for some graphics, for example to include them in third-party programs or to communicate them electronically.
3. It may be necessary to typeset a file in environments where PGF and TikZ are not available. In this case, external images are the only way to ensure compatibility.

The purpose of this library is to provide a way to export any TikZ-picture to separate PDF (or `eps`) images without changing the main document. It is actually a simple user interface to the `\begin{pgfgraphicnamed} ... \end{pgfgraphicnamed}` framework of PGF which is discussed in section ??.

53.2 Requirements

For most users, the library does not need special attention since requirements are met anyway. It collects all tokens between `\begin{tikzpicture}` and the next following `\end{tikzpicture}` and replaces them by the appropriate graphics or it takes steps to generate such an image.

It can't expand macros during this step, so the only requirement is that every picture's end is directly reachable from its beginning, without further macro expansion. Furthermore, the library assumes that all L^AT_EX pictures are ended with `\end{tikzpicture}`.

The library always searches for the *next* picture's end, `\end{tikzpicture}`. As a consequence, you can't use nested pictures directly. You *can* nest pictures, but you have to avoid that the nested picture's `\end` command is found before the outer `\end` command (for example using bracing constructs or by writing the nested picture into a separate macro call).

Consider using the `\tikzexternalisable` method in case you'd like to skip selected pictures which do not meet the requirements.

53.3 A Word About Con^TE_X And Plain T_EX

Currently, the basic layer backend `\begin{pgfgraphicnamed} ... \end{pgfgraphicnamed}` relies on L^AT_EX only, so externalization is currently only supported for L^AT_EX.

53.4 Externalizing Graphics

After loading the library, a call to `\tikzexternalize` is necessary to activate the externalization.

```

\documentclass{article}
% main document, called main.tex
\usepackage{tikz}

\usetikzlibrary{external}
\tikzexternalize % activate!

\begin{document}
\begin{tikzpicture}
\node {root}
    child {node {left}}
    child {node {right}
        child {node {child}}
        child {node {child}}}
    ;
\end{tikzpicture}
A simple image is \tikz \fill (0,0) circle(5pt);.
\end{document}

```

The method works as follows: if the document is typeset normally, the library searches for replacement images for every picture. Filenames are generated automatically in the default configuration. In our case, the two file names will be `main-figure0` and `main-figure1`. If they exist, those images are simply included and the pictures as such are not processed. If graphics files do not exist, steps are taken to generate the missing ones. Since (currently) only one output file can be set, each missing image needs to be generated by a separate run of L^AT_EX in which the `\jobname` is set to the desired image file name. In the default configuration `mode=convert with system call`, these commands are issued automatically by using the `\write18` method to call system commands. It is also possible to output every required file name or to generate a `makefile`; users will need to issue the required commands manually (or with `make`). The probably most comfortable way is to use the default configuration with

```
pdflatex -shell-escape main
```

which authorizes `pdflatex` to call itself recursively to generate the images. When it finishes, all images are generated and the document already includes them.

From this point on, successive runs of L^AT_EX will use the final graphics files, the pictures won't be used anymore. Section 53.5 contains details about how to submit such a file to environments where PGF is not available.

`\tikzexternalize[<optional arguments>]`

This command activates the externalization. It installs commands to replace every TikZ-picture. It needs to be called before `\begin{document}` because it may need to install its separate shipout routine.

The `<optional arguments>` can be any of the keys described below.

Note that the generation/modification of auxiliary files like `.aux`, `.toc` etc. is usually suppressed while a single image is externalized (details for `\label` support follow).

It is also possible to write `\tikzexternalize{<main job name>}` if the argument is delimited by curly braces. This case is mainly for backwards compatibility and is no longer necessary. Since it might be useful in rare circumstances, it is documented in section 53.4.5.

A detailed description about the process of externalization is provided in section 53.4.5.

`\tikzexternalrealjob`

After the library is loaded, this macro will *always* contain the correct main job's name (in the example above, it is `main`). It is to be used instead of `\jobname` when the externalization is in effect.

`\pgfactualjobname`

Once `\tikzexternalize` has been called, `\pgfactualjobname` contains the name of the currently generated output file (which may be `main` or `main-figure0` or `main-figure1` in our example above).

`\jobname`

The value of `\jobname` is one of `\tikzexternalrealjob` or `\pgfactualjobname`, depending on the configuration. In short: if auxiliary file support (`\label` and `\ref`) is activated, `\jobname=\tikzexternalrealjob` (since that's the base file name of auxiliary files).

`/tikz/external/system call={⟨template⟩}` (no default)

A template string used to generate system calls. Inside of {⟨template⟩}, the macro `\image` can be used as placeholder for the image which is about to be generated while `\texsource` contains the main file name (in truth, it contains `\input{⟨main file name⟩}`, but that doesn't matter).

The default depends on the value of `\pgfsysdriver`. For `pgfsys-pdfTeX.def`, it is

```
\tikzset{external/system call={pdflatex \tikzexternalcheckshellescape -halt-on-error  
-interaction=batchmode -jobname "\image" "\texsource"}}
```

where `\tikzexternalcheckshellescape` inserts the value of the configuration key `shell escape` if and only if the current document has been typeset with `-shell-escape`⁶.

Other drivers result in slightly different calls. There is support for `lualatex`, `xelatex`, and `dvips`. The precise values are written to the `.log` file as soon as you attempt to compile a document.

The argument {⟨template⟩} will be expanded using `\edef`, so any control sequences will be expanded. During this evaluation, ‘\’ will result in a normal backslash, ‘\’. Furthermore, double quotes “”, single quotes ‘’, semicolons and dashes ‘-’ will be made to normal characters if any package uses them as macros. This ensures compatibility with the `german` package, for example.

`/tikz/external/shell escape={⟨command-line arg⟩}` (no default, initially `-shell-escape`)

Contains the command line option for `latex` which enables the `\write18` feature. For `TEX-Live`, this is `-shell-escape`. For `MiKTEX`, you should use `\tikzexternalize[shell escape=-enable-write18]`.

53.4.1 Support for Labels and References In External Files

The `external` library comes with extra support for `\label` and `\ref` (and other commands which usually store information in the `.aux` file) inside an external files.

In particular, it supports the two use-cases

- `\ref` to something in the main document inside an externalized graphics or
- `\label` in the externalized graphics which is referenced in the main document.

The only restriction is that you need to compile your document multiple times (as usual for references).

NOTE: support for a) is unavailable for versions up to and including PGF 3.0.1.

`/tikz/external/aux in dpth={⟨boolean⟩}` (no default, initially `true`)

Allows to enable or disable the feature which handles references and labels as part of image externalization. Disabling it will save one `\newwrite` command, i.e. a write register.

Also see the `disabled dependency files` feature.

Here are some implementation details on how references within/from external graphics work for those who would like to know the details:

For point a), a `\ref` inside of an externalized graphics works by reading the main document's `.aux` file. To this end, the standard `mode=convert with system call` detects such references and reschedules the externalization to `\end{document}`.⁷ Other values of `mode` require just one attempt to externalize the picture.

Note that `\pageref` is not supported (sorry).

Point b) works as follows: a `\label` inside of an externalized graphics causes the `external` library to generate separate auxiliary files for every external image. These files are called `⟨imagename⟩.dpth`. The extension `.dpth` indicates that the file also contains the image's depth (the `baseline` key of TikZ). Furthermore, anything which would have been written to an `.aux` file will be redirected to the `.dpth` file – but only things which occur inside of the externalized `tikzpicture` environment. When the main document loads the image, it will copy the `.dpth` file into the main `.aux` file. Then, successive compilations of the main document contain the external `\label` information. In other words, a `\label` in an external graphics needs the following work flow:

⁶Note that this is always true for the default configuration. This security consideration applies mainly for `mode=list` and `make` which will also work *without* shell escapes.

⁷Note that this requires the `atveryend` package. The purpose to reschedule the externalization is to access the main job's `.aux` file, but only after it has been written completely.

1. The external graphics needs to be generated together with its `.dpth` (usually automatically by TikZ).
2. The main document includes the external graphics and copies the `.dpth` content into its main `.aux` file.
3. The main document needs to be translated once again to re-read its `.aux` file⁸.

This does also work if a `\label`/`\ref` combination is implemented itself by a `tikzpicture` (a feature offered by `pgfplots`).

53.4.2 Customizing the Generated File Names

The default filename for externalized graphics is ‘`<real file name>-figure_<number>`’ where `<number>` ranges from 0 to whatever is required. However, there are a couple of ways to change the generated filenames:

- Changing the overall file name using a `prefix`,
- Changing the file name for a single figure using `\tikzsetnextfilename`,
- Changing the file name for a restricted set of figures using `figure name`.

`\tikz/external/prefix={<file name prefix>}` (no default, initially `empty`)

A shortcut for `\tikzsetexternalprefix{<file name prefix>}`, see below.

`\tikzsetexternalprefix{<file name prefix>}`

Assigns a common prefix used by all file names. For example,

`\tikzsetexternalprefix{figures/}`

will prepend `figures/` to every external graphics file name.

Please note that `\tikzsetexternalprefix` is the *only* way to assign a prefix in case you want to prepare your document for environments where PGF is not installed (see section 53.5).

`\tikzsetnextfilename{<file name>}`

Sets the file name for the *next* TikZ picture or `\tikz` short command. It will *only* be used for the next picture.

Pictures for which no explicit file name has been set (or the next file name is empty) will get automatically generated file names.

Please note that `prefix` will still be prepended to `{<file name>}`.

```
\documentclass{article}
% main document, called main.tex
\usepackage{tikz}

\usetikzlibrary{external}
\tikzexternalize[prefix=figures/] % activate

\begin{document}

\tikzsetnextfilename{trees}
\begin{tikzpicture} % will be written to 'figures/trees.pdf'
  \node [root]
    child {node [left]}
    child {node [right]
      child {node [child]}
      child {node [child]}}
  ;
\end{tikzpicture}

\tikzsetnextfilename{simple}
A simple image is \tikz \fill (0,0) circle(5pt);. % will be written to 'figures/simple.pdf'

\begin{tikzpicture} % will be written to 'figures/main-figure0.pdf'
  \draw[help lines] (0,0) grid (5,5);
\end{tikzpicture}
\end{document}
```

⁸Note that it is not possible to activate the content of an auxiliary file after `\begin{document}` in L^AT_EX.

```
pdflatex -shell-escape main
```

`/tikz/external/figure name={⟨name⟩}` (no default)
Same as `\tikzsetfigurename{⟨name⟩}`.

`\tikzsetfigurename{⟨name⟩}`

Changes the names of *all* following figures. It is possible to change `figure name` during the document either using `\tikzset{external/figure name={⟨name⟩}}` or with this command. A unique counter will be used for each different `{⟨name⟩}`, and each counter will start at 0.

The value of `prefix` will be applied after `figure name` has been evaluated.

```
\documentclass{article}
% main document, called main.tex
\usepackage{tikz}

\usetikzlibrary{external}
\tikzexternalize % activate

\begin{document}

\begin{tikzpicture} % will be written to 'main-figure0.pdf'
\node {root}
  child {node {left}}
  child {node {right}
    child {node {child}}
    child {node {child}}}
;
\end{tikzpicture}
{
\tikzsetfigurename{subset_}
A simple image is \tikz \fill (0,0) circle(5pt);. % will be written to 'subset_0.pdf'

\begin{tikzpicture} % will be written to 'subset_1.pdf'
\draw[help lines] (0,0) grid (5,5);
\end{tikzpicture}
% here, the old file name will be restored:

\begin{tikzpicture} % will be written to 'main-figure1.pdf'
\draw (0,0) -- (5,5);
\end{tikzpicture}
\end{document}
```

The scope of `figure name` ends with the next closing brace.

Remark: Use `\tikzset{external/figure name/.add={⟨prefix_⟩}{⟨suffix_⟩}}` to add a `prefix_` and a `_suffix_` to the actual value of `figure name`.

`\tikzappendtofigurename{⟨suffix⟩}`

Appends `⟨suffix⟩` to the actual value of `figure name`.

It is a shortcut for `\tikzset{external/figure name/.add={⟨suffix⟩}}` (a shortcut which is also supported if TikZ is not installed, see below).

53.4.3 Remaking Figures or Skipping Figures

`\tikzpicturedependsonfile{⟨file name⟩}`

Adds a dependency for the *next* picture which is about to be externalized. If the command is invoked within a picture environment, it adds a dependency for the surrounding picture. Dependencies are written into `⟨target file⟩.dep` in the format

`⟨target file⟩.\tikzexternalimgextension: ⟨file name⟩.`

The effect is that if `⟨file name⟩` changes, the external graphics associated with the picture shall be remade.

This command uses the contents of `\tikzexternalimgextension` to check for graphics. If you encounter difficulties with image extensions, consider redefining this macro (after `\tikzexternalize`).

Limitations: this command is currently only supported for `mode=list and make` and the generated `makefile`.

\tikzexternalfiledependsonfile{\langle external graphics\rangle}{\langle file name\rangle}

A variant of `\tikzpicturedependsonfile` which adds a dependency for an `\langle external graphics\rangle`. The argument `\langle external graphics\rangle` must be the path as it would have been generated by the `external` library, i.e. without file extension but including any prefixes.

/tikz/external/disable dependency files

(no value)

Allows to (irreversibly) disable the generation of file dependencies. Disabling it will safe one `\newwrite` command, i.e. a write register. Note that the write register is only allocated if the feature has been used at all. This key needs to be provided as argument to `\tikzexternalize` (or it needs to be set before calling `\tikzexternalize`).

Also see the `aux in dpth` key.

/tikz/external/force remake={\langle boolean\rangle}

(default `true`)

A boolean which is used to customize the up-to-date checks of all following figures. Every up-to-date check will fail, resulting in automatic regeneration of every following figure.

```
\tikzset{external/force remake}
\begin{tikzpicture}
  \draw (0,0) circle(5pt);
\end{tikzpicture}
```

You can also use `force remake` inside of a local T_EX group to remake only selected pictures. The example

```
\tikz \draw (0,0) -- (1,1);

{
\tikzset{external/force remake}
\begin{tikzpicture}
  \draw (0,0) circle(5pt);
\end{tikzpicture}
}

\tikz \draw (0,0) -- (1,1);
```

will only apply `force remake` to the second figure.

Up-to-date checks are applied for `mode=convert with system call` and the `makefile` generated by `mode=list and make`.

/tikz/external/remake next={\langle boolean\rangle}

(default `true`)

A variant of `force remake` which applies only to the next image.

/tikz/external/export next={\langle boolean\rangle}

(default `true`)

A boolean which can be used to disable the export mechanism for single pictures.

/tikz/external/export={\langle boolean\rangle}

(no default, initially `true`)

A boolean which can be used to disable the export mechanism for all pictures inside of the current T_EX-scope.

```

\begin{document}
\begin{tikzpicture} % will be exported
...
\end{tikzpicture}

{
\tikzset{external/export=false}
\begin{tikzpicture} % won't be exported
...
\end{tikzpicture}
...

\begin{tikzpicture} % will be exported
...
\end{tikzpicture}
\end{document}

```

For L^AT_EX, the feature lasts until the next `\end{<...>}` (this holds for every call to `\tikzset`).

/tikz/external/up to date check={⟨choice⟩} (no default, initially `md5`)

The `external` lib has to decide when some existing figure is up-to-date. In such a case, it can be used without remaking it. Outdated pictures will be remade.

The key `up to date check` allows to choose among a couple of heuristics which are supposed to catch the most important reasons to remake a figure.

The `up to date check` can be overrule by any of the `force remake` or `remake next` keys: if one of them is true, the figure is not up-to-date.

The choice `simple` is based on the existence of the file: the file is up-to-date if and only if it exists.

The choice `md5` generates an MD5 checksum of the picture for which the up-to-date check is running. The MD5 is compared against the MD5 of the previous run, which, in turn, will be written into an extra file with the extension `.md5`. This file will be modified if and only if the MD5 comparison indicates a difference. The MD5 computation is based on the pdfT_EX method `\pdfmdfivesum`. If it is unavailable for some reason, the choice `diff` will be used instead.

The choice `diff` is the same as MD5 – except that it compares the picture content as-is instead of a hash. The `.md5` file will be used to compare an old version with the current one – but its content is some “normalized” version of the picture for internal use.

Attention: the content-based strategies `md5` and `diff` operate on the picture content – and only on the picture content. Here, “picture content” only includes the top-level tokens; no expansion is applied and no included files are part of the strategies. If you change preamble styles, you have to rebuild the figures manually (for example by deleting the generated graphics files). If you have include files, consider using `\tikzpicture depends on file` and its variants. Since this key provides heuristics, you should always remake your figures before you finally publish your document. Example: Suppose we have the following picture which depends on a command `\mycommand`:

```

\def\mycommand{My comment}

\begin{tikzpicture}
\node at (0,0) {\mycommand};
\end{tikzpicture}

```

What happens if you change “My comment” to “My super comment”? Well, `external` will *not* pick it up; you will need to handle this manually. However, if you modify anything between `\begin{tikzpicture}` and `\end{tikzpicture}`, the `external` library *will* pick it up and regenerate the picture.

The `up to date check` is applied for `mode=convert with system call` and `mode=list and make`.

\tikzexternaldisable

Allows to disable the complete externalization. While `export next` will still collect the contents of picture environments, this command uninstalls the hooks for the `external` library completely. Thus,

nested picture environments or environments where `\end{tikzpicture}` is not directly reachable won't produce compilation failures – although it is not possible to externalize them automatically.

The externalization remains disabled until the end of the next TeX group (or environment) or until the next call to `\tikzexternalenable`.

`\tikzexternalenable`

Re-enables a previously running externalization after `\tikzexternaldisable`.

53.4.4 Customizing the Externalization

`/tikz/external/figure list={⟨boolean⟩}` (no default, initially `true`)

A boolean which configures whether a figure list shall be generated. A figure list is an output file named `{⟨jobname⟩}.figlist` which is filled with file names of each figure, one per line.

This file is not used by TeX anymore, its purpose is to issue the required conversion commands `pdflatex -jobname {⟨picture file name⟩} {⟨main file⟩}` manually (or in a script). See section 53.4.5 for the details about the expected system call (or activate `mode=convert with system call` and inspect your log file).

```
\documentclass{article}
% main document, called main.tex
\usepackage{tikz}

\usetikzlibrary{external}
\tikzexternal[
    mode=graphics if exists,
    figure list=true,
    prefix=figures/]

\begin{document}

\tikzsetnextfilename{trees}
\begin{tikzpicture}
\node {root}
    child {node {left}}
    child {node {right}}
    child {node {child}}
    child {node {child}}
;
\end{tikzpicture}

\tikzsetnextfilename{simple}
A simple image is \tikz \fill (0,0) circle(5pt);.

\begin{tikzpicture}
\draw[help lines] (0,0) grid (5,5);
\end{tikzpicture}
\end{document}
```

```
pdflatex main
```

generates `main.figlist` containing

```
figures/trees
figures/simple
figures/main-figure0
```

`/tikz/external/mode={⟨choice⟩}` (no default, initially `convert with system call`)

Configures what to do with TikZ pictures (unless we are currently externalizing one particular image, in that case, these modes are ignored).

The preconfigured mode `convert with system call` checks whether external graphics files are up-to-date and includes them if that is the case. Any picture which is not up-to-date will be generated automatically using a system call. The system call can be configured using the `system call` template. The up-to-date check is applied according to the `up to date check` key. As soon as `convert with`

system call is set, the `figure` list will be disabled – such a file is not required. In case you still need or want it, you can enable it after setting `mode`.

Please note that system calls may be disabled for security reasons. For pdflatex, they can be enabled using

```
pdflatex -shell-escape
```

while other TeX variants may need other switches. The feature is sometimes called `\write18`.

The choice `only graphics` always tries to replace pictures with external graphics. It is an error if the graphics file does not exist.

The choice `no graphics` (or, equivalently, `only pictures`) typesets TikZ pictures without checking for external graphics.

A mixture is `graphics if exists`, it checks whether a suitable graphics file exists and includes it if that is the case. If it does not exist, the picture is typeset using TeX.

Mode `list` `only` skips every TikZ picture; it only generates the file `{(main file)}.figlist` containing file names for every picture, the contents of any picture environment is thrown away and a replacement text is shown. This implies `figure list=true`. See also the `list` and `make` mode which includes available graphics.

The mode `list` and `make` is similar to `list` `only`: it generates the same file `{(main file)}.figlist`, but any images which exist already are included as graphics instead of ignoring them. Furthermore, this mode generates an additional file: `{(main file)}.makefile`. This allows to use a work flow like

```
% step 1: generate main.makefile:  
pdflatex main  
% step 2: generate ALL graphics on 2 processors:  
make -j 2 -f main.makefile  
% step 3: include the graphics:  
pdflatex main
```

This last make method is optional: `list` and `make` just assumes that images are generated somehow (not necessarily with the generated makefile). The generated makefile allows parallel externalization of graphics on multi-core systems and it supports any file dependencies configured with `\tikzpicturedependsonfile`. Furthermore, it respects the `force remake` and `remake next` keys.

`/tikz/external/verbose IO={⟨boolean⟩}` (no default, initially `true`)

A boolean which configures whether I/O operations shall be listed in the logfile.

`/tikz/external/verbose optimize={⟨boolean⟩}` (no default, initially `true`)

A boolean which configures whether optimization operations shall be listed in the logfile.

`/tikz/external/verbose={⟨boolean⟩}` (no default, initially `true`)

Sets all verbosity flags to `⟨boolean⟩`.

`/tikz/external/optimize={⟨boolean⟩}` (no default, initially `true`)

Configures whether the conversion process shall be optimized. This affects only the case when `\jobname` differs from the main file name, i.e. when single pictures are converted.

In that case, the main file is compiled as usual – but everything except the selected picture is thrown away. If optimization is enabled, all other pictures won't be processed at all. Furthermore, expensive commands which do not contribute to the selected picture will be thrown away as well.

The default implementation discards `\includegraphics` commands which are *not* inside of the selected picture to reduce conversion time.

It is possible to add commands which shall be optimized away, see below.

`/tikz/external/optimize command away={⟨command⟩}{⟨required argument count⟩}` (no default)

Installs commands to optimize `⟨command⟩` away. As is described above, optimization applies to the case when single pictures are converted: one usually doesn't need to process (probably expensive) commands which do not contribute to the selected picture.

The argument `⟨required argument count⟩` is either empty or a non-negative integer between 0 and 9. It denotes the number of arguments which should be consumed after `⟨command⟩`. In any case, one

argument in square brackets after the command will be recognized as well. To be more precise, the following cases for arguments of `\|command` are supported:

1. If `\{<required argument count>\}` is empty (the default), `\|command` may take one optional argument in square brackets and one in curly braces (which is also optional).
2. If `\{<required argument count>\}` is not empty, `\{<command>\}` may take one optional argument in square brackets. Furthermore, it expects exactly `\{<required argument count>\}` following arguments.

Example:

```
\tikzset{external/optimize command away=\includegraphics}

\newcommand{\myExpensiveMacro}[1]{Very expensive!}

\tikzset{external/optimize command away=\myExpensiveMacro}

\newcommand{\myExpensiveMacroWithThreeArgs}[3]{Very expensive!}

\tikzset{external/optimize command away=\myExpensiveMacroWithThreeArgs[3]}

% A command with optional argument:
\newcommand{\aFurtherExample}[3]{\Very expensive!}

% consume only two arguments: the first optional one will be processed
% anyway:
\tikzset{external/optimize command away=\myExpensiveMacroWithThreeArgs[2]}
```

The argument `\|command` must be the name of a single macro. Any occurrence of this macro, together with its arguments, will be removed.

```
\begin{tikzpicture}
    % this picture is currently converted!
\end{tikzpicture}

This here is outside of the converted picture and contains \myExpensiveMacro. It will be discarded.

This call: \myExpensiveMacro [argument=value]{Argument} as well.
And this here: \myExpensiveMacro{Argument} also.
```

The default is to optimize `\includegraphics` away.

This key is actually a style which sets the `optimize/install` and `optimize/restore` keys.

`/tikz/external/optimize/install` (no value)

A command key which contains code to install optimizations. You can append code here (or clear the macro) if you need to modify the optimization.

`/tikz/external/optimize/restore` (no value)

A command key which contains code to undo optimizations. You can append code here (or clear the macro) if you need to modify the optimization.

`/tikz/external/only named={<boolean>}` (no default, initially `false`)

If enabled, only pictures for which file names have been set explicitly using `\tikzsetnextfilename` will be considered, no file names will be generated automatically.

`/pgf/images/include external` (initially `\pgfimage{#1}`)

This command key constitutes the public interface to exchange the `\includegraphics` command used for the image inclusion. It can be overwritten using `include external/.code={<TEX code>}`.

Its description can be found in the corresponding basic layer documentation on page ??.

Just one example here: you can use

```
\pgfkeys{/pgf/images/include external/.code=\includegraphics[viewport=0 0 211.28 175.686]{#1}}
```

to manually change the viewport (bounding box) for included graphics.

Another example (of probably limited use) is

```
\pgfkeys{/pgf/images/include external/.code={\href{file:#1}{\pgfimage[#1]}}}
```

which will generate a clickable hyperlink around the image. Clicking on it opens the single exported file⁹.

If you want to limit the effects of this key to just one externalized figure, use

```
{
  \pgfkeys{/pgf/images/include external/.code={\includegraphics[viewport=0 0 211.28 175.686]{#1}}}
  \begin{tikzpicture}
    ...
  \end{tikzpicture}
} % this brace ends the effect of 'include external'
```

`\tikzifexternalizing{\langle true code\rangle}{\langle false code\rangle}`

This command can be used to check whether an image is currently written to its separate graphics file (if the “grab” procedure is running). If so, the `\tikzifexternalizing{\langle true code\rangle}{\langle false code\rangle}` will be executed. If not, that means if the main document is being typeset normally, the `\tikzifexternalizing{\langle false code\rangle}{\langle true code\rangle}` will be invoked.

This command must be used *after* `\tikzexternalize`.

`\tikzifexternalizingnext{\langle true code\rangle}{\langle false code\rangle}`

Like `\tikzifexternalizing`, but this variant also checks if the next following figure is the one which is about to be written to its separate graphics file.

53.4.5 Details About The Process

The standard run `pdflatex <main document>` causes the `external` library to check every occurrence of `\begin{tikzpicture}` and every `\tikz` short command. If it finds a picture which shall be exported, it queries the respective file name and checks whether the file exists already. If so, it includes the external graphics. If not, it requires an externalization which can be done automatically (the default), semi-automatically (with `mode=list and make`) or manually (by issuing the requires system calls somehow).

The library can detect whether it runs in “conversion mode”, i.e. if it should only process a single image. To do so, it checks whether the internal macro `\tikzexternalrealjob` exists. If so, its contents is assumed to be `<main document>` (without the suffix `.tex`). Usually, this macro is set by the conversion system call,

```
pdflatex -jobname "main-figure0" "\def\tikzexternalrealjob{main}\input{main}"
```

where `main-figure0` is the picture we are currently externalizing and `main.tex` is the main document.

As soon as “conversion mode” has been detected, PGF changes the output routine. The complete file `main.tex` is processed as normal, but only the part of the desired picture will be written to the output file, in our case `main-figure0.pdf`. The rest of the document is silently thrown away. Of course, such a conversion process is quite expensive since we need to do it for every picture. Since everything except the current picture is thrown away, the library skips all other pictures. Furthermore, any `\includegraphics` commands which are outside of the converted TikZ-picture will be skipped as well. Thus, the conversion process should be much faster than typesetting the complete document, but it still requires its time. Eventually, the call `\input{main}` returns and the picture is ready. From this point on, the external graphics will be used.

There is another possibility to communicate `<main document>` to the subprocess performing the externalization: namely to write ‘`\tikzexternalize{main}`’ into the document. In this case, the conversion system call will be

```
pdflatex -jobname "main-figure0" "main"
```

and the contents of `\tikzexternalrealjob` is set automatically. This case is detected by `\tikzexternalize`, and the `system call` is updated automatically (by patching its `\texsource` template argument). It is not necessary to change the `system call` manually.

The sequence in which system calls are performed and the decision whether they are issued automatically is governed by the `mode` key, consult its documentation for details.

⁹This requires all external graphics files in the same base directory as the main `.pdf` file.

53.5 Using External Graphics Without PGF Installed

Given that every picture has been exported correctly, one may want to compile a file without PGF and TikZ installed. TikZ comes with a minimal package which contains just enough commands to replace every `tikzpicture` environment and the `\tikz` short command with the appropriate external graphics. It can be found at

```
latex/pgf/utilities/tikzexternal.sty
```

and needs to be used instead of `\usepackage{tikz}`. So, we uncomment `\usepackage{tikz}` and our example from the beginning becomes

```
\documentclass{article}
% main document, called main.tex
%\usepackage{tikz}

\usepackage{graphicx}
\usepackage{tikzexternal}

%\usetikzlibrary{external}
\tikzexternalize

\begin{document}
\begin{tikzpicture}
\node {root}
    child {node {left}}
    child {node {right}
        child {node {child}}
        child {node {child}}}
    };
\end{tikzpicture}

A simple image is \tikz \fill (0,0) circle(5pt);.

Furthermore, we might want to draw \tikz[baseline]\draw (0,-1) rectangle (1,1);
\end{document}
```

where the following files are necessary to compile the document:

```
tikzexternal.sty
main.tex
main-figure0.pdf
main-figure1.pdf
main-figure2.pdf
```

If there are any ‘.dpth’ files, for example `main-figure2.dpth`, these files are also required. They contain information for the TikZ `baseline` option (or `\labels` inside external graphics).

Just copy the `.sty` file into the directory of your `main.tex` file and use it as part of your document.

Please keep in mind, that only `tikzpicture` environments and `\tikz` short images are available within the externalization framework. Additionally, calls to `\tikzset` and `\pgfkeys` won’t lead to compilation errors because they are simply ignored. But since `pgfkeys` is not available, any option supplied to `\tikzexternalize` is *ignored*.

Attention: Since the simple replacement `\usepackage{tikzexternal}` doesn’t support the key–value interface, you *need* to use `\tikzsetexternalprefix` instead of the `prefix` option and `\tikzsetfigurename` instead of the `figure name` option since `\tikzset` is not available in such a context.

Remark: Some of the features of this library are mainly useful to improve the speed of successive document compilations. In other words: you can’t use all features in this context, keep it simple.

53.6 eps Graphics Export

It is also possible to use `eps` graphics instead of PDF files. There are different ways to produce them, for example to use `pdflatex` and call `pdftops -eps {\pdf file} {\eps file}` afterwards. You could add this command to the `system call` option.

Alternatively, you can use `latex` and `dvi` for image conversion as is explained for the `system call` option, see page 515. See the documentation for the basic level externalization in section ?? for restrictions of other drivers.

53.7 Bitmap Graphics Export

Occasionally, you may have an extremely large graphics which takes long times to render. It might be interesting to generate a bitmap (raster) image, which displays much faster (for example in a presentation). I have used this feature to speed-up the display of large shadings.

The `external` library can be customized to export bitmap images – with the help of external programs. Due to the dependence of external programs, you may need to adjust these commands manually. For example, on my computer, the ImageMagick Suite is installed which comes with the `convert` tool. Together with `pdflatex`, I can define the following style:

```
\tikzset{
    % Defines a custom style which generates BOTH, .pdf and .png export
    % but prefers the .png on inclusion.
    %
    % This style is not pre-defined, you may need to copy-paste and
    % adjust it.
    png export/.style={%
        external/system call/.add=
            {
                \tikzset{external info,
                    include external/.code=\%
                        \includegraphics[width=\pgfexternalwidth,height=\pgfexternalheight]{##1.png}}
            },
    }
}
```

The example above defines a new style called ‘`png export`’ which, when it is set with `\tikzset{png export}` somewhere in the document, modifies the configuration for both file generation and file input. The file generation is modified by appending the ImageMagick command to `system call` (separated by ‘;’ as usual on Linux). This is, in principle, enough to generate a `.png` file. The `include external` command is overwritten such that it uses the `.png` file instead of the `.pdf` file (which exists as well in the configuration above). But since a `.png` file can have a much higher resolution than the desired image dimensions, we have to add `width` and `height` explicitly. Usually, the `external` library does not provide size information (it is unnecessary for `.pdf` or `.eps` since these formats have their bounding box information). To enable size information, the style uses the `external info` key, which, in turn, provides the `\pgfexternalwidth` and `\pgfexternalheight` commands.

Now we can use `\tikzset{png export}` either document-wide or just for one particular image. The configuration remains in effect until the end of the current environment (or until the next closing curly brace ‘`}`’).

`/pgf/images/external info={⟨boolean⟩}` (no default, initially `false`)

If this key is activated, the size for any externalized image will be stored explicitly into the associated `.dpth` file.

When the file is included by `\pgfincludeexternalgraphics` (or automatically by the `external` library), the width is available as `\pgfexternalwidth` and the height as `\pgfexternalheight`.

53.8 Compatibility Issues

53.8.1 References In External Pictures

It is allowed if a picture contains references, for example `\tikz \node {Reference to \ref{a:label}};`.

There is just one issue: if the main job is currently compiling, its `.aux` file is not in its final state (even worse: it may not be readable at all). The picture externalization, however, needs the main `.aux` file to query any references.

Thus, you *will* need to invoke `pdflatex -jobname ⟨image⟩ ⟨mainfile⟩ manually` for any image which contains references.

This problem arises only for `mode=convert with system call`. In this case, the `external` library creates a special `\jobname.auxlock` file to check whether the main `.aux` file is currently usable.

53.8.2 Compatibility With Other Libraries or Packages

The `external` library has the following compatibility issues:

1. The `external` library comes with special support for `\usetikzlibrary{fadings}`: the `fadings` library may define local pictures which would be externalized (although they shouldn't). There is special handling to suppress this bug if `\tikzexternalize` is called *after* `\usetikzlibrary{fadings}` or if all fadings are defined *before* `\tikzexternalize`.
2. Problems have been reported when using `\tikzexternalize` (or the basic layer externalization) together with `\usepackage{glossary}`. This problem disappears if `\tikzexternalize` is called *before* `\usepackage{glossary}`.
3. Problems with `\usepackage{pdfpages}` and `\usepackage{vmargin}`: The `external` library replaces the current shipout routine of TeX during its externalization. This might raise problems with other packages which also manipulate the shipout routine (like the mentioned ones). To fix those problems, use

```
\usetikzlibrary{external}

\tikzifexternalizing{%
    % don't include package XYZ here
}{%
    \usepackage{pdfpages}
    \usepackage{vmargin}
    ...
}%
```

This uses the requested packages for the main document, but not for the single, exported graphics.

In general, the `\tikzifexternalizing` feature might be used to solve package conflicts and the `\tikzexternalisable` and `\tikzexternalenable` features can be used to solve problems with single pictures.

53.8.3 Compatibility With Bounding Box Restrictions

Bounding box restrictions provide no problem when used with `eps` graphics. However, they pose problems for `pdflatex`, so you may need to use the `latex/dvips` combination if you use bounding box restrictions and externalization. Currently, the only possibility for bounding box restrictions and `pdflatex` is to use a combination of `trim left/trim right/baseline`: these keys do not *really* truncate the bounding box, they only store horizontal and vertical shifts (also see the `trim lowlevel` key in this context).

53.8.4 Interoperability With The Basic Layer Externalization

This library is fully compatible with `\begin{pgfgraphicnamed}... \end{pgfgraphicnamed}` environments. However, you will need to use the `export next=false` key to avoid conflicts:

```
\begin{pgfgraphicnamed}[picture4]
\tikzset{external/export next=false}
\begin{tikzpicture}
    \draw (0,0) -- (4,4);
\end{tikzpicture}
\end{pgfgraphicnamed}
```

Please keep in mind that file prefixes do not apply to the basic layer.

54 Fading Library

TikZ Library `fadings`

```
\usepgflibrary{fadings} % LATEX and plain TEX and pure pgf
\usepgflibrary[fadings] % ConTeXt and pure pgf
\usetikzlibrary{fadings} % LATEX and plain TEX when using TikZ
\usetikzlibrary[fadings] % ConTeXt when using TikZ
```

The package defines a number of fadings, see Section 23 for an introduction. The TikZ version defines special TikZ commands for creating fadings. These commands are explained in Section 23.

<i>Fading name</i>	<i>Example (solid blue faded on checkerboard)</i>
<code>west</code>	
<code>east</code>	
<code>north</code>	
<code>south</code>	
<code>circle with fuzzy edge 10 percent</code>	
<code>circle with fuzzy edge 15 percent</code>	
<code>circle with fuzzy edge 20 percent</code>	
<code>fuzzy ring 15 percent</code>	

55 Fitting Library

TikZ Library `fit`

```
\usetikzlibrary{fit} % LATEX and plain TEX
\usetikzlibrary[fit] % ConTeXt
```

The library defines (currently only two) options for fitting a node so that it contains a set of coordinates.

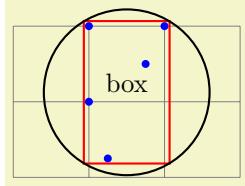
When you load this library, the following options become available:

`/tikz/fit=⟨coordinates or nodes⟩` (no default)

This option must be given to a `node` path command. The `⟨coordinates or nodes⟩` should be a sequence of TikZ coordinates or node names, one directly after the other without commas (like with the `plot coordinates` path operation). Examples are `(1,0)` `(2,2)` or `(a)` `(1,0)` `(b)`, where `a` and `b` are nodes.

For this sequence of coordinates, a minimal bounding box is computed that encompasses all the listed `⟨coordinates or nodes⟩`. For coordinates in the list, the bounding box is guaranteed to contain this coordinate, for nodes it is guaranteed to contain the `east`, `west`, `north` and `south` anchors of the node. In principle (the details will be explained in a moment), things are now set up such that the text box of the node will be exactly this bounding box.

Here is an example: We fit several points in a rectangular node. By setting the `inner sep` to zero, we see exactly the text box of the node. Then we fit these points again in a circular node. Note how the circle encompasses exactly the same bounding box.



```
\usetikzlibrary {fit}
\begin{tikzpicture}[inner sep=0pt, thick,
dot/.style={fill=blue,circle,minimum size=3pt}]
\draw[help lines] (0,0) grid (3,2);
\node[dot] (a) at (1,1) {};
\node[dot] (b) at (2,2) {};
\node[dot] (c) at (1,2) {};
\node[dot] (d) at (1.25,0.25) {};
\node[dot] (e) at (1.75,1.5) {};

\node[draw=red, fit=(a) (b) (c) (d) (e)] {box};
\node[draw,circle,fit=(a) (b) (c) (d) (e)] {};
\end{tikzpicture}
```

Every time the `fit` option is used, the following style is also applied to the node:

`/tikz/every fit` (style, initially empty)

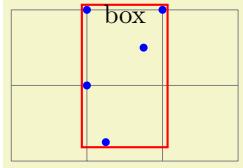
Set this style to change the appearance of a node that uses the `fit` option.

The exact effects of the `fit` option are the following:

1. A minimal bounding box containing all coordinates is computed. Note that if a coordinate like `(a)` is used that contains a node name, this has the same effect as explicitly providing the `(a.north)` and `(a.south)` and `(a.west)` and `(a.east)`. If you wish to refer only to the center of the `a` node, use `(a.center)` instead.
2. The `text width` option is set to the width of this bounding box.
3. The `align=center` option is set.
4. The `anchor` is set to `center`.
5. The `at` position of the node is set to the center of the computed bounding box.
6. After the node has been typeset, its height and depth are adjusted such that they add up to the height of the computed bounding box and such that the text of the node is vertically centered inside the box.

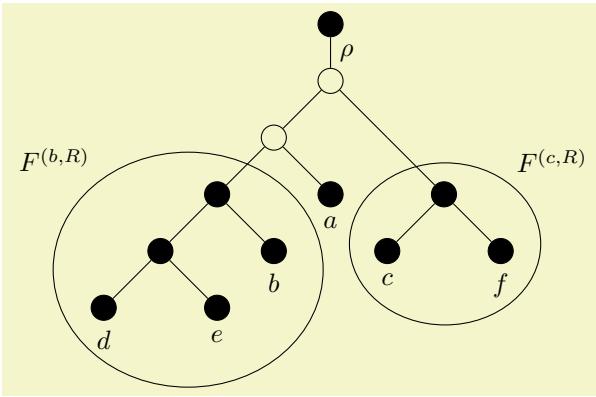
The above means that, generally speaking, if the node contains text like `box` in the above example, it will be centered inside the box. It will be difficult to put the text elsewhere, in particular, changing the `anchor` of the node will not have the desired effect. Instead, what you should do is to create a node with the `fit` option that does not contain any text, give it a name, and then use normal nodes to add text at the desired positions. Alternatively, consider using the `label` or `pin` options.

Suppose, for instance, that in the above example we want the word “box” to appear inside the box, but at its top. This can be achieved as follows:



```
\usetikzlibrary {fit}
\begin{tikzpicture}[inner sep=0pt, thick,
dot/.style={fill=blue,circle,minimum size=3pt}]
\draw[help lines] (0,0) grid (3,2);
\node[dot] (a) at (1,1) {};
\node[dot] (b) at (2,2) {};
\node[dot] (c) at (1,2) {};
\node[dot] (d) at (1.25,0.25) {};
\node[dot] (e) at (1.75,1.5) {};
\node[draw=red, fit=(a) (b) (c) (d) (e)] (fit) {};
\node[below] at (fit.north) {box};
\end{tikzpicture}
```

Here is a real-life example that uses fitting:



```
\usetikzlibrary {fit,shapes.geometric}
\begin{tikzpicture}
[vertex/.style={minimum size=2pt,fill,draw,circle},
open/.style={fill=None},
sibling distance=1.5cm, level distance=.75cm,
every fit/.style={ellipse, draw, inner sep=-2pt},
leaf/.style=[label={[name=#1]below:$#1$},auto]

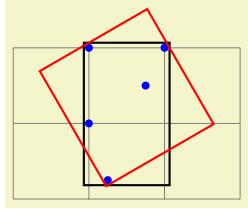
\node [vertex] (root) {}
child { node [vertex,open] {}
    child { node [vertex,open] {}
        child { node [vertex] (b's parent) {}
            child { node [vertex,leaf=d] {} } }
            child { node [vertex,leaf=e] {} } }
            child { node [vertex,leaf=b] {} } }
            child { node [vertex,leaf=a] {} } }
            child { node [coordinate] {}
                child[missing]
                child { node [vertex] (f's parent) {}
                    child { node [vertex,leaf=c] {} } }
                    child { node [vertex,leaf=f] {} } } }
                    edge from parent node {\$\\rho\$};

\node [fit=(d) (e) (b) (b's parent),label=above left:$F^{(b,R)}\$"] {};
\node [fit=(c) (f) (f's parent),label=above right:$F^{(c,R)}\$"] {};
\end{tikzpicture}
```

`/tikz/rotate fit=<angle>`

(no default, initially 0)

This key fits `<coordinates or nodes>` inside a node that is rotated by `<angle>`. As a side effect, it also sets the `/tikz/rotate` key.



```
\usetikzlibrary {fit}
\begin{tikzpicture}[inner sep=0pt, thick,
dot/.style={fill=blue,circle,minimum size=3pt}]
\draw[help lines] (0,0) grid (3,2);
\node[dot] (a) at (1,1) {};
\node[dot] (b) at (2,2) {};
\node[dot] (c) at (1,2) {};
\node[dot] (d) at (1.25,0.25) {};
\node[dot] (e) at (1.75,1.5) {};
\node[draw, fit=(a) (b) (c) (d) (e)] {};
\node[draw=red, rotate fit=30, fit=(a) (b) (c) (d) (e)] {};
\end{tikzpicture}
```

56 Fixed Point Arithmetic Library

TikZ Library `fixedpointarithmetic`

```
\usepgflibrary{fixedpointarithmetic} % LATEX and plain TEX and pure pgf
\usepgflibrary[fixedpointarithmetic] % ConTeXt and pure pgf
\usetikzlibrary{fixedpointarithmetic} % LATEX and plain TEX when using TikZ
\usetikzlibrary[fixedpointarithmetic] % ConTeXt when using TikZ
```

This library provides an interface to the L^AT_EX package `fp` for fixed point arithmetic. In addition to loading this library you must ensure `fp` is loaded, otherwise errors will occur.

56.1 Overview

Whilst the mathematical engine that comes with PGF is reasonably fast and flexible when it comes to parsing, the accuracy tends to be fairly low, particularly for expressions involving many operations chained together. In addition the range of values that can be computed is very small: ± 16383.99999 . Conversely, the `fp` package has a reasonably high accuracy, and can perform computations over a wide range of values (approximately $\pm 9.999 \times 10^{17}$), but is comparatively slow and not very flexible, particularly regarding parsing.

This library enables the combination of the two: the flexible parser of the PGF mathematical engine with the evaluation accuracy of `fp`. There are, however, a number of important points to bear in mind:

- Whilst `fp` supports very large numbers, PGF and TikZ do not. It is possible to calculate the result of 2^{20} or $1.2e10+3.4e10$, but it is not possible to use these results in pictures directly without some “extra work”.
- The PGF mathematical engine will still be used to evaluate lengths, such as `10pt` or `3em`, so it is not possible for a length to exceed the range of values supported by T_EX-dimensions ($\pm 16383.99999pt$), even though the resulting expression is within the range of `fp`. So, for example, one can calculate `3cm*10000`, but not `3*10000cm`.
- Not all of the functions listed in Section ??, have been mapped onto `fp` equivalents. Of those that have been, it is not guaranteed that functions will perform in the same way as they do in PGF. Reference should be made to the documentation for `fp`.
- In PGF, trigonometric functions such as `sin` and `cos` assume arguments are in degrees, and functions such as `asin` and `acos` return results in degrees. Although `fp` uses radians for such functions, PGF automatically converts arguments from degrees to radians, and converts results from radians to degrees, to ensure everything “works properly”.
- The overall speed will actually be slower than using PGF mathematical engine. The calculating power of `fp` comes at the cost of an increased processing time.

56.2 Using Fixed Point Arithmetic in PGF and TikZ

The following key is provided to use `fp` in PGF and TikZ:

`/pgf/fixed point arithmetic=<options>` (no default)
alias `/tikz/fixed point arithmetic`

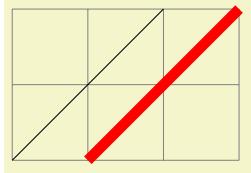
This key will set the key path to `/pgf/fixed point`, and execute `<options>`. Then it will install the necessary commands so that the PGF parser will use `fp` to perform calculations. The best way to use this key is as an argument to a scope or picture. This means that `fp` does not always have to be used, and PGF can use its own mathematical engine at other times, which can lead to a significant reduction in the time for a document to compile.

Currently there are only a few keys key supported for `<options>`:

`/pgf/fixed point/scale results=<factor>` (no default)

As noted above, `fp` can process a far greater range of numbers than PGF and TikZ. In order to use results from `fp` in a `{pgfpicture}` or a `{tikzpicture}` they need to be scaled. When this key is used PGF will scale results of any evaluation by `<factor>`. However, as it is not desirable for every part of every expression to be scaled, scaling will only take place if a special prefix `*` is used. If `*` is used at

the beginning of an expression the evaluation of the expression will be evaluated and then multiplied by $\langle factor \rangle$.



```
\usepgflibrary {fixedpointarithmetic}
\begin{tikzpicture}[fixed point arithmetic={scale results=10^-6}]
\draw [help lines] grid (3,2);
\draw (0,0) -- (2,2);
\draw [red, line width=4pt] (*1.0e6,0) -- (*3.0e6,*2.0e6);
\end{tikzpicture}
```

A special case of scaling involves plots of data containing large numbers from files. It is possible to “pre-process” a file, typically using the application that generates the data, to either precede the relevant column with $*$ or to perform the scaling as part of the calculation process. However, it may be desirable for the data in a plot to appear in a table as well, so, two files would be required, one pre-processed for plotting, and one not. This extra work may be undesirable so the following keys are provided:

/pgf/fixed point/scale file plot x= $\langle factor \rangle$ (no default)

This key will scale the first column of data read from a file before it is plotted. It is independent of the **scale results** key.

/pgf/fixed point/scale file plot y= $\langle factor \rangle$ (no default)

This key will scale the second column of data read from a file before it is plotted.

/pgf/fixed point/scale file plot z= $\langle factor \rangle$ (no default)

This key will scale the third column of data read from a file before it is plotted.

57 Floating Point Unit Library

by Christian Feuersänger

TikZ Library `fpu`

```
\usepgflibrary{fpu} % LATEX and plain TEX and pure pgf
\usepgflibrary[fpu] % ConTEX and pure pgf
\usetikzlibrary{fpu} % LATEX and plain TEX when using TikZ
\usetikzlibrary[fpu] % ConTEX when using TikZ
```

The floating point unit (fpu) allows the full data range of scientific computing for use in PGF. Its core is the PGF math routines for mantissa operations, leading to a reasonable trade-off between speed and accuracy. It does not require any third-party packages or external programs.

57.1 Overview

The fpu provides a replacement set of math commands which can be installed in isolated places to achieve large data ranges at reasonable accuracy. It provides at least¹⁰ the IEEE double precision data range, $-1 \cdot 10^{324}, \dots, 1 \cdot 10^{324}$. The absolute smallest number bigger than zero is $1 \cdot 10^{-324}$. The FPU's relative precision is at least $1 \cdot 10^{-4}$ although operations like addition have a relative precision of $1 \cdot 10^{-6}$.

Note that the library has not really been tested together with any drawing operations. It should be used to work with arbitrary input data which is then transformed somehow into PGF precision. This, in turn, can be processed by PGF.

57.2 Usage

`/pgf/fpu={⟨boolean⟩}` (default `true`)

This key installs or uninstalls the FPU. The installation exchanges any routines of the standard math parser with those of the FPU: `\pgfmathadd` will be replaced with `\pgfmathfloatadd` and so on. Furthermore, any number will be parsed with `\pgfmathfloatparsenumber`.

```
1Y2.0e0] \usepgflibrary {fpu}
\pgfkeys{/pgf/fpu}
\pgfmathparse{1+1}\pgfmathresult
```

The FPU uses a low-level number representation consisting of flags, mantissa and exponent¹¹. To avoid unnecessary format conversions, `\pgfmathresult` will usually contain such a cryptic number. Depending on the context, the result may need to be converted into something which is suitable for PGF processing (like coordinates) or may need to be typeset. The FPU provides such methods as well.

Use `fpu=false` to deactivate the FPU. This will restore any change. Please note that this is not necessary if the FPU is used inside of a T_EX group – it will be deactivated afterwards anyway.

It does not hurt to call `fpu=true` or `fpu=false` multiple times.

Please note that if the `fixedpointarithmetic` library of PGF will be activated after the FPU, the FPU will be deactivated automatically.

`/pgf/fpu/output format=⟨format⟩` (no default, initially `float`)

This key allows to change the number format in which the FPU assigns `\pgfmathresult`.

The predefined choice `float` uses the low-level format used by the FPU. This is useful for further processing inside of any library.

```
1Y2.17765411e23] \usepgflibrary {fpu}
\pgfkeys{/pgf/fpu,/pgf/fpu/output format=⟨format⟩}
\pgfmathparse{exp(50)*42}\pgfmathresult
```

The choice `sci` returns numbers in the format $⟨mantissa⟩e⟨exponent⟩$. It provides almost no computational overhead.

¹⁰To be more precise, the FPU's exponent is currently a 32-bit integer. That means it supports a significantly larger data range than an IEEE double precision number – but if a future T_EX version may provide low-level access to doubles, this may change.

¹¹Users should *always* use high level routines to manipulate floating point numbers as the format may change in a future release.

```
5.6154816e14 \usepgflibrary {fpu}
\pgfkeys{/pgf/fpu,/pgf/fpu/output format=sci}
\pgfmathparse{4.22e-8^2}\pgfmathresult
```

The choice `fixed` returns normal fixed point numbers and provides the highest compatibility with the PGF engine. It is activated automatically in case the FPU scales results.

```
0.000000999985 \usepgflibrary {fpu}
\pgfkeys{/pgf/fpu,/pgf/fpu/output format=fixed}
\pgfmathparse{\sqrt{1e-12}}\pgfmathresult
```

`/pgf/fpu/scale results={<scale>}` (no default)

A feature which allows semi-automatic result scaling. Setting this key has two effects: first, the output format for *any* computation will be set to `fixed` (assuming results will be processed by PGF's kernel). Second, any expression which starts with a star, `*`, will be multiplied with `<scale>`.

`/pgf/fpu/scale file plot x={<scale>}` (no default)
`/pgf/fpu/scale file plot y={<scale>}` (no default)
`/pgf/fpu/scale file plot z={<scale>}` (no default)

These keys will patch PGF's `plot file` command to automatically scale single coordinates by `<scale>`.

The initial setting does not scale `plot file`.

`\pgflibraryfpufactive{<true-code>}{<false-code>}`

This command can be used to execute either `<true-code>` or `<false-code>`, depending on whether the FPU has been activated or not.

57.3 Comparison to the fixed point arithmetics library

There are other ways to increase the data range and/or the precision of PGF's math parser. One of them is the `fp` package, preferable combined with PGF's `fixedpointarithmetic` library. The differences between the FPU and `fp` are:

- The FPU supports at least the complete IEEE double precision number range, while `fp` covers only numbers of magnitude $\pm 1 \cdot 10^{17}$.
- The FPU has a uniform relative precision of about 4–5 correct digits. The fixed point library has an absolute precision which may perform good in many cases – but will fail at the ends of the data range (as every fixed point routines does).
- The FPU has potential to be faster than `fp` as it has access to fast mantissa operations using PGF's math capabilities (which use T_EX registers).

57.4 Command Reference and Programmer's Manual

57.4.1 Creating and Converting Floats

`\pgfmathfloatparsenumber{<x>}`

Reads a number of arbitrary magnitude and precision and stores its result into `\pgfmathresult` as floating point number $m \cdot 10^e$ with mantissa and exponent base 10.

The algorithm and the storage format is purely text-based. The number is stored as a triple of flags, a positive mantissa and an exponent, such as

```
1Y2.0e0] \pgfmathfloatparsenumber{2}
\pgfmathresult
```

Please do not rely on the low-level representation here, use `\pgfmathfloattomacro` (and its variants) and `\pgfmathfloatcreate` if you want to work with these components.

The flags encoded in `\pgfmathresult` are represented as a digit where '0' stands for the number $\pm 0 \cdot 10^0$, '1' stands for a positive sign, '2' means a negative sign, '3' stands for 'not a number', '4' means $+\infty$ and '5' stands for $-\infty$.

The mantissa is a normalized real number $m \in \mathbb{R}$, $1 \leq m < 10$. It always contains a period and at least one digit after the period. The exponent is an integer.

Examples:

Flags: 0; Mantissa 0.0; Exponent 0.

```
\pgfmathfloatparsenumber{0}
\pgfmathfloattomacro{\pgfmathresult}{\F}{\M}{\E}
Flags: \F; Mantissa \M; Exponent \E.
```

Flags: 1; Mantissa 2.0; Exponent -1.

```
\pgfmathfloatparsenumber{0.2}
\pgfmathfloattomacro{\pgfmathresult}{\F}{\M}{\E}
Flags: \F; Mantissa \M; Exponent \E.
```

Flags: 1; Mantissa 4.2; Exponent 1.

```
\pgfmathfloatparsenumber{42}
\pgfmathfloattomacro{\pgfmathresult}{\F}{\M}{\E}
Flags: \F; Mantissa \M; Exponent \E.
```

Flags: 1; Mantissa 2.05; Exponent 3.

```
\pgfmathfloatparsenumber{20.5E+2}
\pgfmathfloattomacro{\pgfmathresult}{\F}{\M}{\E}
Flags: \F; Mantissa \M; Exponent \E.
```

Flags: 1; Mantissa 1.0; Exponent 6.

```
\pgfmathfloatparsenumber{1e6}
\pgfmathfloattomacro{\pgfmathresult}{\F}{\M}{\E}
Flags: \F; Mantissa \M; Exponent \E.
```

Flags: 1; Mantissa 5.21513; Exponent -11.

```
\pgfmathfloatparsenumber{5.21513e-11}
\pgfmathfloattomacro{\pgfmathresult}{\F}{\M}{\E}
Flags: \F; Mantissa \M; Exponent \E.
```

The argument $\langle x \rangle$ may be given in fixed point format or the scientific “e” (or “E”) notation. The scientific notation does not necessarily need to be normalized. The supported exponent range is (currently) only limited by the TeX-integer range (which uses 31 bit integer numbers).

/pgf/fpu/handlers/empty number={⟨input⟩}{⟨unreadable part⟩} (no default)

This command key is invoked in case an empty string is parsed inside of `\pgfmathfloatparsenumber`. You can overwrite it to assign a replacement `\pgfmathresult` (in `float!`).

The initial setting is to invoke `invalid number`, see below.

/pgf/fpu/handlers/invalid number={⟨input⟩}{⟨unreadable part⟩} (no default)

This command key is invoked in case an invalid string is parsed inside of `\pgfmathfloatparsenumber`. You can overwrite it to assign a replacement `\pgfmathresult` (in `float!`).

The initial setting is to generate an error message.

/pgf/fpu/handlers/wrong lowlevel format={⟨input⟩}{⟨unreadable part⟩} (no default)

This command key is invoked whenever `\pgfmathfloattoregisters` or its variants encounter something which is not a properly formatted low-level floating point number. As for `invalid number`, this key may assign a new `\pgfmathresult` (in floating point) which will be used instead of the offending `⟨input⟩`.

The initial setting is to generate an error message.

`\pgfmathfloatqparseumber{<x>}`

The same as `\pgfmathfloatparsenumber`, but does not perform sanity checking.

`\pgfmathfloattofixed{<x>}`

Converts a number in floating point representation to a fixed point number. It is a counterpart to `\pgfmathfloatparsenumber`. The algorithm is purely text based and defines `\pgfmathresult` as a string sequence which represents the floating point number $<x>$ as a fixed point number (of arbitrary precision).

Flags: 1; Mantissa 5.2; Exponent -4 → 0.00052

```
\pgfmathfloatparsenumber{0.00052}
\pgfmathfloattomacro{\pgfmathresult}{\F}{\M}{\E}
Flags: \F; Mantissa \M; Exponent \E
\$ \to $
\pgfmathfloattofixed{\pgfmathresult}
\pgfmathresult
```

Flags: 1; Mantissa 1.23456; Exponent 6 → 1234560.00000000

```
\pgfmathfloatparsenumber{123.456e4}
\pgfmathfloattomacro{\pgfmathresult}{\F}{\M}{\E}
Flags: \F; Mantissa \M; Exponent \E
\$ \to $
\pgfmathfloattofixed{\pgfmathresult}
\pgfmathresult
```

`\pgfmathfloattoint{<x>}`

Converts a number from low-level floating point representation to an integer (by truncating the fractional part).

123456	<code>\pgfmathfloatparsenumber{123456}</code>
	<code>\pgfmathfloattoint{\pgfmathresult}</code>
	<code>\pgfmathresult</code>

See also `\pgfmathfloatint` which returns the result as float.

`\pgfmathfloattosci{<float>}`

Converts a number from low-level floating point representation to scientific format, $1.234e4$. The result will be assigned to the macro `\pgfmathresult`.

`\pgfmathfloatvalueof{<float>}`

Expands a number from low-level floating point representation to scientific format, $1.234e4$.

Use `\pgfmathfloatvalueof` in contexts where only expandable macros are allowed.

`\pgfmathfloatcreate{<flags>}{<mantissa>}{<exponent>}`

Defines `\pgfmathresult` as the floating point number encoded by $<\text{flags}>$, $<\text{mantissa}>$ and $<\text{exponent}>$.

All arguments are characters and will be expanded using `\edef`.

Flags: 1; Mantissa 1.0; Exponent 327

```
\pgfmathfloatcreate{1}{1.0}{327}
\pgfmathfloattomacro{\pgfmathresult}{\F}{\M}{\E}
Flags: \F; Mantissa \M; Exponent \E
```

`\pgfmathfloatiff{<floating point number>}{<flag>}{<true-code>}{<false-code>}`

Invokes $<\text{true-code}>$ if the flag of $<\text{floating point number}>$ equals $<\text{flag}>$ and $<\text{false-code}>$ otherwise.

The argument $<\text{flag}>$ can be one of

0 to test for zero,

- 1** to test for positive numbers,
- +** to test for positive numbers,
- 2** to test for negative numbers,
- to test for negative numbers,
- 3** for “not-a-number”,
- 4** for $+\infty$,
- 5** for $-\infty$.

It's not zero!It's positive!It's not negative!It's positive!It's not negative!

```
\usetikzlibrary {fpu}
\pgfmathfloatparsenumber{42}
\pgfmathfloatifflags{\pgfmathresult}{0}{It's zero!}{It's not zero!}
\pgfmathfloatifflags{\pgfmathresult}{1}{It's positive!}{It's not positive!}
\pgfmathfloatifflags{\pgfmathresult}{2}{It's negative!}{It's not negative!}

% or, equivalently
\pgfmathfloatifflags{\pgfmathresult}{+}{It's positive!}{It's not positive!}
\pgfmathfloatifflags{\pgfmathresult}{-}{It's negative!}{It's not negative!}
```

\pgfmathfloattomacro{ x }{{ $\langle flagsmacro \rangle$ }{{ $\langle mantissamacro \rangle$ }{{ $\langle exponentmacro \rangle$ }}}

Extracts the flags of a floating point number x to $\langle flagsmacro \rangle$, the mantissa to $\langle mantissamacro \rangle$ and the exponent to $\langle exponentmacro \rangle$.

\pgfmathfloattoregisters{ x }{{ $\langle flagscount \rangle$ }{{ $\langle mantissadimen \rangle$ }{{ $\langle exponentcount \rangle$ }}}

Takes a floating point number x as input and writes flags to count register $\langle flagscount \rangle$, mantissa to dimen register $\langle mantissadimen \rangle$ and exponent to count register $\langle exponentcount \rangle$.

Please note that this method rounds the mantissa to TeX-precision.

\pgfmathfloattoregisterstok{ x }{{ $\langle flagscount \rangle$ }{{ $\langle mantissatoks \rangle$ }{{ $\langle exponentcount \rangle$ }}}

A variant of **\pgfmathfloattoregisters** which writes the mantissa into a token register. It maintains the full input precision.

\pgfmathfloatgetflags{ x }{{ $\langle flagscount \rangle$ }}

Extracts the flags of x into the count register $\langle flagscount \rangle$.

\pgfmathfloatgetflagstomacro{ x }{{ $\langle macro \rangle$ }}

Extracts the flags of x into the macro $\langle macro \rangle$.

\pgfmathfloatgetmantissa{ x }{{ $\langle mantissadimen \rangle$ }}

Extracts the mantissa of x into the dimen register $\langle mantissadimen \rangle$.

\pgfmathfloatgetmantissatok{ x }{{ $\langle mantissatoks \rangle$ }}

Extracts the mantissa of x into the token register $\langle mantissatoks \rangle$.

\pgfmathfloatgetexponent{ x }{{ $\langle exponentcount \rangle$ }}

Extracts the exponent of x into the count register $\langle exponentcount \rangle$.

57.4.2 Symbolic Rounding Operations

Commands in this section constitute the basic level implementations of the rounding routines. They work symbolically, i.e. they operate on text, not on numbers and allow arbitrarily large numbers.

\pgfmathroundto{ x }

Rounds a fixed point number to prescribed precision and writes the result to **\pgfmathresult**.

The desired precision can be configured with **/pgf/number format/precision**, see section ???. This section does also contain application examples.

Any trailing zeros after the period are discarded. The algorithm is purely text based and allows to deal with precisions beyond TeX's fixed point support.

As a side effect, the global boolean `\ifpgfmathfloatroundhasperiod` will be set to true if and only if the resulting mantissa has a period. Furthermore, `\ifpgfmathfloatroundmayneedrenormalize` will be set to true if and only if the rounding result's floating point representation would have a larger exponent than $\langle x \rangle$.

```
1   \pgfmathroundto{1}
      \pgfmathresult
```

```
4.69 \pgfmathroundto{4.685}
      \pgfmathresult
```

```
20000 \pgfmathroundto{19999.9996}
      \pgfmathresult
```

`\pgfmathroundtozerooffill{\langle x \rangle}`

A variant of `\pgfmathroundto` which always uses a fixed number of digits behind the period. It fills missing digits with zeros.

```
1.00 \pgfmathroundtozerooffill{1}
      \pgfmathresult
```

```
4.69 \pgfmathroundto{4.685}
      \pgfmathresult
```

```
20000.00 \pgfmathroundtozerooffill{19999.9996}
      \pgfmathresult
```

`\pgfmathfloatround{\langle x \rangle}`

Rounds a normalized floating point number to a prescribed precision and writes the result to `\pgfmathresult`.

The desired precision can be configured with `/pgf/number format/precision`, see section ??.

This method employs `\pgfmathroundto` to round the mantissa and applies renormalization if necessary.

As a side effect, the global boolean `\ifpgfmathfloatroundhasperiod` will be set to true if and only if the resulting mantissa has a period.

```
5.26e1 \pgfmathfloatparsenumber{52.5864}
      \pgfmathfloatround{\pgfmathresult}
      \pgfmathfloattosci{\pgfmathresult}
      \pgfmathresult
```

```
1e1 \pgfmathfloatparsenumber{9.995}
      \pgfmathfloatround{\pgfmathresult}
      \pgfmathfloattosci{\pgfmathresult}
      \pgfmathresult
```

`\pgfmathfloatroundzerooffill{\langle x \rangle}`

A variant of `\pgfmathfloatround` produces always the same number of digits after the period (it includes zeros if necessary).

```
5.26e1 \pgfmathfloatparsenumber{52.5864}
      \pgfmathfloatroundzerooffill{\pgfmathresult}
      \pgfmathfloattosci{\pgfmathresult}
      \pgfmathresult
```

```
1.00e1 \pgfmathfloatparsenumber{9.995}
      \pgfmathfloatroundzerooffill{\pgfmathresult}
      \pgfmathfloattosci{\pgfmathresult}
      \pgfmathresult
```

57.4.3 Math Operations Commands

This section describes some of the replacement commands in more detail.

Please note that these commands can be used even if the `fpu` as such has not been activated – it is sufficient to load the library.

`\pgfmathfloat{op}`

Methods of this form constitute the replacement operations where `<op>` can be any of the well-known math operations.

Thus, `\pgfmathfloatadd` is the counterpart for `\pgfmathadd` and so on. The semantics and number of arguments is the same, but all input and output arguments are *expected* to be floating point numbers.

`\pgfmathfloattoextendedprecision{<x>}`

Renormalizes `<x>` to extended precision mantissa, meaning $100 \leq m < 1000$ instead of $1 \leq m < 10$.

The “extended precision” means we have higher accuracy when we apply pgfmath operations to mantissas.

The input argument is expected to be a normalized floating point number; the output argument is a non-normalized floating point number (well, normalized to extended precision).

The operation is supposed to be very fast.

`\pgfmathfloatsettextprecision{<shift>}`

Sets the precision used inside of `\pgfmathfloattoextendedprecision` to `<shift>`.

The different choices are

0	normalization to	0	$\leq m < 1$	(disable extended precision)
1	normalization to	10	$\leq m < 100$	
2	normalization to	100	$\leq m < 1000$	(default of <code>\pgfmathfloattoextendedprecision</code>)
3	normalization to	1000	$\leq m < 10000$	

`\pgfmathfloatlessthan{<x>}{<y>}`

Defines `\pgfmathresult` as 1.0 if `<x> < <y>`, but 0.0 otherwise. It also sets the global TeX-boolean `\pgfmathfloatcomparison` accordingly. The arguments `<x>` and `<y>` are expected to be numbers which have already been processed by `\pgfmathfloatparsenumber`. Arithmetic is carried out using TeX-registers for exponent- and mantissa comparison.

`\pgfmathfloatmultiplyfixed{<float>}{<fixed>}`

Defines `\pgfmathresult` to be `<float> · <fixed>` where `<float>` is a floating point number and `<fixed>` is a fixed point number. The computation is performed in floating point arithmetics, that means we compute $m \cdot <fixed>$ and renormalize the result where m is the mantissa of `<float>`.

This operation renormalizes `<float>` with `\pgfmathfloattoextendedprecision` before the operation, that means it is intended for relatively small arguments of `<fixed>`. The result is a floating point number.

`\pgfmathfloatifapproxequalrel{<a>}{}{<true-code>}{<false-code>}`

Computes the relative error between `<a>` and `` (assuming ` ≠ 0`) and invokes `<true-code>` if the relative error is below `/pgf/fpu/rel thresh` and `<false-code>` if that is not the case.

The input arguments will be parsed with `\pgfmathfloatparsenumber`.

`/pgf/fpu/rel thresh={<number>}` (no default, initially `1e-4`)

A threshold used by `\pgfmathfloatifapproxequalrel` to decide whether numbers are approximately equal.

`\pgfmathfloatshift{<x>}{<num>}`

Defines `\pgfmathresult` to be `<x> · 10<num>`. The operation is an arithmetic shift base ten and modifies only the exponent of `<x>`. The argument `<num>` is expected to be a (positive or negative) integer.

`\pgfmathfloatabserror{<x>}{<y>}`

Defines `\pgfmathresult` to be the absolute error between two floating point numbers x and y , $|x - y|$ and returns the result as floating point number.

`\pgfmathfloatreerror{ x }{ y }`

Defines `\pgfmathresult` to be the relative error between two floating point numbers x and y , $|x - y|/|y|$ and returns the result as floating point number.

`\pgfmathfloatint{ x }`

Returns the integer part of the floating point number $\langle x \rangle$, by truncating any digits after the period. This methods truncates the absolute value $|x|$ to the next smaller integer and restores the original sign afterwards.

The result is returned as floating point number as well.

See also `\pgfmathfloattoint` which returns the number in integer format.

`\pgfmathlog{ x }`

Defines `\pgfmathresult` to be the natural logarithm of $\langle x \rangle$, $\ln(\langle x \rangle)$. This method is logically the same as `\pgfmathln`, but it applies floating point arithmetics to read number $\langle x \rangle$ and employs the logarithm identity

$$\ln(m \cdot 10^e) = \ln(m) + e \cdot \ln(10)$$

to get the result. The factor $\ln(10)$ is a constant, so only $\ln(m)$ with $1 \leq m < 10$ needs to be computed. This is done using standard pgf math operations.

Please note that $\langle x \rangle$ needs to be a number, expression parsing is not possible here.

If $\langle x \rangle$ is *not* a bounded positive real number (for example $\langle x \rangle \leq 0$), `\pgfmathresult` will be *empty*, no error message will be generated.

```
-15.7452
\usetikzlibrary {fpu}
\pgfmathlog{1.452e-7}
\pgfmathresult
```

```
20.28096
\usetikzlibrary {fpu}
\pgfmathlog{6.426e+8}
\pgfmathresult
```

57.4.4 Accessing the Original Math Routines for Programmers

As soon as the library is loaded, every private math routine will be copied to a new name. This allows library and package authors to access the TeX-register based math routines even if the FPU is activated. And, of course, it allows the FPU as such to perform its own mantissa computations.

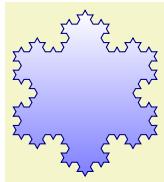
The private implementations of PGF math commands, which are of the form `\pgfmath<name>`, will be available as `\pgfmath@basic@<name>` as soon as the library is loaded.

58 Lindenmayer System Drawing Library

58.1 Overview

Lindenmayer systems (also commonly known as “L-systems”), were originally developed by Aristid Lindenmayer as a theory of algae growth patterns and then subsequently used to model branching patterns in plants and produce fractal patterns. Typically, an L-system consists of a set of symbols, each of which is associated with some graphical action (such as “turn left” or “move forward”) and a set of rules (“production” or “rewrite” rules). Given a string of symbols, the rewrite rules are applied several times and the resulting string is processed the action associated with each symbol is executed.

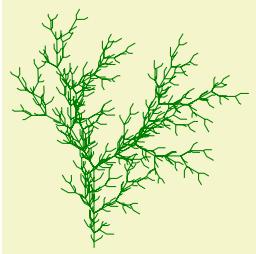
In PGF, L-systems can be used to create simple 2-dimensional fractal patterns...



```
\usetikzlibrary {lindenmayersystems}
\begin{tikzpicture}
\pgfdeclarelindenmayersystem{Koch curve}{
  \rule{F -> F-F++F-F}
}

\shadedraw [top color=white, bottom color=blue!50, draw=blue!50!black]
  [l-system={Koch curve, step=2pt, angle=60, axiom=F++F+F, order=3}]
  lindenmayer system -- cycle;
\end{tikzpicture}
```

...and “plant like” patterns...



```
\usetikzlibrary {lindenmayersystems}
\begin{tikzpicture}
\draw [green!50!black, rotate=90]
  [l-system={rule set={F -> FF-[-F+F]+[+F-F]}, axiom=F, order=4, step=2pt,
    randomize step percent=25, angle=30, randomize angle percent=5}]
  lindenmayer system;
\end{tikzpicture}
```

...but it is important to bear in mind that even moderately complex L-systems can exceed the available memory of TeX, and can be very slow. If possible, you are advised to increase the main memory and save stack to their maximum possible values for your particular TeX distribution. However, even by doing this you may find you still run out of memory quite quickly.

For an excellent introduction to L-systems (containing some “really cool” pictures – many of which are sadly not possible in PGF) see *The Algorithmic Beauty of Plants* by Przemyslaw Prusinkiewicz and Aristid Lindenmayer (which is freely available via the internet).

TikZ Library `lindenmayersystems`

```
\usepgflibrary{lindenmayersystems} % LEX and plain TeX and pure pgf
\usepgflibrary[lindenmayersystems] % ConTeXt and pure pgf
\usetikzlibrary{lindenmayersystems} % LEX and plain TeX when using TikZ
\usetikzlibrary[lindenmayersystems] % ConTeXt when using TikZ
```

This PGF-library provides basic commands for defining and using simple L-systems. The TikZ-library provides, furthermore, a front end for using L-systems in TikZ.

58.1.1 Declaring L-systems

Before an L-system can be used, it must be declared using the following command:

```
\pgfdeclarelindenmayersystem{\langle name\rangle}{\langle specification\rangle}
```

This command declares a Lindenmayer system called `\langle name\rangle`. The `\langle specification\rangle` argument contains a description of the L-system’s symbols and rules. Two commands `\symbol` and `\rule` are only defined when the `\langle specification\rangle` argument is executed.

```
\symbol{\langle name\rangle}{\langle code\rangle}
```

This defines a symbol called `\langle name\rangle` for a specific L-system, and associates it with `\langle code\rangle`.

A symbol should consist of a single alpha-numeric character (i.e., A-Z, a-z or 0-9). The symbols F, f, +, -, [and] are available by default so do not need to be defined for each L-system. However, if you are feeling adventurous, they can be redefined for specific L-systems if required. The L-system treats the default symbols as follows (the commands they execute are described below):

- F move forward a certain distance, drawing a line. Uses \pgflsystemdrawforward.
- f move forward a certain distance, without drawing a line. Uses \pgflsystemmoveforward.
- + turn left by some angle. Uses \pgflsystemturnleft.
- - turn right by some angle. Uses \pgflsystemturnright.
- [save the current state (i.e., the position and direction). Uses \pgflsystemsavestate.
-] restore the last saved state. Uses \pgflsystemrestorestate.

The symbols [and] act like a stack: [pushes the state of the L-system on to the stack, and] pops a state off the stack.

When *<code>* is executed, the transformation matrix is set up so that the origin is at the current position and the positive x-axis “points forward”, so \pgfpathlineto{\pgfpoint{1cm}{0cm}} draws a line 1cm forward.

The following keys can alter the production of an L-system. However, they do not store values in themselves.

/pgf/lindenmayer system/step=⟨length⟩ (no default, initially 5pt)

How far the L-system moves forward if required. This key sets the TeX dimension \pgflsystemstep.

/pgf/lindenmayer system/randomize step percent=⟨percentage⟩ (no default, initially 0)

If the step is to be randomized, this key specifies by how much. The value is stored in the TeX macro \pgflsystemrandomizesteppercent.

/pgf/lindenmayer system/left angle=⟨angle⟩ (no default, initially 90)

This key sets the angle through which the L-system turns when it turns left. The value is stored in the TeX macro \pgflsystemleftangle.

/pgf/lindenmayer system/right angle=⟨angle⟩ (no default, initially 90)

This key sets the angle through which the L-system turns when it turns right. The value is stored in the TeX macro \pgflsystemrightangle.

/pgf/lindenmayer system/randomize angle percent=⟨percentage⟩ (no default, initially 0)

If the angles are to be randomized, this key specifies by how much. The value is stored in the TeX macro \pgflsystemrandomizeanglepercent.

For speed and convenience, when the code for a symbol is executed, the following commands are available.

\pgflsystemcurrentstep

The current “step” of the L-system (i.e., how far the system will move forward if required). This is initially set to the value in the TeX-dimensions \pgflsystemstep, but the actual value may be changed if \pgflsystemrandomizestep is used (see below).

\pgflsystemcurrentleftangle

The angle the L-system will turn when it turns left. The value stored in this macro may be changed if \pgflsystemrandomizeleftangle is used.

\pgflsystemcurrentrightangle

The angle the L-system will turn when it turns right. The value stored in this macro may be changed if \pgflsystemrandomizerightangle is used.

The following commands may be useful if you wish to define your own symbols.

\pgflsystemrandomizestep

Randomizes the value in \pgflsystemcurrentstep according to the current value of the key randomize step percent.

`\pgf{system}{randomizeleftangle}`

Randomizes the value in `\pgf{system}{currentleftangle}` according to the value of the `randomize angle` percent key.

`\pgf{system}{randomizerightangle}`

Randomizes the value in `\pgf{system}{currentrightangle}` according to the value of the `randomize angle` key.

`\pgf{system}{drawforward}`

Move forward in the current direction, by `\pgf{system}{currentstep}`, drawing a line in the process. This macro calls `\pgf{system}{randomizestep}`. Internally, PGF simply shifts the transformation matrix in the positive direction of the current (transformed) x-axis by `\pgf{system}{step}` and then executes a line-to to the (newly transformed) origin.

`\pgf{system}{moveforward}`

Move forward in the current direction, by `\pgf{system}{currentstep}`, without drawing a line. This macro calls `\pgf{system}{randomizestep}`. PGF executes a transformation as above, but executes a move-to to the (newly transformed) origin.

`\pgf{system}{turnleft}`

Turn left by `\pgf{system}{currentleftangle}`. Internally, PGF simply rotates the transformation matrix. This macro calls `\pgf{system}{randomizeleftangle}`.

`\pgf{system}{turnright}`

Turn right by `\pgf{system}{currentrightangle}`. Internally, PGF simply rotates the transformation matrix. This macro calls `\pgf{system}{randomizerightangle}`.

`\pgf{system}{savestate}`

Save the current position and orientation. Internally, PGF simply starts a new T_EX-group.

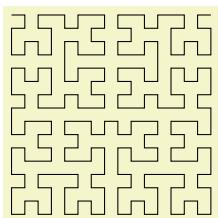
`\pgf{system}{restorestate}`

Restore the last saved position and orientation. Internally, PGF closes a T_EX-group, restoring the transformation matrix of the outer scope, and a move-to command is executed to the (transformed) origin.

`\rule{\langle head \rangle -> \langle body \rangle}`

Declare a rule. `\langle head \rangle` should consist of a single symbol, which need not have been declared using `\symbol` or exist as a default symbol (in fact, the more interesting L-systems depend on using symbols with no corresponding code, to control the “growth” of the system). `\langle body \rangle` consists of a string of symbols, which again need not necessarily have any code associated with them.

As an example, the following shows an L-system that uses some of these commands. This example illustrates the point that some symbols, in this case A and B, do not have to have code associated with them. They simply control the growth of the system.



```

\usetikzlibrary {lindenmayersystems}
\pgfdeclarelindenmayersystem{Hilbert curve}{
  \symbol{X}{\pgf{system}{drawforward}}
  \symbol{+}{\pgf{system}{turnright}} % Explicitly define + and - symbols.
  \symbol{-}{\pgf{system}{turnleft}}
  \rule{A -> +BX-AXA-XB+}
  \rule{B -> -AX+BXB+XA-}
}
\tikz\draw[lindenmayer system={Hilbert curve, axiom=A, order=4, angle=90}]
  lindenmayer system;

```

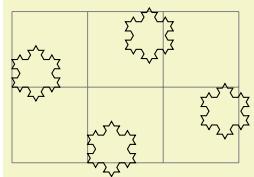
58.2 Using Lindenmayer Systems

58.2.1 Using L-Systems in PGF

The following command is used to run an L-system in PGF:

```
\pgflindenmayersystem{\<name>}{\<axiom>}{\<order>}
```

Runs the L-system called $\langle name \rangle$ using the input string $\langle axiom \rangle$ for $\langle order \rangle$ iterations. In general, prior to calling this command, the transformation matrix should be set appropriately for shifting and rotating, and a move-to to the (transformed) origin should be executed. This origin will be where the L-system starts. In addition, the relevant keys should be set appropriately.



```
\usetikzlibrary {lindenmayersystems}
\begin{tikzpicture}
  \draw [help lines] grid (3,2);
  \pgfset{lindenmayer system/.cd, angle=60, step=2pt}
  \foreach \x/\y in {0cm/1cm, 1.5cm/1.5cm, 2.5cm/0.5cm, 1cm/0cm}{
    \pgftransformshift{\pgfqpoint{\x}{\y}}
    \pgfpathmoveto{\pgfpointorigin}
    \pgflindenmayersystem{Koch curve}{F++F++F}{2}
    \pgfusepath{stroke}
  }
\end{tikzpicture}
```

Note that it is perfectly feasible for an L-system to define special symbols which perform the move-to and use-path operations.

58.2.2 Using L-Systems in TikZ

In TikZ, an L-system is created using a path operation. However, TikZ is more flexible regarding the positioning of the L-system and also provides keys to create L-systems “on-line”.

```
\path ... lindenmayer system [\<keys>] ...;
```

This will run an L-system according to the parameters specified in $\langle keys \rangle$ (which can also contain normal keys such as `draw` or `thin`). The syntax is flexible regarding the L-system parameters and the following all do the same thing:

```
\draw lindenmayer system [lindenmayer system={Hilbert curve, axiom=4, order=3}];
```

```
\draw [lindenmayer system={Hilbert curve, axiom=4, order=3}] lindenmayer system;
```

```
\tikzset{lindenmayer system={Hilbert curve, axiom=4, order=3}}
\draw lindenmayer system;
```

```
\path ... l-system [\<keys>] ...;
```

A more compact version of the `lindenmayer system` path command.

This library adds some additional keys for specifying L-systems. These keys only work in TikZ and all have the same path, namely, `/pgf/lindenmayer system`, but the following keys are provided for convenience, so that you do not have to keep repeating this path:

`/pgf/lindenmayer system={\<keys>}` (style, no default)
alias `/tikz/lindenmayer system`

This key changes the key path to `/pgf/lindenmayer systems` and executes $\langle keys \rangle$.

`/pgf/l-system={\<keys>}` (style, no default)
alias `/tikz/l-system`

A more compact version of the previous key.

`/pgf/lindenmayer system/name={\<name>}` (no default)

Sets the name for the L-system.

`/pgf/lindenmayer system/axiom={\<string>}` (no default)

Sets the axiom (or input string) for the L-system.

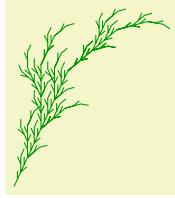
`/pgf/lindenmayer system/order={\<integer>}` (no default)

Sets the number of iterations the L-system will perform.

/pgf/lindenmayer system/rule set={⟨list⟩}

(no default)

This key allows an (anonymous) L-system to be declared “on-line”. There is, however, a restriction that only the default symbols can be used for drawing (empty symbols can still be used to control the growth of the system). The rules in ⟨list⟩ should be separated by commas.



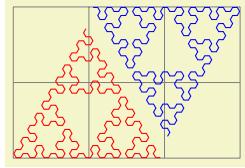
```
\usetikzlibrary {lindenmayersystems}
\tikz [rotate=65] \draw [green!60!black] l-system
```

```
[l-system={rule set={F -> F[+F]F[-F]}, axiom=F, order=4, angle=25, step=3pt}];
```

/pgf/lindenmayer system/anchor=⟨anchor⟩

(no default)

By default, when this key is not used, the L-system will start from the last specified coordinate. By using this key, the L-system will be placed inside a special (rectangle) node which can be positioned using ⟨anchor⟩.



```
\usetikzlibrary {lindenmayersystems}
\begin{tikzpicture}[l-system={step=1.75pt, order=5, angle=60}]
\pgfdeclarelindenmayersystem{Sierpinski triangle}{
  \symbol{X}{\pgflsystemdrawforward}
  \symbol{Y}{\pgflsystemdrawforward}
  \rule{X -> Y-X-Y}
  \rule{Y -> X+Y+X}
}
\draw [help lines] grid (3,2);
\draw [red] (0,0) l-system
  [l-system={Sierpinski triangle, axiom=+++X, anchor=south west}];
\draw [blue] (3,2) l-system
  [l-system={Sierpinski triangle, axiom=X, anchor=north east}];
\end{tikzpicture}
```

59 Math Library

TikZ Library `math`

```
\usetikzlibrary{math} % LATEX and plain TEX
\usetikzlibrary[math] % ConTEX
```

This library defines a simple mathematical language to define simple functions and perform sequences of basic mathematical operations.

59.1 Overview

PGF and TikZ both use the PGF mathematical engine which provides many commands for parsing expressions. Unfortunately the PGF math engine is somewhat cumbersome for long sequences of mathematical operations, particularly when assigning values to multiple variables. The TikZ `calc` library provides some additional “convenience” operations for doing calculations (particularly with coordinates), but this can only be used inside TikZ path commands.

This `math` library provides a means to perform sequences of mathematical operations in a more ‘user friendly’ manner than the PGF math engine. In addition, the coordinate calculations of the `calc` library can be accessed (provided it is loaded). However as the `math` library uses the PGF math engine – which uses pure T^EX to perform all its calculations – it is subject to the same speed and accuracy limitations. It is worth bearing this in mind, before trying to implement algorithms requiring intensive and highly accurate computation. You can, of course use the `fp` or the `fpu` libraries to increase the accuracy (but not necessarily the speed) of computations.

For most purposes, the features provided by this library are accessed using the following command:

```
\tikzmath{\langle statements \rangle}
```

This command process a series of $\langle \text{statements} \rangle$ which can represent assignments, function definitions, conditional evaluation, and iterations. It provides, in effect, a miniature mathematical language to perform basic mathematical operations. Perhaps the most important thing to remember is that *every statement should end with a semi-colon*. This is likely to be the most common reason why the `\tikzmath` command fails.

```
0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610, 987, 1597, 2584, 4181, 6765,
```

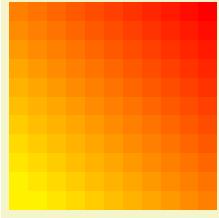
```
\usetikzlibrary {math}
\tikzmath{
    % Adapted from http://www.cs.northwestern.edu/academics/courses/110/html/fib_rec.html
    function fibonacci(\n) {
        if \n == 0 then {
            return 0;
        } else {
            return fibonacci2(\n, 0, 1);
        };
    };
    function fibonacci2(\n, \p, \q) {
        if \n == 1 then {
            return \q;
        } else {
            return fibonacci2(\n-1, \q, \p+\q);
        };
    };
    int \f, \i;
    for \i in {0,1,...,20}{
        \f = fibonacci(\i);
        print {\f, };
    };
}
```

In addition to this command the following key is provided:

```
/tikz/evaluate=\langle statements \rangle
```

(no default)

This key simply executes `\tikzmath{\langle statements \rangle}`.



```
\usetikzlibrary {math}
\begin{tikzpicture}[x=0.25cm,y=0.25cm,
    evaluate={
        int \i, \j;
        for \i in {0,...,10}{
            for \j in {0,...,10}{
                \a{\i,\j} = (\i+\j)*5;
            };
        };
    }
\foreach \i in {0,...,10}
\foreach \j in {0,...,10}
\fill [red!\a{\i,\j}!yellow] (\i,\j) rectangle ++(1, 1);

```

The following sections describe the miniature language that this library provides and can be used in the `\tikzmath` command and the `evaluate` key. The language consists only of simple keywords and expressions but the mini-parser allows you to format code in a reasonably versatile way (much like the `tikz` parser) except that *all the keywords must be followed by at least one space*. This is the second most important thing to remember (after remembering to insert semi-colons at the end of every statement).

59.2 Assignment

In the simplest case, you will want to evaluate an expression and assign it to a macro, or a `TeX` count or dimension register. In this case, use of the `math` library is straightforward:

26.0, 2.0, 11, 225.0pt

```
\usetikzlibrary {math}
\newcount\mycount
\newdimen\mydimen
\tikzmath{
    \a = 4*5+6;
    \b = sin(30)*4;
    \mycount = log10(2048) / log10(2);
    \mydimen = 15^2;
}
\a, \b, \the\mycount, \the\mydimen
```

In addition, `TeX`-macros (*not* `TeX` registers) can be suffixed with an index, similar to indices in mathematical notation, for example, x_1 , x_2 , x_3 :

7.0, 70.0, 700.0

```
\usetikzlibrary {math}
\tikzmath{
    \x1 = 3+4; \x2 = 30+40; \x3 = 300+400;
}
\x1, \x2, \x3
```

The index does not have to be a number. By using braces {}, more sophisticated indices can be created:

The speed of sound in air is 340 m/s. The speed of sound in steel is 6100 m/s.

```
\usetikzlibrary {math}
\tikzmath{
    \c{air} = 340; \c{water} = 1435; \c{steel} = 6100;
}
\foreach \medium in {air,steel}{The speed of sound in \medium\ is \c{\medium} m/s. }
```

You should not, however, try to mix indexed and non-indexed variables. Once an assignment is made using an index, the `math` library expects all instances of the variable on the right hand side of an assignment to be followed by an index. This effect is reversed if you subsequently make an assignment to the variable without an index: the `math` library (or to be precise the PGF math-engine) will then ignore any index following the variable on the right hand side of an assignment.

In some cases, you may wish to assign a value or expression to a variable without evaluating it with the PGF math-engine. In this case, you can use the following keyword:

`let <variable> = <expression>;`

This keyword assigns $\langle expression \rangle$ to $\langle variable \rangle$ without evaluation. The $\langle expression \rangle$ is however fully expanded using `\edef`. Any spaces preceding $\langle expression \rangle$ are removed, but any trailing spaces (before the semi-colon) are included.

```
(5*4)+1, "blue"
\usetikzlibrary {math}
\tikzmath{
  let \x = (5*4)+1;
  let \c1 = blue;
}
\x, "\c1"
```

59.3 Integers, “Real” Numbers, and Coordinates

By default, assignments are made by evaluating expressions using the PGF math-engine and results are usually returned as number with a decimal point (unless you are assigning to a count register or use the `int` function). As this is not always desirable, the `math` library allows variables – which *must* be TeX macros – to be ‘declared’ as being a particular ‘type’. The library recognizes three types: integers (numbers without a decimal point), real numbers (numbers with a decimal point¹²), and coordinates.

To declare a variable as being one of the three types, you can use the keywords shown below. It is important to remember that by telling the `math` library you want it to do a particular assignment for a variable, it will also do the same assignment when the variable is indexed.

```
7, 70, 700
\usetikzlibrary {math}
\tikzmath{
  integer \x;
  \x1 = 3+4; \x2 = 30+40; \x3 = 300+400;
}
\x1, \x2, \x3
```

integer $\langle variable \rangle, \langle additional\ variables \rangle;$

The `integer` keyword indicates that assignments to the $\langle variable \rangle$ or the comma separated list of $\langle additional\ variables \rangle$ should be truncated (not rounded) to integers. The variables should be ordinary macros – *not* TeX registers. In addition the variables should *not* be indexed.

```
x = 26, y = 2, z = 9
\usetikzlibrary {math}
\tikzmath{
  integer \x, \y, \z;
  \x = 4*5+6;
  \y = sin(30)*4;
  \z = log10(512) / log10(2);
  print {$x=\x$, $y=\y$, $z=\z$};
}
```

int $\langle variable \rangle, \langle additional\ variables \rangle;$

Short version of the `integer` keyword.

Having declared a variable as an integer, the `math` library will continue to assign only integers to that variable within the current TeX scope. If you wish to assign non-integer (i.e., *real*) numbers to the same variable, the following keyword can be used.

real $\langle variable \rangle, \langle additional\ variables \rangle;$

The `real` keyword ensures that assignments $\langle variable \rangle$ (and $\langle additional\ variables \rangle$) will not be truncated to integers.

In order to take advantage of `math` library interface to the `calc` library you must indicate that a variable is to be assigned coordinates, using the following keyword.

coordinate $\langle variable \rangle, \langle additional\ variables \rangle;$

This keyword enables TikZ-style coordinates such as `(2cm,3pt)` or `(my node.east)` to be parsed and assigned to $\langle variable \rangle$ in the form x, y , which can then be used in a `tikzpicture`:

¹²Strictly speaking, due to the finite range and precision of TeX numerical capabilities, the term “real” is not correct.

```

    / \usetikzlibrary {math}
      \tikzmath{
        coordinate \c;
        \c = (45:10pt);
    }
    \tikz\draw (0,0) -- (\c);

```

If the TikZ `calc` library is loaded, coordinate calculations can be performed; the coordinate expression does not have to be surrounded by `($...$)`.

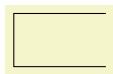


```

    / \usetikzlibrary {math}
      \tikzmath{
        coordinate \c, \d;
        \c = (-1,2)+(1,-1);
        \d = (4,1)-(2,-1);
    }
    \tikz\draw (\c) -- (\d);

```

In addition to assigning the x and y coordinates to $\langle variable \rangle$ (possibly with an optional index), two further variables are defined. The first takes the name of $\langle variable \rangle$ (e.g., `\c`) suffixed with `x` (i.e., `\cx`) and is assigned the x coordinate of `\c`. The second takes the name of $\langle variable \rangle$ suffixed with `y` (i.e., `\cy`) and is assigned the y coordinate of `\c`.



```

    / \usetikzlibrary {math}
      \tikzmath{
        coordinate \c;
        \c1 = (30:20pt);
        \c2 = (210:20pt);
    }
    \tikz\draw (\cx1,\cy1) -- (\cx2,\cy1) -- (\cx2,\cy2) -- (\cx1,\cy2);

```

59.4 Repeating Things

```
for <variable> in {<list>}{<expressions>};
```

This is a “trimmed down” version of the `\foreach` command available as part of PGF and TikZ, but cannot currently be used outside of the `\tikzmath` command. It is important to note the following:

- Every value in $\langle list \rangle$ is evaluated using the PGF mathematical engine. However, if an item in $\langle list \rangle$ contains a comma, it *must* be surrounded by braces, for example, `{mod(5, 2)}`.

```

x = 8, v = 256
    / \usetikzlibrary {math}
      \tikzmath{
        int \x, \v;
        \v=1;
        for \x in {1,...,{random(3,10)}}{
          \v=\v*2;
        };
        print {$x=\x, v=\v$};
    }

```

- Because each item is evaluated, you cannot use TikZ coordinates in $\langle list \rangle$.
- Only single variable assignment is supported.
- The “dots notation” (e.g., $1, 2, \dots, 9$) can be used in $\langle list \rangle$, but is not as sophisticated as the PGF `\foreach` command. In particular, contextual replacement is not possible.
- Assignments that occur in the loop body *are not scoped*. They last beyond the body of each iteration and the end of the `for` statement. This includes the values assigned to the $\langle variable \rangle$.

```
x1 = 5, x2 = 50, y = 2250
```

```

    / \usetikzlibrary {math}
      \tikzmath{
        int \x, \y;
        \y = 0;
        for \x1 in {1,...,5}{
          for \x2 in {10,20,...,50}{
            \y = \y+\x1*\x2;
          };
        };
        $x\_1=\x1, x\_2=\x2, y=\y$ 
    }

```

59.5 Branching Statements

Sometimes you may wish to execute different statements depending on the value of an expression. In this case the following keyword can be used:

```
if <condition> then {<if-non-zero-statements>};
```

This keyword executes `<if-non-zero-statements>` if the expression in `<condition>` evaluates to any value other than zero.

```
if <condition> then {<if-non-zero-statements>} else {<if-zero-statements>};
```

This keyword executes `<if-non-zero-statements>` if the expression in `<condition>` evaluates to any value other than zero and the `<if-zero-statements>` are executed if the expression in `<condition>` evaluates to zero.



```
\usetikzlibrary {math}
\begin{tikzpicture}
\tikzmath{
    int \x;
    for \k in {0,10,...,350}{
        if \k>260 then { let \c = orange; } else {
            if \k>170 then { let \c = blue; } else {
                if \k>80 then { let \c = red; } else {
                    let \c = green; }; }; };
        {
            \path [fill=\c!50, draw=\c] (\k:0.5cm) -- (\k:1cm) --
                (\k+5:1cm) -- (\k+5:0.5cm) -- cycle;
        };
    };
}
\end{tikzpicture}
```

59.6 Declaring Functions

You can add functions by using the following keywords:

```
function <name>(<arguments>) {<definition>};
```

This keyword works much like the `declare function` provided by the PGF math-engine. The function `<name>` can be any name that is not already a function name in the current scope. The list of `<arguments>` are comma separated T_EX macros such as `\x`, or `\y` (it is not possible to declare functions that take variable numbers of arguments). If the function takes no arguments then the parentheses need not be used. It is very important to note that the arrays that the PGF math engine supports *cannot currently be passed as arguments to functions*.

The function `<definition>` should be a sequence of statements that can be parsed by the `\tikzmath` command and should use the commands specified in the `<arguments>`. The `return` keyword (described below) should be used to indicate the value returned by the function. Although `<definition>` can take any statements accepted by `\tikzmath`, it is not advisable try to define functions inside other functions.

$$2 \times 33 = 66$$

```
\usetikzlibrary {math}
\tikzmath{
    function product(\x,\y) {
        return \x*\y;
    };
    int \i, \i, \k;
    \i = random(1,10);
    \j = random(20, 40);
    \k = product(\i, \j);
    print {\$ \i \times \j = \k \$};
}
```

```
return <expression>;
```

This keyword should be used as the last executed statement in a function definition to indicate the value that should be returned.

59.7 Executing Code Outside the Parser

Sometimes you may wish to do “something” outside the parser, perhaps display some intermediate result or execute some code. In this case you have two options. Firstly, the following keyword can be used:

```
print {code};
```

Execute *code* immediately. This is intended as convenience keyword for displaying information in a document (analogous to the `print` command in real programming languages). The *code* is executed inside a TeX group.

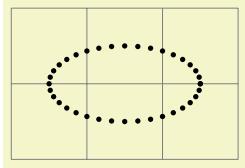
```

$$3^0 = 1, \quad 3^1 = 3, \quad 3^2 = 9, \quad 3^3 = 27, \quad 3^4 = 81, \quad 3^5 = 243, \quad 3^6 = 729,$$

```

```
\usetikzlibrary {math}
\begin{tikzmath}
    int \x, \y, \z;
    \x = random(2, 5);
    for \y in {0,...,6}{
        \z = \x^\y;
        print {\$ \x^\y = \z \$}, \y;
    };
\end{tikzmath}
```

Secondly, if a statement begins with a brace `{`, then everything up to the closing brace `}` is collected and executed (the closing brace *must* be followed by a semi-colon). Like the `print` keyword, the contents of the braces is executed inside a TeX group. Unlike the `print` keyword, the brace notation can be used in functions so that `tikz` path commands can be safely executed inside a `tikzpicture`.



```
\usetikzlibrary {math}
\begin{tikzpicture}[help lines]
\draw [help lines] grid (3,2);
\begin{tikzmath}
    coordinate \c;
    for \x in {0,10,...,360}{
        \c = (1.5cm, 1cm) + (\x:1cm and 0.5cm);
        \fill (\c) circle [radius=1pt];
    };
\end{tikzmath}
```

60 Matrix Library

TikZ Library `matrix`

```
\usetikzlibrary{matrix} % LATEX and plain TEX  
\usetikzlibrary[matrix] % ConTEXt
```

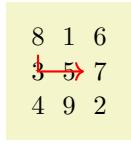
This library package defines additional styles and options for creating matrices.

60.1 Matrices of Nodes

A *matrix of nodes* is a TikZ matrix in which each cell contains a node. In this case it is bothersome having to write `\node{` at the beginning of each cell and `};` at the end of each cell. The following key simplifies typesetting such matrices.

`/tikz/matrix of nodes` (no value)

Conceptually, this key adds `\node{` at the beginning and `};` at the end of each cell and sets the `anchor` of the node to `base`. Furthermore, it adds the option `name` option to each node, where the name is set to `<matrix name>-<row number>-<column number>`. For example, if the matrix has the name `my matrix`, then the node in the upper left cell will get the name `my matrix-1-1`.



```
\usetikzlibrary {matrix}  
\begin{tikzpicture}  
  \matrix (magic) [matrix of nodes]  
  {  
    8 & 1 & 6 \\\  
    3 & 5 & 7 \\\  
    4 & 9 & 2 \\\  
  };  
  
  \draw[thick,red,->] (magic-1-1) |- (magic-2-3);  
\end{tikzpicture}
```

You may wish to add options to certain nodes in the matrix. This can be achieved in three ways.

1. You can modify, say, the `row 2 column 3` style to pass special options to this particular cell.



```
\usetikzlibrary {matrix}  
\begin{tikzpicture}[row 2 column 3/.style=red]  
  \matrix [matrix of nodes]  
  {  
    8 & 1 & 6 \\\  
    3 & 5 & 7 \\\  
    4 & 9 & 2 \\\  
  };  
\end{tikzpicture}
```

2. At the beginning of a cell, you can use a special syntax. If a cell starts with a vertical bar, then everything between this bar and the next bar is passed on to the `node` command.



```
\usetikzlibrary {matrix}  
\begin{tikzpicture}  
  \matrix [matrix of nodes]  
  {  
    8 & 1 & 6 & \\\  
    3 & 5 & |[red]| 7 & \\\  
    4 & 9 & 2 & \\\  
  };  
\end{tikzpicture}
```

You can also use an option like `| [red] (seven)|` to give a different name to the node. Note that the `&` character also takes an optional argument, which is an extra column skip.

8	1	6
3	5	7
4	9	2

```
\usetikzlibrary {matrix}
\begin{tikzpicture}
 \matrix [matrix of nodes]
 {
 8 &[1cm] 1 &[3mm] |[red]| 6 \\
 3 & 5 & |[red]| 7 \\
 4 & 9 & 2 \\
};
\end{tikzpicture}
```

3. If your cell starts with a `\path` command or any command that expands to `\path`, which includes `\draw`, `\node`, `\fill` and others, the `\node{` startup code and the `};` code are suppressed. This means that for this particular cell you can provide totally different contents.

8	1	6
3	5	7
4	9	2

```
\usetikzlibrary {matrix}
\begin{tikzpicture}
 \matrix [matrix of nodes]
 {
 8 & 1 & 6 \\
 3 & 5 & \node [red]{7}; \draw(0,0) circle(10pt); \\
 4 & 9 & 2 \\
};
\end{tikzpicture}
```

/tikz/matrix of math nodes

(no value)

This style is almost the same as the previous style, only \$ is added at the beginning and at the end of each node, so math mode will be switched on in all nodes.

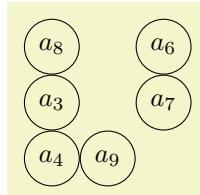
a_8	a_1	a_6
a_3	a_5	a_7
a_4	a_9	a_2

```
\usetikzlibrary {matrix}
\begin{tikzpicture}
 \matrix [matrix of math nodes]
 {
 a_8 & a_1 & a_6 \\
 a_3 & a_5 & a_7 \\
 a_4 & a_9 & a_2 \\
};
\end{tikzpicture}
```

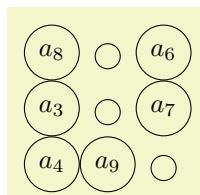
/tikz/nodes in empty cells=<true or false>

(default true)

When set to `true`, a node (with empty contents) is put in empty cells. Normally, empty cells are just, well, empty. The style can be used together with both a `matrix of nodes` and a `matrix of math nodes`.



```
\usetikzlibrary {matrix}
\begin{tikzpicture}
 \matrix [matrix of math nodes,nodes={circle,draw}]
 {
 a_8 & & a_6 \\
 a_3 & & a_7 \\
 a_4 & a_9 & \\
};
\end{tikzpicture}
```



```
\usetikzlibrary {matrix}
\begin{tikzpicture}
 \matrix [matrix of math nodes,nodes={circle,draw},nodes in empty cells]
 {
 a_8 & & a_6 \\
 a_3 & & a_7 \\
 a_4 & a_9 & \\
};
\end{tikzpicture}
```

60.2 End-of-Lines and End-of-Row Characters in Matrices of Nodes

Special care must be taken about the usage of the `\\"` command inside a matrix of nodes. The reason is that this character is overloaded in TeX: On the one hand, it is used to denote the end of a line in normal

text; on the other hand it is used to denote the end of a row in a matrix. Now, if a matrix contains node which in turn may have multiple lines, it is unclear which meaning of `\\"` should be used.

This problem arises only when you use the `text width` option of nodes. Suppose you write a line like

```
\matrix [text width=5cm,matrix of nodes]
{
  first row & upper line \\ lower line \\
  second row & hmm \\
};
```

This leaves TeX trying to riddle out how many rows this matrix should have. Do you want two rows with the upper right cell containing a two-line text. Or did you mean a three row matrix with the second row having only one cell?

Since TeX is not clairvoyant, the following rules are used:

1. Inside a matrix, the `\\"` command, by default, signals the end of the row, not the end of a line in a cell.
2. However, there is an exception to this rule: If a cell starts with a TeX-group (this is, with `{`), then inside this first group the `\\"` command retains the meaning of “end of line” character. Note that this special rule works only for the first group in a cell and this group must be at the beginning.

The net effect of these rules is the following: Normally, `\\"` is an end-of-row indicator; if you want to use it as an end-of-line indicator in a cell, just put the whole cell in curly braces. The following example illustrates the difference:

row 1	upper line
lower line	
row 2	hmm

```
\usetikzlibrary {matrix}
\begin{tikzpicture}
 \matrix [matrix of nodes, nodes={text width=16mm, draw}]
 {
   row 1 & upper line \\ lower line \\
   row 2 & hmm \\
 };
\end{tikzpicture}
```

row 1	upper line
	lower line
row 2	hmm

```
\usetikzlibrary {matrix}
\begin{tikzpicture}
 \matrix [matrix of nodes, nodes={text width=16mm, draw}]
 {
   row 1 & {upper line \\ lower line} \\
   row 2 & hmm \\
 };
\end{tikzpicture}
```

Note that this system is not fool-proof. If you write things like `a&b{c\\d}\\` in a matrix of nodes, an error will result (because the second cell did not start with a brace, so `\\"` retained its normal meaning and, thus, the second cell contained the text `b{c`, which is not balanced with respect to the number of braces).

60.3 Delimiters

Delimiters are parentheses or braces to the left and right of a formula or a matrix. The `matrix` library offers options for adding such delimiters to a matrix. However, delimiters can actually be added to any node that has the standard anchors `north`, `south`, `north west` and so on. In particular, you can add delimiters to any `rectangle` box. They are implemented by “measuring the height” of the node and then adding a delimiter of the correct size to the left or right using some after node magic.

`/tikz/left delimiter=<delimiter>` (no default)

This option can be given to any node that has the standard anchors `north`, `south` and so on. The `<delimiter>` can be any delimiter that is acceptable to TeX’s `\left` command.

$$\left(\begin{array}{ccc} a_8 & a_1 & a_6 \\ a_3 & a_5 & a_7 \\ a_4 & a_9 & a_2 \end{array} \right)$$

```
\usetikzlibrary {matrix}
\begin{tikzpicture}
 \matrix [matrix of math nodes, left delimiter=(,right delimiter=)]
 {
   a_8 & a_1 & a_6 \\
   a_3 & a_5 & a_7 \\
   a_4 & a_9 & a_2 \\
 };
\end{tikzpicture}
```

$$\left(\int_0^1 x \, dx \right)$$

```
\usetikzlibrary {matrix}
\begin{tikzpicture}
  \node [fill=red!20, left delimiter=(, right delimiter=)] {
    $\displaystyle \int_0^1 x \, dx$};
\end{tikzpicture}
```

/tikz/every delimiter

(style, initially empty)

This style is executed for every delimiter. You can use it to shift or color delimiters or do whatever.

/tikz/every left delimiter

(style, initially empty)

This style is additionally executed for every left delimiter.

$$\left(\begin{array}{ccc} a_8 & a_1 & a_6 \\ a_3 & a_5 & a_7 \\ a_4 & a_9 & a_2 \end{array} \right)$$

```
\usetikzlibrary {matrix}
\begin{tikzpicture}
  [every left delimiter/.style={red,xshift=1ex},
   every right delimiter/.style={xshift=-1ex}]
  \matrix [matrix of math nodes, left delimiter=(, right delimiter=)] {
    a_8 & a_1 & a_6 \\
    a_3 & a_5 & a_7 \\
    a_4 & a_9 & a_2 \\
  };
\end{tikzpicture}
```

/tikz/right delimiter=<delimiter>

(no default)

Works as above.

/tikz/every right delimiter

(style, initially empty)

Works as above.

/tikz/above delimiter=<delimiter>

(no default)

This option allows you to add a delimiter above the node. It is implemented by rotating a left delimiter.

$$\left\| \begin{array}{ccc} a_8 & a_1 & a_6 \\ a_3 & a_5 & a_7 \\ a_4 & a_9 & a_2 \end{array} \right\|$$

```
\usetikzlibrary {matrix}
\begin{tikzpicture}
  \matrix [matrix of math nodes,%
          left delimiter=\|,right delimiter=\rmoustache,%
          above delimiter=(,below delimiter=)] {
    a_8 & a_1 & a_6 \\
    a_3 & a_5 & a_7 \\
    a_4 & a_9 & a_2 \\
  };
\end{tikzpicture}
```

/tikz/every above delimiter

(style, initially empty)

Works as above.

/tikz/below delimiter=<delimiter>

(no default)

Works as above.

/tikz/every below delimiter

(style, initially empty)

Works as above.

61 Mindmap Drawing Library

TikZ Library `mindmap`

```
\usetikzlibrary{mindmap} % LATEX and plain TEX  
\usetikzlibrary[mindmap] % ConTEXt
```

This package provides styles for drawing mindmap diagrams.

61.1 Overview

This library is intended to make the creation of mindmaps or concept maps easier. A *mindmap* is a graphical representation of a concept together with related concepts and annotations. Mindmaps are, essentially, trees, possibly with a few extra edges added, but they are usually drawn in a special way: The root concept is placed in the middle of the page and is drawn as a huge circle, ellipse, or cloud. The related concepts then “leave” this root concept via branch-like tendrils.

The `mindmap` library of TikZ produces mindmaps that look a bit different from the standard mindmaps: While the big root concept is still a circle, related concepts are also depicted as (smaller) circles. The related concepts are linked to the root concept via organic-looking connections. The overall effect is visually rather pleasing, but readers may not immediately think of a mindmap when they see a picture created with this library.

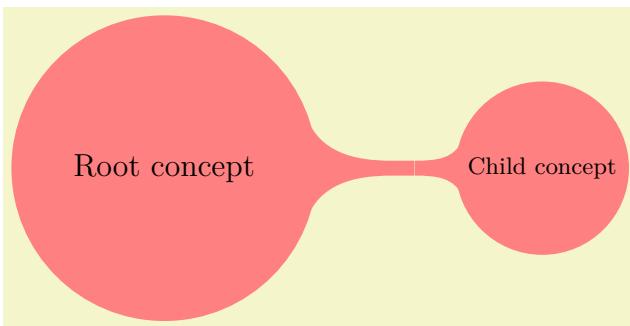
Although it is not strictly necessary, you will usually create mindmaps using TikZ’s tree mechanism and some of the styles and macros of the package work best when used inside trees. However, it is still possible and sometimes necessary to treat parts of a mindmap as a graph with arbitrary edges and this is also possible.

61.2 The Mindmap Style

Every mindmap should be put in a scope or a picture where the `mindmap` style is used. This style installs some internal settings.

`/tikz/mindmap` (style, no value)

Use this style with all pictures or at least scopes that contain a mindmap. It installs a whole bunch of settings that are useful for drawing mindmaps.

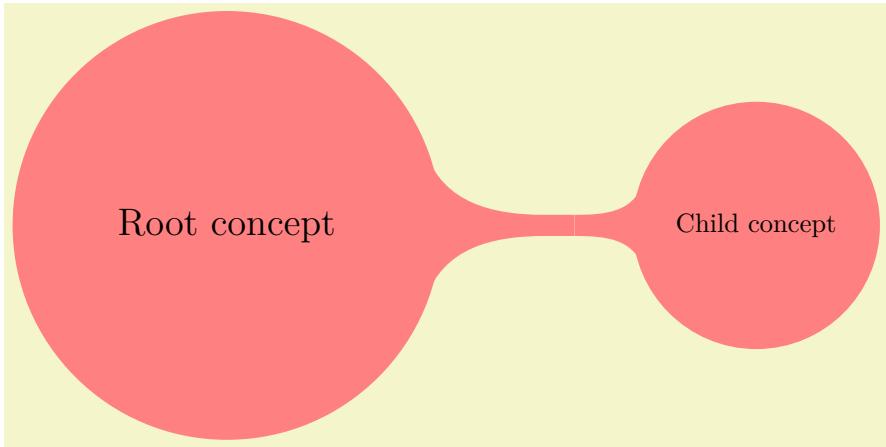


```
\usetikzlibrary {mindmap}  
\tikz[mindmap,concept color=red!50]  
  \node [concept] {Root concept}  
    child[grow=right] {node[concept] {Child concept}};
```

The sizes of concepts are predefined in such a way that a medium-size mindmap will fit on an A4 page (more or less).

`/tikz/every mindmap` (style, no value)

This style is included by the `mindmap` style. Change this style to add special settings to your mindmaps.



```
\usetikzlibrary {mindmap}
\tikz[large mindmap,concept color=red!50]
\node [concept] {Root concept}
    child[grow=right] {node[concept] {Child concept}};
```

Remark: Note that `mindmap` redefines font sizes and `sibling angle` depending on the current concept level (i.e. inside of `level 1 concept`, `level 2 concept` etc.). Thus, if you need to redefine these variables, use

`level 1 concept/.append style={font=\small}`

or

`level 2 concept/.append style={sibling distance=90}`

after the `mindmap` style.

/tikz/small mindmap (style, no value)

This style includes the `mindmap` style, but additionally changes the default size of concepts, fonts and distances so that a medium-sized mindmap will fit on an A5 page (A5 pages are half as large as A4 pages). Mindmaps with `small mindmap` will also fit onto a standard frame of the `beamer` package.

/tikz/large mindmap (style, no value)

This style includes the `mindmap` style, but additionally changes the default size of concepts, fonts and distances so that a medium-sized mindmap will fit on an A3 page (A3 pages are twice as large as A4 pages).

/tikz/huge mindmap (style, no value)

This style causes concepts to be even bigger and it is best used with A2 paper and above.

61.3 Concepts Nodes

The basic entities of mindmaps are called *concepts* in Ti_KZ. A concept is a node of style `concept` and it must be circular for some of the connection macros to work.

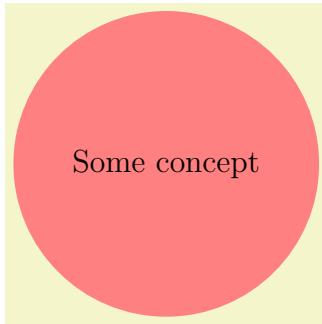
61.3.1 Isolated Concepts

The following styles influence how isolated concepts are rendered:

/tikz/concept (style, no value)

This style should be used with all nodes that are concepts, although some styles like `extra concept` install this style automatically.

Basically, this style makes the concept node circular and installs a uniform color called `concept color`, see below. Additionally, the style `every concept` is called.



```
\usetikzlibrary {mindmap}
\tikz[mindmap,concept color=red!50] \node [concept] {Some concept};
```

`/tikz/every concept`

(style, no value)

In order to change the appearance of concept nodes, you should change this style. Note, however, that the color of a concept should be uniform for some of the connection bar stuff to work, so you should not change the color or the draw/fill state of concepts using this option. It is mostly useful for changing the text color and font.

`/tikz/concept color=(color)`

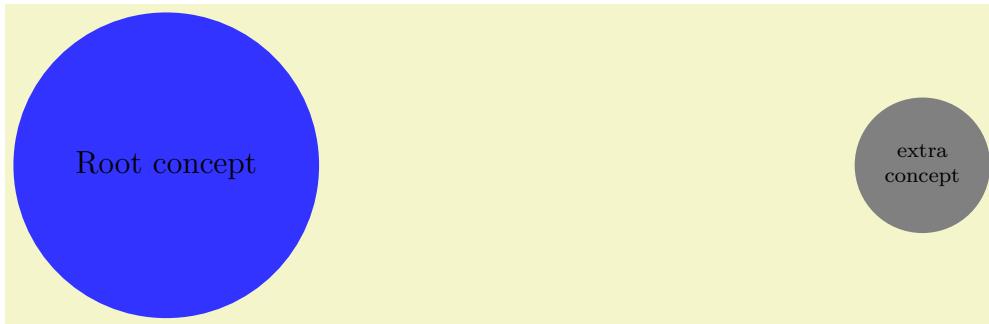
(no default)

This option tells TikZ which color should be used for filling and stroking concepts. The difference between this option and just setting `every concept` to the desired color is that this option allows TikZ to keep track of the colors used for concepts. This is important when you *change* the color between two connected concepts. In this case, TikZ can automatically create a shading that provides a smooth transition between the old and the new concept color; we will come back to this in the next section.

`/tikz/extra concept`

(style, no value)

This style is intended for concepts that are not part of the “mindmap tree”, but stand beside it. Typically, they will have a subdued color or be smaller. In order to have these concepts appear in a uniform way and in order to indicate in the code that these concepts are additional, you can use this style.



```
\usetikzlibrary {mindmap}
\begin{tikzpicture}[mindmap,concept color=blue!80]
\node [concept] {Root concept};
\node [extra concept] at (10,0) {extra concept};
\end{tikzpicture}
```

`/tikz/every extra concept`

(style, no value)

Change this style to change the appearance of extra concepts.

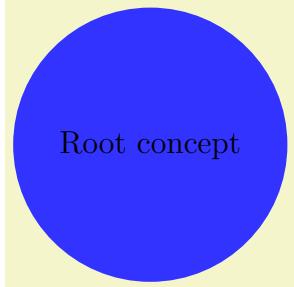
61.3.2 Concepts in Trees

As pointed out earlier, TikZ assumes that your mindmap is built using the `child` facilities of TikZ. There are numerous options that influence how concepts are rendered at the different levels of a tree.

/tikz/root concept

(style, no value)

This style is used for the roots of mindmap trees. By adding something to this, you can change how the root of a mindmap will be rendered.



```
\usetikzlibrary {mindmap}
\begin{tikzpicture}
  [root concept/.append style={concept color=blue!80,minimum size=3.5cm},
   mindmap]
  \node [concept] {Root concept};

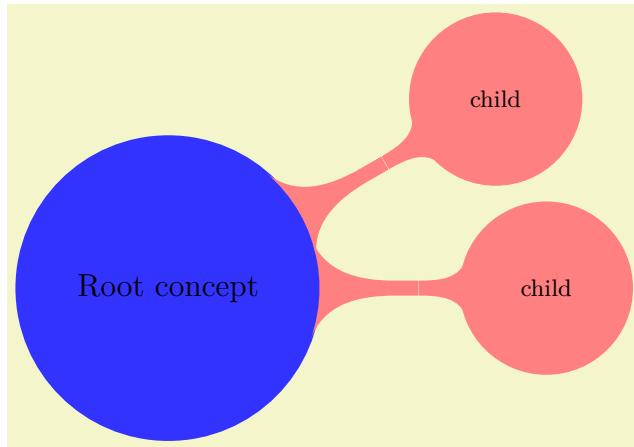
```

Note that styles like `large` `mindmap` redefine these styles, so you should add something to this style only inside the picture.

/tikz/level 1 concept

(style, no value)

The `mindmap` style adds this style to the `level 1` style. This means that the first level children of a mindmap tree will use this style.



```
\usetikzlibrary {mindmap}
\begin{tikzpicture}
  [root concept/.append style={concept color=blue!80},
   level 1 concept/.append style={concept color=red!50},
   mindmap]
  \node [concept] {Root concept}
    child[grow=30] {node[concept] {child}}
    child[grow=0] {node[concept] {child}};

```

/tikz/level 2 concept

(style, no value)

Works like `level 1 concept`, only for second level children.

/tikz/level 3 concept

(style, no value)

Works like `level 1 concept`.

/tikz/level 4 concept

(style, no value)

Works like `level 1 concept`. Note that there are no fifth and higher level styles, you need to modify `level 5` directly in such cases.

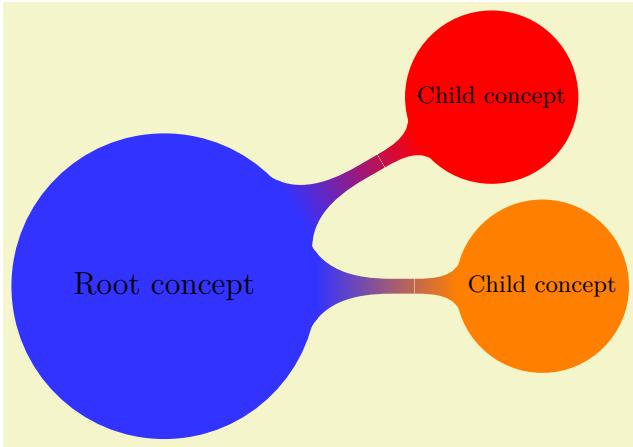
/tikz/concept color=<color>

(no default)

We saw already that this option is used to change the color of concepts. We now have a look at its effect when used on child nodes of a concept. Normally, this option simply changes the color of the children.

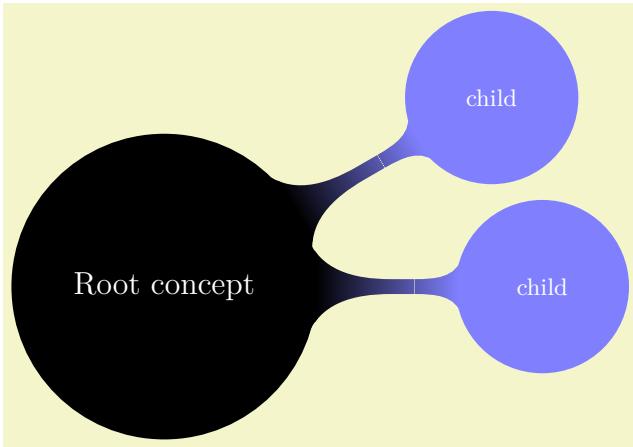
However, when the option is given as an option to the `child` operation (and not to the `node` operation and also not as an option to all children via the `level 1 style`), TikZ will smoothly change the concept color from the parent's color to the color of the child concept.

Here is an example:



```
\usetikzlibrary {mindmap}
\tikz[mindmap,concept color=blue!80]
\node [concept] {Root concept}
    child[concept color=red,grow=30] {node[concept] {Child concept}}
    child[concept color=orange,grow=0] {node[concept] {Child concept}};
```

In order to have a concept color which changes with the hierarchy level, a tiny bit of magic is needed:



```
\usetikzlibrary {mindmap}
\tikz[mindmap,text=white,
      root concept/.style={concept color=blue},
      level 1 concept/.append style=
      {every child/.style={concept color=blue!50}}]
\node [concept] {Root concept}
    child[grow=30] {node[concept] {child}}
    child[grow=0] {node[concept] {child}};
```

61.4 Connecting Concepts

61.4.1 Simple Connections

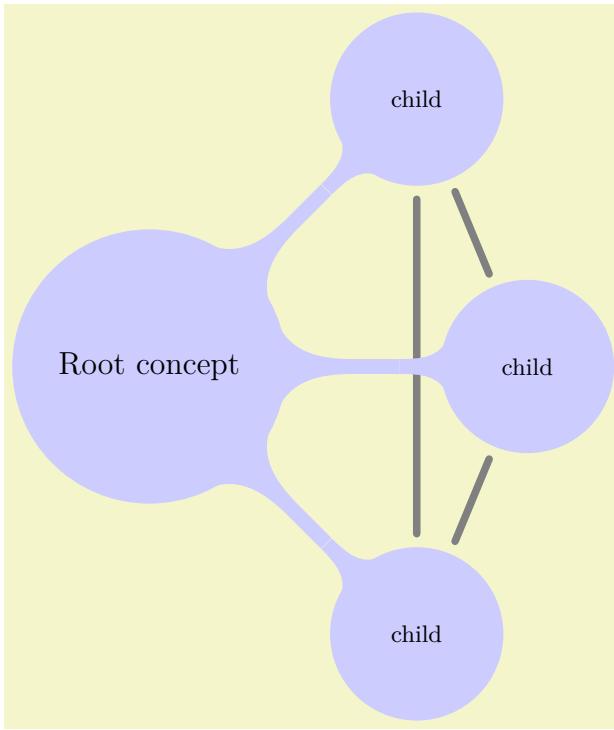
The easiest way to connect two concepts is to draw a line between them. In order to give such lines a consistent appearance, it is recommendable to use the following style when drawing such lines:

`/tikz/concept connection`

(style, no value)

This style can be used for lines between two concepts. Feel free to redefine this style.

A problem arises when you need to connect concepts after the main mindmap has been drawn. In this case you will want the connection lines to lie *behind* the main mindmap. However, you can draw the lines only after the coordinates of the concepts have been determined. In this case you should place the connecting lines on a background layer as in the following example:



```
\usetikzlibrary {backgrounds,mindmap}
\begin{tikzpicture}
[root concept/.append style={concept color=blue!20,minimum size=2cm},
 level 1 concept/.append style={sibling angle=45},
 mindmap]
\node [concept] {Root concept}
 [clockwise from=45]
 child { node[concept] (c1) {child}}
 child { node[concept] (c2) {child}}
 child { node[concept] (c3) {child}};
\begin{pgfonlayer}{background}
 \draw [concept connection] (c1) edge (c2)
 edge (c3)
 (c2) edge (c3);
\end{pgfonlayer}
\end{tikzpicture}
```

61.4.2 The Circle Connection Bar Decoration

Instead of a simple line between two concepts, you can also add a bar between the two nodes that has slightly organic ends. These bars are also used by default as the edges from parents in the mindmap tree.

For the drawing of the bars a special decoration is used, which is defined in the `mindmap` library:

Decoration `circle connection bar`

This decoration can be used to connect two circles. The start of the to-be-decorated path should lie on the border of the first circle, the end should lie on the border of the second circle. The following two decoration keys should be initialized with the sizes of the circles:

- `start radius`
- `end radius`

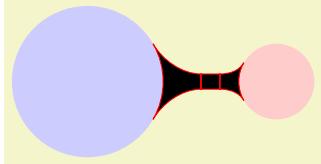
Furthermore, the following two decoration keys influence the decoration:

- `amplitude`

- `angle`

The decoration turns a straight line into a path that starts on the border of the first circle at the specified angle relative to the line connecting the centers of the circles. The path then changes into a rectangle whose thickness is given by the amplitude. Finally, the path ends with the same angles on the second circle.

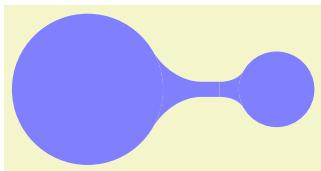
Here is an example that should make this clearer:



```
\usetikzlibrary {mindmap}
\begin{tikzpicture}
    [decoration={start radius=1cm,end radius=.5cm,amplitude=2mm,angle=30}]
    \fill[blue!20] (0,0) circle (1cm);
    \fill[red!20] (2.5,0) circle (.5cm);

    \filldraw [draw=red,fill=black,
               decorate,decoration=circle connection bar] (1,0) -- (2,0);
\end{tikzpicture}
```

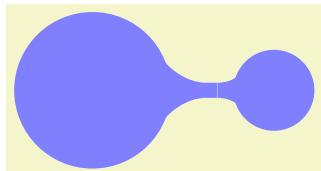
As can be seen, the decorated path consists of three parts and is not really useful for drawing. However, if you fill the decorated path only, and if you use the same color as for the circles, the result is better.



```
\usetikzlibrary {mindmap}
\begin{tikzpicture}
    [blue!50,decoration={start radius=1cm,
                         end radius=.5cm,amplitude=2mm,angle=30}]
    \fill (0,0) circle (1cm);
    \fill (2.5,0) circle (.5cm);

    \fill [decorate,decoration=circle connection bar] (1,0) -- (2,0);
\end{tikzpicture}
```

In the above example you may notice the small white line between the circles and the decorated path. This is due to rounding errors. Unfortunately, for larger distances, the errors can accumulate quite strongly, especially since TikZ and TeX are not very good at computing square roots. For this reason, it is a good idea to make the circles slightly larger to cover up such problems. When using nodes of shape `circle`, you can just add the `draw` option with a `line width` of one or two points (for very large distances you may need line width up to 4pt).



```
\usetikzlibrary {mindmap}
\begin{tikzpicture}
    [blue!50,decoration={start radius=1cm,
                         end radius=.5cm,amplitude=2mm,angle=30}]
    \fill (0,0) circle (1cm+1pt);
    \fill (2.4,0) circle (.5cm+1pt);

    \fill [decorate,decoration=circle connection bar] (1,0) -- (1.9,0);
\end{tikzpicture}
```

61.4.3 The Circle Connection Bar To-Path

The `circle connection bar` decoration is a bit complicated to use. Especially specifying the radii is quite bothersome (the amplitude and the angle can be set once and for all). For this reason, the `mindmap` library defines a special to-path that performs the necessary computations for you.

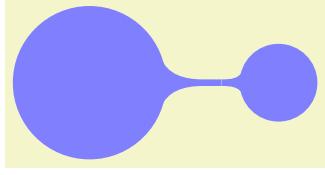
`/tikz/circle connection bar` (style, no value)

This style installs a rather involved to-path. Unlike normal to-paths, this path requires that the start and the target of the to-path are named nodes of shape `circle` – if this is not the case, this path will produce errors.

Assuming that the start and the target are circles, the to-path will first compute the radii of these circles (by measuring the distance from the `center` anchor to some anchor on the border) and will set the `start circle` keys accordingly. Next, the `fill` option is set to the `concept color` while `draw=none` is set. The decoration is set to `circle connection bar`. Finally, the following style is included:

`/tikz/every circle connection bar` (style, no value)

Redefine this style to change the appearance of circle connection bar to-paths.



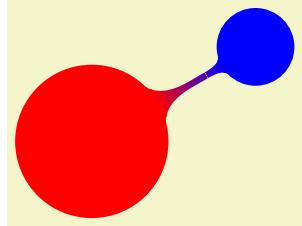
```
\usetikzlibrary {mindmap}
\begin{tikzpicture}[concept color=blue!50,blue!50,outer sep=0pt]
  \node (n1) at (0,0) [circle,minimum size=2cm,fill,draw,thick] {};
  \node (n2) at (2.5,0) [circle,minimum size=1cm,fill,draw,thick] {};
  \path (n1) to[circle connection bar] (n2);
\end{tikzpicture}
```

Note that it is not a good idea to have more than one `to` operation together with the option `circle connection bar` in a single `\path`. Use the `edge` operation, instead, for creating multiple connections and this operation creates a new scope for each edge.

In a mindmap we sometimes want colors to change from one concept color to another. Then, the connection bar should, ideally, consist of a smooth transition between these two colors. Getting this right using shadings is a bit tricky if you try this “by hand”, so the `mindmap` library provides a special option for facilitating this procedure.

`/tikz/circle connection bar switch color=from(<first color>)to(<second color>)` (no default)

This style works similarly to the `circle connection bar`. The only difference is that instead of filling the path with a single color a shading is used.



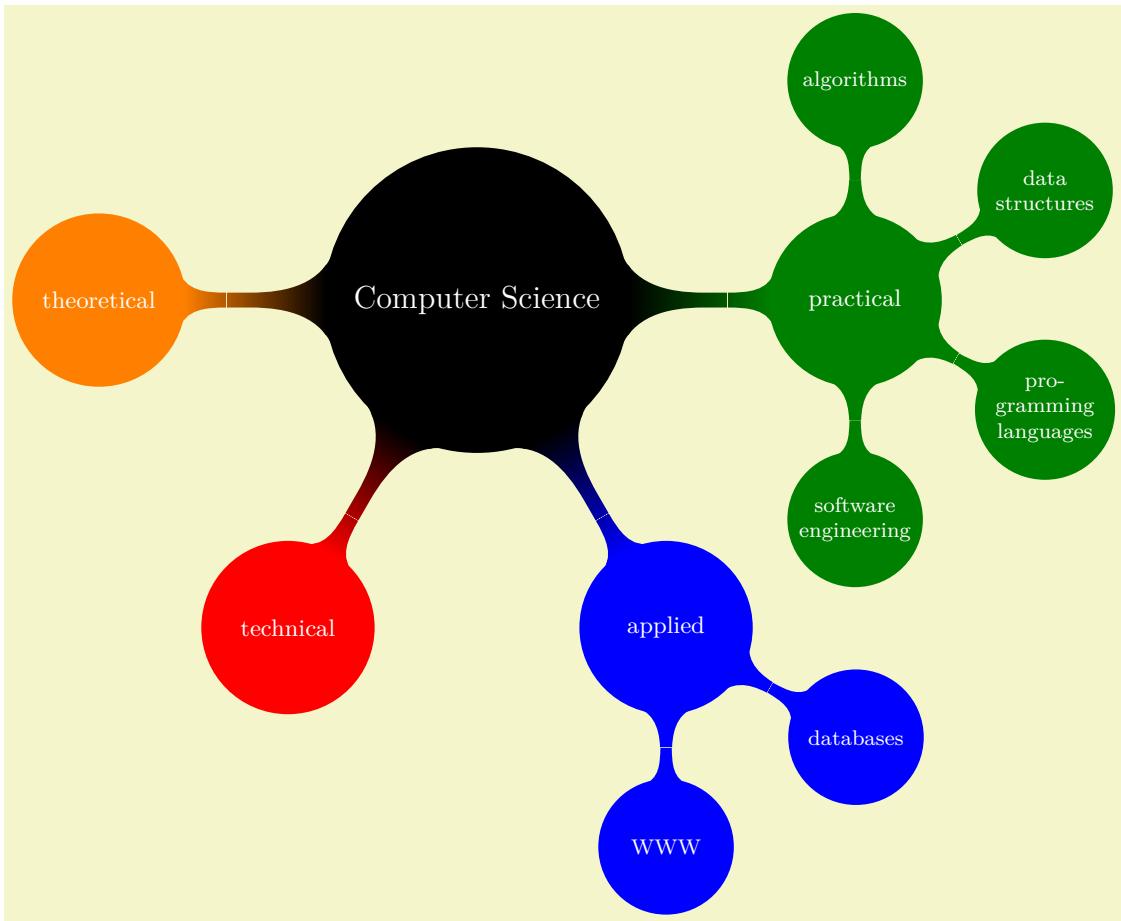
```
\usetikzlibrary {mindmap}
\begin{tikzpicture}[outer sep=0pt]
  \node (n1) at (0,0) [circle,minimum size=2cm,fill,draw,thick,red] {};
  \node (n2) at (30:2.5) [circle,minimum size=1cm,fill,draw,thick,blue] {};
  \path (n1) to[circle connection bar switch color=from (red) to (blue)] (n2);
\end{tikzpicture}
```

61.4.4 Tree Edges

Most of the time, concepts in a mindmap are connected automatically when the mindmap is built as a tree. The reason is that the `mindmap` installs a `circle connection bar` path as the `edge from parent` path. Also, the `mindmap` option takes care of things like setting the correct `draw` and `outer sep` settings and some other stuff.

In detail, the `mindmap` option sets the `edge from parent` path to a path that uses the `to`-path `circle connection bar` to connect the parent node and the child node. The `concept color` option (locally) changes this by using `circle connection bar switch color` instead with the from-color set to the old (parent’s) concept color and the to-color set to the new (child’s) concept color. This means that when you provide the `concept color` option to a `child` command, the color will change from the parent’s concept color to the specified color.

Here is an example of a tree built in this way:



```

\usetikzlibrary {mindmap}
\begin{tikzpicture}
\path [mindmap,concept color=black,text=white]
node[concept] {Computer Science}
[clockwise from=0]
% note that 'sibling angle' can only be defined in
% 'level 1 concept/.append style={}'
child[concept color=green!50!black] {
    node[concept] {practical}
    [clockwise from=90]
    child { node[concept] {algorithms} }
    child { node[concept] {data structures} }
    child { node[concept] {pro\-\-gramming languages} }
    child { node[concept] {software engineer\-\-ing} }
}
child[concept color=blue] {
    node[concept] {applied}
    [clockwise from=-30]
    child { node[concept] {databases} }
    child { node[concept] {WWW} }
}
child[concept color=red] { node[concept] {technical} }
child[concept color=orange] { node[concept] {theoretical} };
\end{tikzpicture}

```

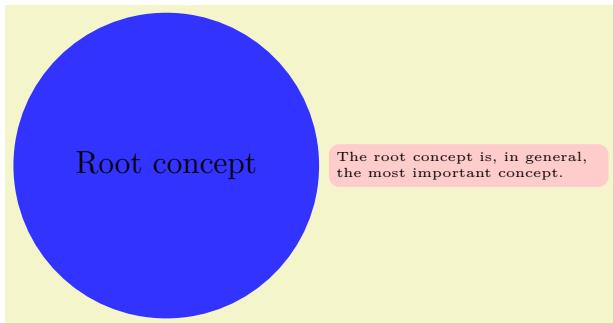
61.5 Adding Annotations

An *annotation* is some text outside a mindmap that, unlike an extra concept, simply explains something in the mindmap. The following style is mainly intended to help readers of the code see that a node in an annotation node.

`/tikz/annotation`

(style, no value)

This style indicates that a node is an annotation node. It includes the style `every annotation`, which allows you to change this style in a convenient fashion.



```
\usetikzlibrary {mindmap}
\begin{tikzpicture}
[mindmap,concept color=blue!80,
 every annotation/.style={fill=red!20}]
\node [concept] (root) {Root concept};

\node [annotation,right] at (root.east)
{The root concept is, in general, the most important concept.};
\end{tikzpicture}
```

`/tikz/every annotation`

(style, no value)

This style is included by `annotation`.

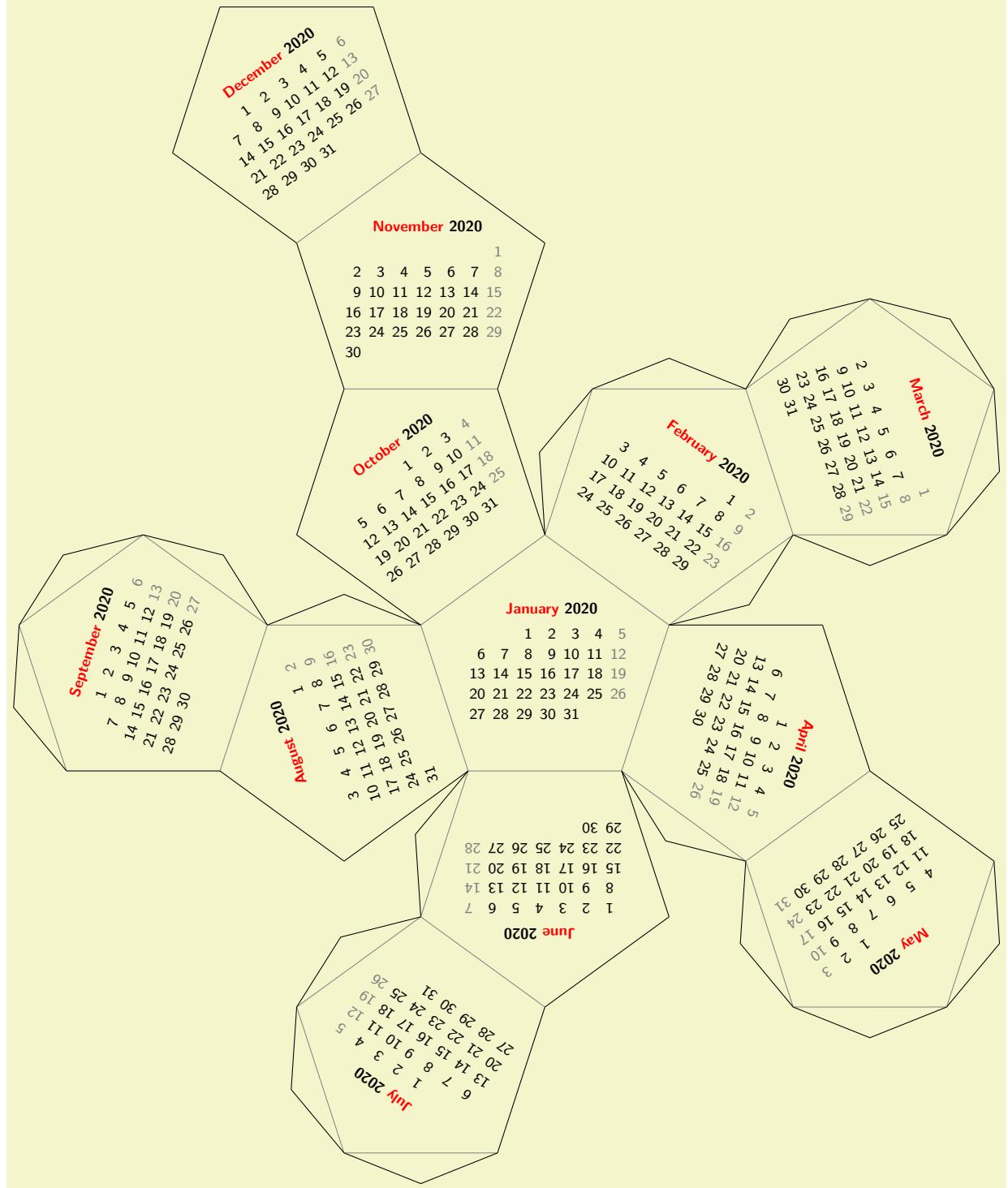
62 Paper-Folding Diagrams Library

TikZ Library `folding`

```
\usetikzlibrary{folding} % LATEX and plain TEX
\usetikzlibrary[folding] % ConTEX
```

This library defines pic types for creating paper-folding diagrams. Many thanks to Nico van Cleemput for providing most of the code.

Here is a big example that produces a diagram for a calendar:



```
\usetikzlibrary {calendar,folding}
\sffamily\scriptsize
\tikz \pic [
    transform shape,
    every calendar/.style={
        at={(-8ex,4ex)},
        week list,
        month label above centered,
        month text=\bfseries\textcolor{red}{\%mt} \%y0,
        if={(Sunday) [black!50]}
    },
    folding line length=2.5cm,
    face 1={ \calendar [dates=\the\year-01-01 to \the\year-01-last];},
    face 2={ \calendar [dates=\the\year-02-01 to \the\year-02-last];},
    face 3={ \calendar [dates=\the\year-03-01 to \the\year-03-last];},
    face 4={ \calendar [dates=\the\year-04-01 to \the\year-04-last];},
    face 5={ \calendar [dates=\the\year-05-01 to \the\year-05-last];},
    face 6={ \calendar [dates=\the\year-06-01 to \the\year-06-last];},
    face 7={ \calendar [dates=\the\year-07-01 to \the\year-07-last];},
    face 8={ \calendar [dates=\the\year-08-01 to \the\year-08-last];},
    face 9={ \calendar [dates=\the\year-09-01 to \the\year-09-last];},
    face 10={\calendar [dates=\the\year-10-01 to \the\year-10-last];},
    face 11={\calendar [dates=\the\year-11-01 to \the\year-11-last];},
    face 12={\calendar [dates=\the\year-12-01 to \the\year-12-last];}
] {dodecahedron folding};
```

The foldings are sorted by number of faces.

Pic type `tetrahedron folding`

This pic type draws a folding diagram for a tetrahedron. The following keys influence the pic:

`/tikz/folding line length=<dimension>` (no default)

Sets the length of the base line for folding. For the dodecahedron this is the length of all the sides of the pentagons.

`/tikz/face 1=<code>` (no default)

The `<code>` is executed for the first face of the dodecahedron. When it is executed, the coordinate system will have been shifted and rotated such that it lies at the middle of the first face of the dodecahedron.

`/tikz/face 2=<code>` (no default)

Same as `face 1`, but for the second face.

`/tikz/face 3=<code>` (no default)

`/tikz/face 4=<code>` (no default)

There are further similar options for more faces (for commands shown later).

Here is a simple example:



```
\usetikzlibrary {folding}
\tikz \pic [
    transform shape,
    folding line length=6mm,
    face 1={ \node[red] {1};},
    face 2={ \node {2};},
    face 3={ \node {3};},
    face 4={ \node {4};}
] {tetrahedron folding};
```

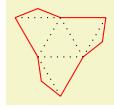
The appearance of the cut and folding lines can be influenced using the following styles:

`/tikz/every cut` (style, initially empty)

Executed for every line that should be cut using scissors.

`/tikz/every fold` (style, initially `help lines`)

Executed for every line that should be folded.



```
\usetikzlibrary {folding}
\tikz \pic [
  every cut/.style=red,
  every fold/.style=dotted,
  folding line length=6mm
] { tetrahedron folding };
```

There is one style that is mainly useful for the present documentation:

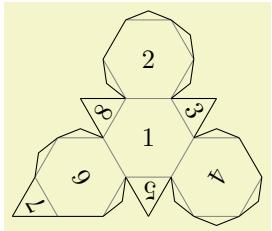
/tikz/numbered faces

(style, no value)

Sets face $\langle i \rangle$ to `\node {\langle i \rangle};` for all i .

Pic type **tetrahedron truncated folding**

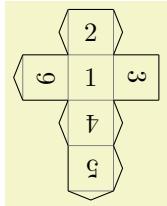
A folding of a truncated tetrahedron.



```
\usetikzlibrary {folding}
\tikz \pic [folding line length=6mm, numbered faces, transform shape]
{ tetrahedron truncated folding };
```

Pic type **cube folding**

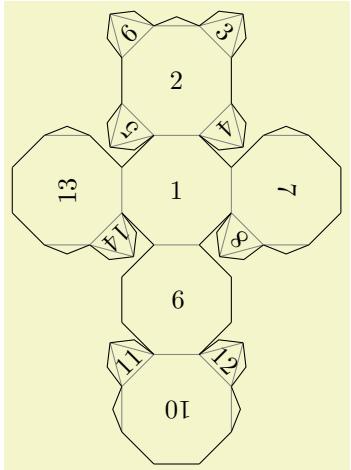
A folding of a cube.



```
\usetikzlibrary {folding}
\tikz \pic [folding line length=6mm, numbered faces, transform shape]
{ cube folding };
```

Pic type **cube truncated folding**

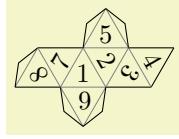
A folding of a truncated cube.



```
\usetikzlibrary {folding}
\tikz \pic [folding line length=6mm, numbered faces, transform shape]
{ cube truncated folding };
```

Pic type **octahedron folding**

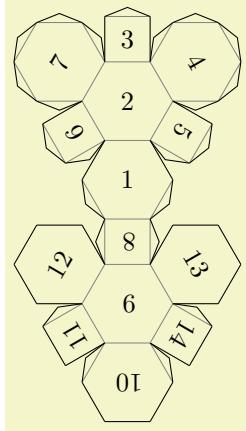
A folding of an octahedron.



```
\usetikzlibrary {folding}
\begin{tikzpicture} [folding line length=6mm, numbered faces, transform shape]
\octahedronFolding;
\end{tikzpicture}
```

Pic type **octahedron folding**

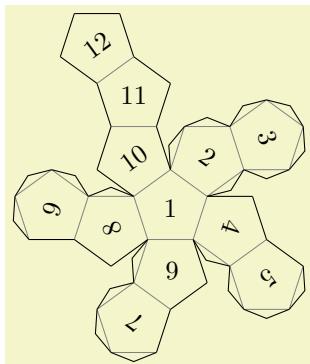
A folding of a truncated octahedron.



```
\usetikzlibrary {folding}
\begin{tikzpicture} [folding line length=6mm, numbered faces, transform shape]
\octahedronTruncatedFolding;
\end{tikzpicture}
```

Pic type **dodecahedron folding**

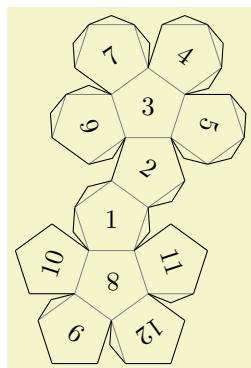
A folding of a dodecahedron.



```
\usetikzlibrary {folding}
\begin{tikzpicture} [folding line length=6mm, numbered faces, transform shape]
\dodecahedronFolding;
\end{tikzpicture}
```

Pic type **dodecahedron' folding**

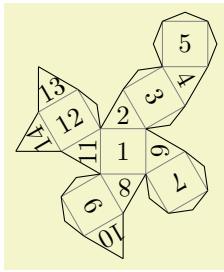
This is an alternative folding of a dodecahedron.



```
\usetikzlibrary {folding}
\begin{tikzpicture} [folding line length=6mm, numbered faces, transform shape]
\dodecahedronFolding;
\end{tikzpicture}
```

Pic type cuboctahedron folding

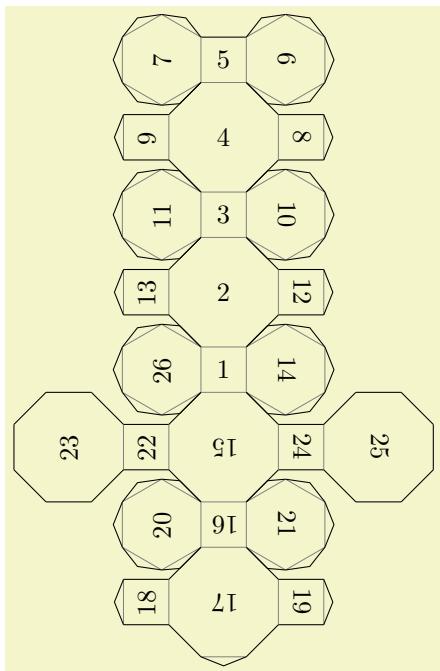
A folding of a cuboctahedron.



```
\usetikzlibrary {folding}
\begin{tikzpicture} [folding line length=6mm, numbered faces, transform shape]
\cuboctahedron folding;
\end{tikzpicture}
```

Pic type cuboctahedron truncated folding

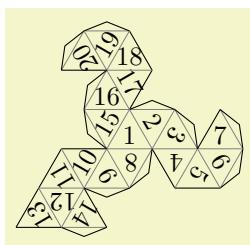
A folding of a truncated cuboctahedron.



```
\usetikzlibrary {folding}
\begin{tikzpicture} [folding line length=6mm, numbered faces, transform shape]
\cuboctahedron truncated folding;
\end{tikzpicture}
```

Pic type icosahedron folding

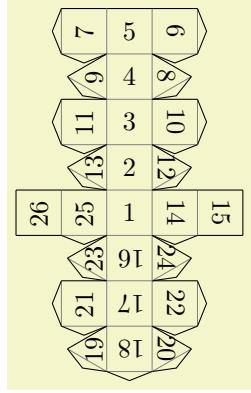
A folding of an icosahedron.



```
\usetikzlibrary {folding}
\begin{tikzpicture} [folding line length=6mm, numbered faces, transform shape]
\icosahedron folding;
\end{tikzpicture}
```

Pic type rhombicuboctahedron folding

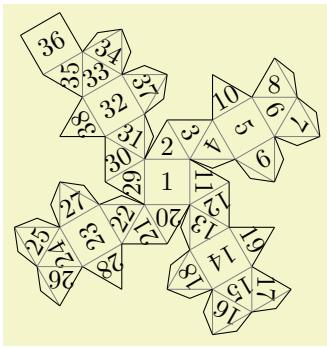
A folding of an rhombicuboctahedron.



```
\usetikzlibrary {folding}
\begin{tikzpicture}[folding line length=6mm, numbered faces, transform shape]
  \pic [rhombicuboctahedron folding] {};
\end{tikzpicture}
```

Pic type **snub cube folding**

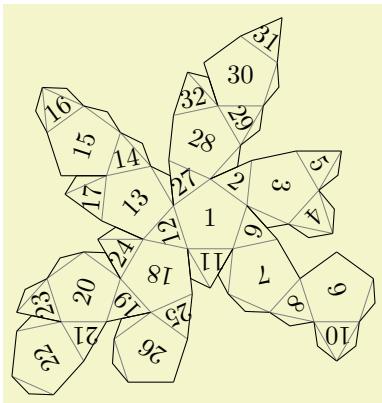
A folding of a snub cube.



```
\usetikzlibrary {folding}
\begin{tikzpicture}[folding line length=6mm, numbered faces, transform shape]
  \pic [snub cube folding] {};
\end{tikzpicture}
```

Pic type **icosidodecahedron folding**

A folding of an icosidodecahedron.



```
\usetikzlibrary {folding}
\begin{tikzpicture}[folding line length=6mm, numbered faces, transform shape]
  \pic [icosidodecahedron folding] {};
\end{tikzpicture}
```

63 Pattern Library

TikZ Library `patterns`

```
\usepgflibrary{patterns} % LATEX and plain TEX and pure pgf
\usepgflibrary[patterns] % ConTEXt and pure pgf
\usetikzlibrary{patterns} % LATEX and plain TEX when using TikZ
\usetikzlibrary[patterns] % ConTEXt when using TikZ
```

The package defines patterns for filling areas.

63.1 Form-Only Patterns

Pattern name	Example (pattern in black, blue, and red on faded checkerboard)		
horizontal lines			
vertical lines			
north east lines			
north west lines			
grid			
crosshatch			
dots			
crosshatch dots			
fivepointed stars			
sixpointed stars			
bricks			
checkerboard			

63.2 Inherently Colored Patterns

Pattern name	Example
checkerboard light gray	
horizontal lines light gray	
horizontal lines gray	
horizontal lines dark gray	
horizontal lines light blue	
horizontal lines dark blue	
crosshatch dots gray	
crosshatch dots light steel blue	

63.3 User-Defined Patterns

by Mark Wibrow

TikZ Library `patterns.meta`

```
\usepgflibrary{patterns.meta} % LATEX and plain TEX and pure pgf
\usepgflibrary[patterns.meta] % ConTeXt and pure pgf
\usetikzlibrary{patterns.meta} % LATEX and plain TEX when using TikZ
\usetikzlibrary[patterns.meta] % ConTeXt when using TikZ
```

Define your own patterns with a syntax similar to `arrows.meta`.

Caveat: This library is currently experimental and might change without notice. There are some known shortcomings that will hopefully be fixed in the future.

`\pgfdeclarepattern{<config>}`

This command is used to declare a new pattern. In contrast to the normal patterns and in the spirit of `arrows.meta` this command takes a list of keys and values to define the pattern. The following keys are available:

`/pgf/patterns/name=<name>` (no default)

The name of the pattern by which it can be used later on.

`/pgf/patterns/type=<type>` (default `uncolored`)

The type of the pattern maps to what was called “form only” and “inherently colored” in the language of the normal patterns. The available choices are:

- `uncolored` the pattern will obey the surrounding color.
- `colored` the pattern will have an intrinsic color.
- `form only` synonym for `uncolored`
- `inherently colored` synonym for `colored`

`/pgf/patterns/x=<dimension>` (default `1cm`)

Unit vector of the coordinate system in the *x*-direction.

`/pgf/patterns/y=<dimension>` (default `1cm`)

Unit vector of the coordinate system in the *y*-direction.

`/pgf/patterns/parameters=<comma separated list>` (default `empty`)

A list of parameters that are passed to the pattern. This is usually a list of T_EX macros. It is very important that these macros are fully expandable because the values they hold are being used for deduplication in the PDF file.

`/pgf/patterns/defaults=<comma separated list>` (default `empty`)

This list holds default assignments to the parameters passed to the pattern. The default keys can then be found under the `/pgf/pattern keys/` prefix.

`/pgf/patterns/bottom left=<pgfpoint>` (no default)

Bottom left corner of the pattern’s bounding box, e.g. `\pgfqpoint{-1pt}{-1pt}`.

`/pgf/patterns/top right=<pgfpoint>` (no default)

Top right corner of the pattern’s bounding box, e.g. `\pgfqpoint{3.1pt}{3.1pt}`.

`/pgf/patterns/tile size=<pgfpoint>` (no default)

Width and height of a single of the pattern as a PGF point specification, i.e. the *x* coordinate is the width and the *y* coordinate is the height, e.g. `\pgfqpoint{3pt}{3pt}`.

`/pgf/patterns/tile transformation=<pgftransformation>` (default `empty`)

A PGF transformation, e.g. `\pgftransformrotate{30}`.

`/pgf/patterns/code=(code)`

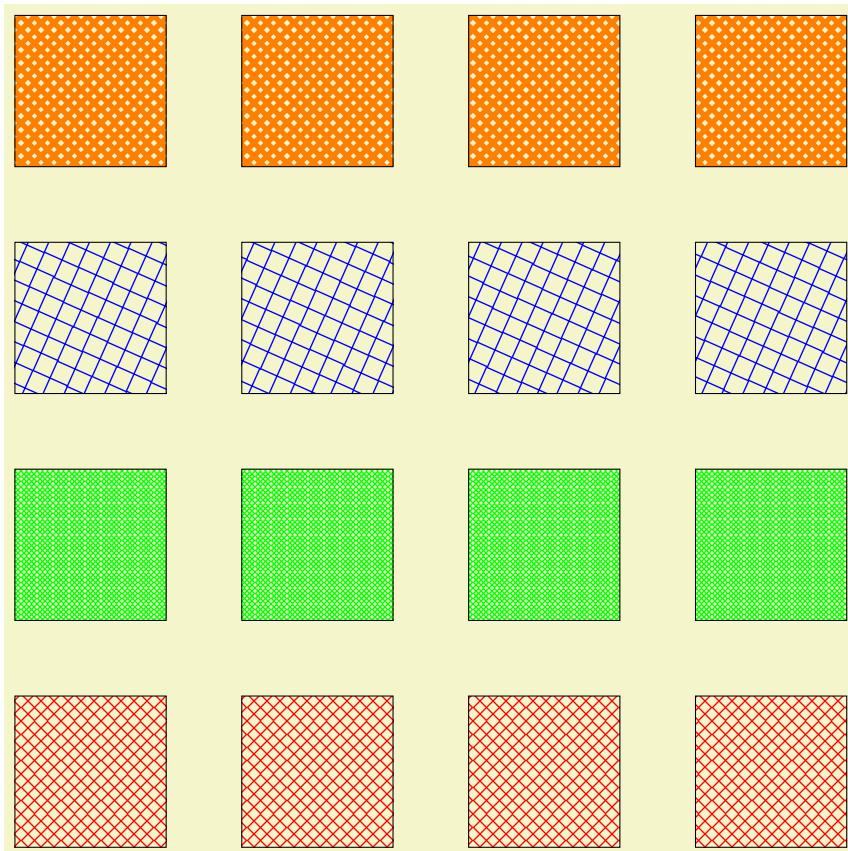
(no default)

The code should be PGF code that can be protocolled. It should not contain any color code or nodes.

`/pgf/patterns/set up code=(code)`

(default `empty`)

This code can be set if parameters have to be preprocessed before the actual pattern code can be run.



```

\usetikzlibrary {patterns.meta}
\pgfdeclarepattern{
  name=hatch,
  parameters={\hatchsize,\hatchangle,\hatchlinewidth},
  bottom left={\pgfpoint{-1pt}{-1pt}},
  top right={\pgfpoint{\hatchsize+.1pt}{\hatchsize+.1pt}},
  tile size={\pgfpoint{\hatchsize}{\hatchsize}},
  tile transformation={\pgftransformrotate{\hatchangle}},
  code={
    \pgfsetlinewidth{\hatchlinewidth}
    \pgfpathmoveto{\pgfpoint{-1pt}{-1pt}}
    \pgfpathlineto{\pgfpoint{\hatchsize+.1pt}{\hatchsize+.1pt}}
    \pgfpathmoveto{\pgfpoint{-1pt}{\hatchsize+.1pt}}
    \pgfpathlineto{\pgfpoint{\hatchsize+.1pt}{-1pt}}
    \pgfusepath{stroke}
  }
}

\tikzset{
  hatch size/.store in=\hatchsize,
  hatch angle/.store in=\hatchangle,
  hatch line width/.store in=\hatchlinewidth,
  hatch size=5pt,
  hatch angle=0pt,
  hatch line width=.5pt,
}

\begin{tikzpicture}
\foreach \r in {1,...,4}
  \draw [pattern=hatch, pattern color=red]
    (\r*3,0) rectangle +(2,2);

\foreach \r in {1,...,4}
  \draw [pattern=hatch, pattern color=green, hatch size=2pt]
    (\r*3,3) rectangle +(2,2);

\foreach \r in {1,...,4}
  \draw [pattern=hatch, pattern color=blue, hatch size=10pt, hatch angle=21]
    (\r*3,6) rectangle +(2,2);

\foreach \r in {1,...,4}
  \draw [pattern=hatch, pattern color=orange, hatch line width=2pt]
    (\r*3,9) rectangle +(2,2);
\end{tikzpicture}

```

There are a couple of predefined PGF patterns which are similar to their normal counterparts.

Pattern Lines

The **Lines** pattern replaces the **horizontal lines**, **vertical lines**, **north east lines**, and **north west lines** patterns. Unfortunately, due to the way the old patterns are constructed, namely that they are not simply related to each other by rotation, the **Lines** pattern cannot be used as a drop-in replacement.

However, the pattern options can be tuned to resemble the other versions closely. The available parameters are:

/pgf/pattern keys/distance (initially 3pt)

Distance between lines.

/pgf/pattern keys/angle (initially 0)

By default the lines are horizontal. The whole pattern is rotated by this angle. The rotation angle is measured in the mathematically positive sense.

/pgf/pattern keys/xshift (initially 0pt)

Shifts the whole pattern in *x*-direction (before applying the rotation).

/pgf/pattern keys/yshift (initially 0pt)

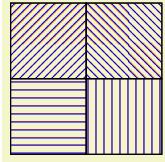
Shifts the whole pattern in *y*-direction (before applying the rotation).

`/pgf/pattern keys/line width`

(initially `\the\pgflinewidth`)

Thickness of the lines.

The following settings can be used to reproduce the other ... `lines` patterns.



```
\usetikzlibrary {patterns.meta}
\begin{tikzpicture}
\draw[pattern={horizontal lines},pattern color=orange]
(0,0) rectangle +(1,1);
\draw[pattern={Lines[yshift=.5pt]},pattern color=blue]
(0,0) rectangle +(1,1);

\draw[pattern={vertical lines},pattern color=orange]
(1,0) rectangle +(1,1);
\draw[pattern={Lines[angle=90,yshift=-.5pt]},pattern color=blue]
(1,0) rectangle +(1,1);

\draw[pattern={north east lines},pattern color=orange]
(0,1) rectangle +(1,1);
\draw[pattern={Lines[angle=45,distance={3pt/sqrt(2)}]},pattern color=blue]
(0,1) rectangle +(1,1);

\draw[pattern={north west lines},pattern color=orange]
(1,1) rectangle +(1,1);
\draw[pattern={Lines[angle=-45,distance={3pt/sqrt(2)}]},pattern color=blue]
(1,1) rectangle +(1,1);
\end{tikzpicture}
```

Pattern Hatch

The `Hatch` pattern replaces the `grid` and `crosshatch` patterns. The `Hatch` pattern without options is a drop-in replacement for the `grid` pattern.

`/pgf/pattern keys/distance`

(initially 3pt)

Distance between crosses.

`/pgf/pattern keys/angle`

(initially 0)

By default the lines are horizontal and vertical. The whole pattern is rotated by this angle. The rotation angle is measured in the mathematically positive sense.

`/pgf/pattern keys/xshift`

(initially 0pt)

Shifts the whole pattern in x -direction (before applying the rotation).

`/pgf/pattern keys/yshift`

(initially 0pt)

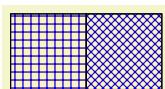
Shifts the whole pattern in y -direction (before applying the rotation).

`/pgf/pattern keys/line width`

(initially `\the\pgflinewidth`)

Thickness of the lines.

The following settings can be used to reproduce the `grid` and `crosshatch` patterns.



```
\usetikzlibrary {patterns.meta}
\begin{tikzpicture}
\draw[pattern={grid},pattern color=orange]
(0,0) rectangle +(1,1);
\draw[pattern={Hatch},pattern color=blue]
(0,0) rectangle +(1,1);

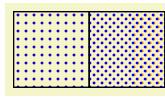
\draw[pattern={crosshatch},pattern color=orange]
(1,0) rectangle +(1,1);
\draw[pattern={Hatch[angle=45,distance={3pt/sqrt(2)}],xshift=.1pt},pattern color=blue]
(1,0) rectangle +(1,1);
\end{tikzpicture}
```

Pattern Dots

The `Dots` pattern replaces the `dots` and `crosshatch dots` patterns. The `Dots` pattern without options is a drop-in replacement for the `dots` pattern.

<code>/pgf/pattern keys/distance</code>	(initially 3pt)
Distance between dots.	
<code>/pgf/pattern keys/angle</code>	(initially 0)
By default the lines are arranged on a regular grid. The whole pattern is rotated by this angle. The rotation angle is measured in the mathematically positive sense.	
<code>/pgf/pattern keys/xshift</code>	(initially 0pt)
Shifts the whole pattern in <i>x</i> -direction (before applying the rotation).	
<code>/pgf/pattern keys/yshift</code>	(initially 0pt)
Shifts the whole pattern in <i>y</i> -direction (before applying the rotation).	
<code>/pgf/pattern keys/radius</code>	(initially 0.5pt)
Radius of the dots.	

The following settings can be used to reproduce the `dots` and `crosshatch dots` patterns.



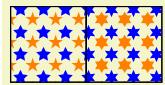
```
\usetikzlibrary {patterns.meta}
\begin{tikzpicture}
  \draw[pattern={dots},pattern color=orange]
    (0,0) rectangle +(1,1);
  \draw[pattern={Dots},pattern color=blue]
    (0,0) rectangle +(1,1);

  \draw[pattern={crosshatch dots},pattern color=orange]
    (1,0) rectangle +(1,1);
  \draw[pattern={Dots[angle=45,distance={3pt/sqrt(2)}]}, 
        pattern color=blue] (1,0) rectangle +(1,1);
\end{tikzpicture}
```

Pattern Stars

The `Stars` pattern replaces the `fivepointed stars` and `sixpointed stars` patterns. However, the stars of the `Stars` pattern are constructed in a fundamentally different fashion, so it can't be used as a drop-in replacement.

<code>/pgf/pattern keys/distance</code>	(initially 3mm)
Distance between stars.	
<code>/pgf/pattern keys/angle</code>	(initially 0)
By default the stars are arranged on a regular grid. The whole pattern is rotated by this angle. The rotation angle is measured in the mathematically positive sense.	
<code>/pgf/pattern keys/xshift</code>	(initially 0pt)
Shifts the whole pattern in <i>x</i> -direction (before applying the rotation).	
<code>/pgf/pattern keys/yshift</code>	(initially 0pt)
Shifts the whole pattern in <i>y</i> -direction (before applying the rotation).	
<code>/pgf/pattern keys/radius</code>	(initially 1mm)
Outer radius of the enclosing circle of the stars.	
<code>/pgf/pattern keys/points</code>	(initially 5)
Number of pointy ends of the stars.	



```
\usetikzlibrary {patterns.meta}
\begin{tikzpicture}
  \draw[pattern={fivepointed stars},pattern color=orange]
    (0,0) rectangle +(1,1);
  \draw[pattern={Stars},pattern color=blue]
    (0,0) rectangle +(1,1);

  \draw[pattern={sixpointed stars},pattern color=orange]
    (1,0) rectangle +(1,1);
  \draw[pattern={Stars[points=6]},pattern color=blue]
    (1,0) rectangle +(1,1);
\end{tikzpicture}
```

\tikzdeclarepattern{\(config)}

A pattern declared with `\pgfdeclarepattern` can only execute PGF code. This command extends the functionality to also allow TikZ code. All the same keys of `\pgfdeclarepattern` are valid, but some of them have been overloaded to give a more natural TikZ syntax.

`/tikz/patterns/bottom left=\(point)` (no default)

Instead of a PGF name point, this key takes a TikZ point, e.g. `(-.1,-.1)`.

`/tikz/patterns/top right=\(point)` (no default)

Instead of a PGF name point, this key takes a TikZ point, e.g. `(3.1,3.1)`.

`/tikz/patterns/tile size=\(point)` (no default)

Instead of a PGF name point, this key takes a TikZ point, e.g. `(3,3)`.

`/tikz/patterns/tile transformation=\(transformation)` (no default)

Instead of a PGF transformation, this key takes a list of keys and value and extracts the resulting transformation from them, e.g. `rotate=30`.

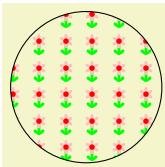
In addition to the overloaded keys, some new keys have been added.

`/tikz/patterns/bounding box=\(point) and \(\point)` (no default)

This is a shorthand to set the bounding box. It will assign the first point to `bottom left` and the second point to `top right`.

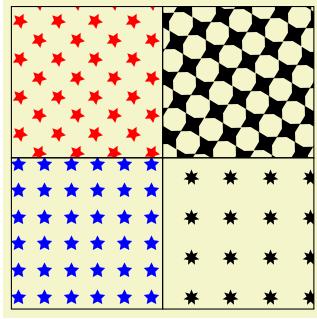
`/tikz/patterns/infer tile bounding box=\(dimension)` (default 0pt)

Instead of specifying the bounding box by hand, you can ask TikZ to infer the size of the bounding box for you. The `\(\dimension)` parameter is padding that is added around the bounding box.



```
\usetikzlibrary {patterns.meta}
\tikzdeclarepattern{
  name=flower,
  type=colored,
  bottom left={(-.1pt,-.1pt)},
  top right={(10.1pt,10.1pt)},
  tile size={(10pt,10pt)},
  code={
    \tikzset{x=1pt,y=1pt}
    \path [draw=green] (5,2.5) -- (5, 7.5);
    \foreach \i in {0,60,...,300}
      \path [fill=pink, shift={(5,7.5)}, rotate=-\i]
        (0,0) .. controls ++(120:4) and ++(60:4) .. (0,0);
    \path [fill=red] (5,7.5) circle [radius=1];
    \foreach \i in {-45,45}
      \path [fill=green, shift={(5,2.5)}, rotate=-\i]
        (0,0) .. controls ++(120:4) and ++(60:4) .. (0,0);
  }
}

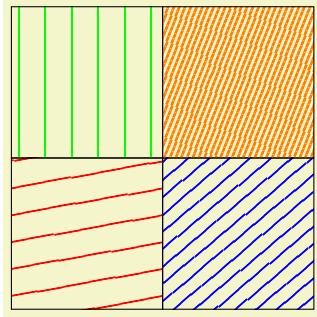
\tikz\draw [pattern=flower] circle [radius=1];
```



```
\usetikzlibrary {patterns.meta}
\tikzdeclarepattern{
  name=mystars,
  type=uncolored,
  bounding box={(-5pt,-5pt) and (5pt,5pt)},
  tile size={(tikztilesize,tikztilesize)},
  parameters={\tikzstarpoints,\tikzstarradius,\tikzstarrotate,\tikztilesize},
  tile transformation={rotate=\tikzstarrotate},
  defaults={
    points/.store in=\tikzstarpoints,points=5,
    radius/.store in=\tikzstarradius,radius=3pt,
    rotate/.store in=\tikzstarrotate,rotate=0,
    tile size/.store in=\tikztilesize,tile size=10pt,
  },
  code=[
    \pgfmathparse{180/\tikzstarpoints}\let\aa=\pgfmathresult
    \fill (90:\tikzstarradius) \foreach \i in {1,...,\tikzstarpoints} {
      -- (90+2*\i*\aa:\tikzstarradius/2) -- (90+2*\i*\aa:\tikzstarradius)
    } -- cycle;
  ]
}

\begin{tikzpicture}
\draw[pattern=mystars,pattern color=blue] (0,0) rectangle +(2,2);
\draw[pattern={mystars[points=7,tile size=15pt]}] (2,0) rectangle +(2,2);
\draw[pattern={mystars[rotate=45]},pattern color=red] (0,2) rectangle +(2,2);
\draw[pattern={mystars[rotate=30,points=4,radius=5pt]}] (2,2) rectangle +(2,2);
\end{tikzpicture}
```

Instead of macros you can also use PGF keys as parameters, if that is what you prefer.



```
\usetikzlibrary {patterns.meta}
\tikzdeclarepattern{
  name=mylines,
  parameters={
    \pgfkeyvalueof{/pgf/pattern keys/size},
    \pgfkeyvalueof{/pgf/pattern keys/angle},
    \pgfkeyvalueof{/pgf/pattern keys/line width},
  },
  bounding box={
    (0,-0.5*\pgfkeyvalueof{/pgf/pattern keys/line width}) and
    (\pgfkeyvalueof{/pgf/pattern keys/size},0.5*\pgfkeyvalueof{/pgf/pattern keys/line width})},
  tile size=(\pgfkeyvalueof{/pgf/pattern keys/size},
    \pgfkeyvalueof{/pgf/pattern keys/size}),
  tile transformation={rotate=\pgfkeyvalueof{/pgf/pattern keys/angle}},
  defaults={
    size/.initial=5pt,
    angle/.initial=45,
    line width/.initial=.4pt,
  },
  code=[
    \draw [line width=\pgfkeyvalueof{/pgf/pattern keys/line width}]
      (0,0) -- (\pgfkeyvalueof{/pgf/pattern keys/size},0);
  ],
}

\begin{tikzpicture}
\draw[pattern={mylines[size=10pt,line width=.8pt,angle=10]},pattern color=red] (0,0) rectangle +(2,2);
\draw[pattern={mylines[size= 5pt,line width=.8pt,angle=40]},pattern color=blue] (2,0) rectangle +(2,2);
\draw[pattern={mylines[size=10pt,line width=.4pt,angle=90]},pattern color=green] (0,2) rectangle +(2,2);
\draw[pattern={mylines[size= 2pt,line width= 1pt,angle=70]},pattern color=orange] (2,2) rectangle +(2,2);
\end{tikzpicture}
```

64 Three Point Perspective Drawing Library

by Max Snippe

TikZ Library `perspective`

```
\usetikzlibrary{perspective} % LATEX and plain TEX  
\usetikzlibrary[perspective] % ConTEXt
```

This library provides tools for perspective drawing with one, two, or three vanishing points.

64.1 Coordinate Systems

Coordinate system `three point perspective`

The `three point perspective` coordinate system is very similar to the `xyz` coordinate system, save that it will display the provided coordinates with a perspective projection.

`/tikz/cs/x=<number>` (no default, initially 0)

The *x* component of the coordinate. Should be given *without* unit.

`/tikz/cs/y=<number>` (no default, initially 0)

Same as `x`.

`/tikz/cs/z=<number>` (no default, initially 0)

Same as `x`.

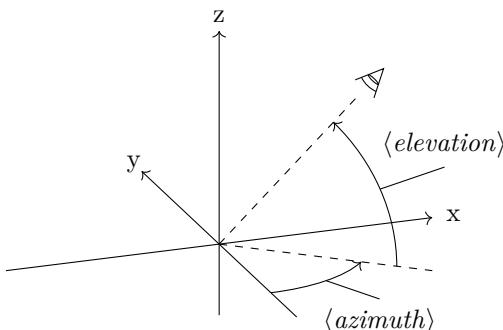
Coordinate system `tpp`

The `tpp` coordinate system is an alias for the `three point perspective` coordinate system.

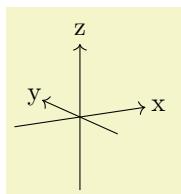
64.2 Setting the view

`/tikz/3d view={<azimuth>}{<elevation>}` (default {-30}{15})

With the `3d view` option, the projection of the 3D coordinates on the 2D page is defined. It is determined by rotating the coordinate system by $-\langle\text{azimuth}\rangle$ around the *z*-axis, and by $\langle\text{elevation}\rangle$ around the (new) *x*-axis, as shown below.



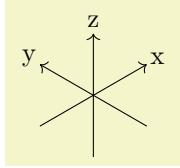
For example, when both $\langle\text{azimuth}\rangle$ and $\langle\text{elevation}\rangle$ are 0° , $+z$ will be pointing upward, and $+x$ will be pointing right. The default is as shown below.



```
\usetikzlibrary {perspective}  
\begin{tikzpicture}[3d view]  
  \draw[->] (-1,0,0) -- (1,0,0) node[pos=1.1]{x};  
  \draw[->] (0,-1,0) -- (0,1,0) node[pos=1.1]{y};  
  \draw[->] (0,0,-1) -- (0,0,1) node[pos=1.1]{z};  
\end{tikzpicture}
```

`/tikz/isometric view` (style, no value)

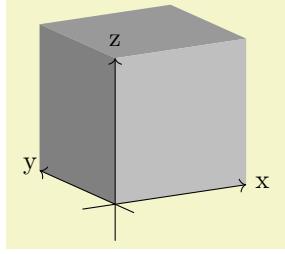
A special kind of 3d view is isometric, which can be set with the `isometric view` style. It simply sets `3d view={-45}{35.26}`. The value for $\langle elevation \rangle$ is determined with $\arctan(1/\sqrt{2})$. In isometric projection the angle between any pair of axes is 120° , as shown below.



```
\usetikzlibrary {perspective}
\begin{tikzpicture}[isometric view]
  \draw[->] (-1,0,0) -- (1,0,0) node[pos=1.1]{x};
  \draw[->] (0,-1,0) -- (0,1,0) node[pos=1.1]{y};
  \draw[->] (0,0,-1) -- (0,0,1) node[pos=1.1]{z};
\end{tikzpicture}
```

64.3 Defining the perspective

In this section, the following example cuboid will be used with various scaling. As a reference, the axes will be shown too, without perspective projection.

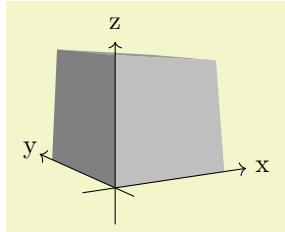


```
\usetikzlibrary {perspective}
\newcommand{\simplecuboid}[3]{%
  \fill[gray!80!white] (tpp cs:x=0,y=0,z=#3)
  -- (tpp cs:x=0,y=#2,z=#3)
  -- (tpp cs:x=#1,y=#2,z=#3)
  -- (tpp cs:x=#1,y=0,z=#3) -- cycle;
  \fill[gray] (tpp cs:x=0,y=0,z=0)
  -- (tpp cs:x=0,y=#2,z=0)
  -- (tpp cs:x=0,y=#2,z=0) -- cycle;
  \fill[gray!50!white] (tpp cs:x=0,y=0,z=0)
  -- (tpp cs:x=0,y=0,z=#3)
  -- (tpp cs:x=#1,y=0,z=#3)
  -- (tpp cs:x=#1,y=0,z=0) -- cycle;}
\newcommand{\simpleaxes}[3]{%
  \draw[->] (-0.5,0,0) -- (#1,0,0) node[pos=1.1]{x};
  \draw[->] (0,-0.5,0) -- (0,#2,0) node[pos=1.1]{y};
  \draw[->] (0,0,-0.5) -- (0,0,#3) node[pos=1.1]{z};}

\begin{tikzpicture}[3d view]
  \simplecuboid{2}{2}{2}
  \simpleaxes{2}{2}{2}
\end{tikzpicture}
```

`/tikz/perspective=⟨vanishing points⟩` (default $p=\{(10,0,0)\}, q=\{(0,10,0)\}, r=\{(0,0,20)\}$)

The ‘strength’ of the perspective can be determined by setting the location of the vanishing points. The default values have a stronger perspective towards x and y than towards z , as shown below.

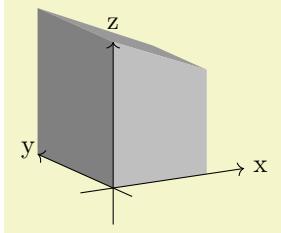


```
\usetikzlibrary {perspective}
\begin{tikzpicture}[3d view,perspective]
  \simplecuboid{2}{2}{2}
  \simpleaxes{2}{2}{2}
\end{tikzpicture}
```

From this example it also shows that the maximum dimensions of the cuboid are no longer 2 by 2 by 2. This is inherent to the perspective projection.

`/tikz/perspective/p=⟨⟨x,y,z⟩⟩` (no default, initially $(0,0,0)$)

The location of the vanishing point that determines the ‘strength’ of the perspective in x -direction can be set with the `p` key.

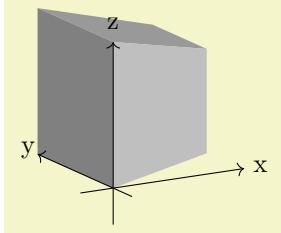


```
\usetikzlibrary {perspective}
\begin{tikzpicture}[
  3d view,
  perspective={%
    p = {(5,0,0)}}
]
\simplecuboid{2}{2}{2}
\simplexes{2}{2}{2}
\end{tikzpicture}
```

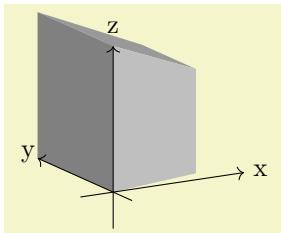
Note also that when only `p` is provided, the perspective in y and z direction is turned off.

To turn off the perspective in x -direction, one must set the x component of `p` to 0 (e.g. `p={(0,a,b)}`, where `a` and `b` can be any number and will be ignored). Or one can provide `q` and `r` and omit `p`.

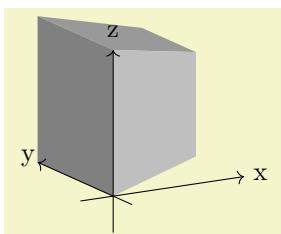
By changing the y and z components of `p`, one can achieve various effects.



```
\usetikzlibrary {perspective}
\begin{tikzpicture}[
  3d view,
  perspective={%
    p = {(5,0,1)}}
]
\simplecuboid{2}{2}{2}
\simplexes{2}{2}{2}
\end{tikzpicture}
```



```
\usetikzlibrary {perspective}
\begin{tikzpicture}[
  3d view,
  perspective={%
    p = {(5,1,0)}}
]
\simplecuboid{2}{2}{2}
\simplexes{2}{2}{2}
\end{tikzpicture}
```

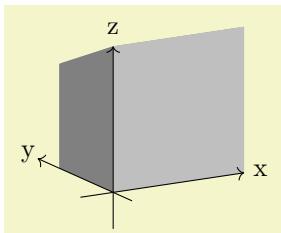


```
\usetikzlibrary {perspective}
\begin{tikzpicture}[
  3d view,
  perspective={%
    p = {(5,1,1)}}
]
\simplecuboid{2}{2}{2}
\simplexes{2}{2}{2}
\end{tikzpicture}
```

`/tikz/perspective/q={⟨x,y,z⟩}`

(no default, initially $(0,0,0)$)

Similar to `p`, but can be turned off by setting its y component to 0.

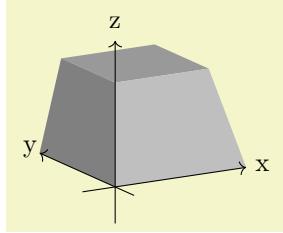


```
\usetikzlibrary {perspective}
\begin{tikzpicture}[
  3d view,
  perspective={%
    q = {(0,5,0)}}
]
\simplecuboid{2}{2}{2}
\simplexes{2}{2}{2}
\end{tikzpicture}
```

`/tikz/perspective/r={⟨x,y,z⟩}`

(no default, initially $(0,0,0)$)

Similar to `p`, but can be turned off by setting its z component to 0.



```
\usetikzlibrary {perspective}
\begin{tikzpicture}[
  3d view,
  perspective={r = {(0,0,5)}}
]
\simplecuboid{2}{2}{2}
\simpleaxes{2}{2}{2}
\end{tikzpicture}
```

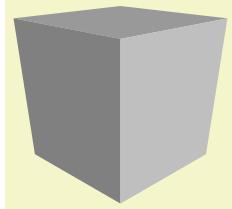
64.4 Shortcomings

Currently a number of things are not working, mostly due to the fact that PGF uses a 2D coordinate system underwater, and perspective projection is a non-linear affine transformation which needs to be aware of all three coordinates. These three coordinates are currently lost when processing a 3D coordinate. The issues include, but possibly are not limited to:

- Keys like `shift`, `xshift`, `yshift` are not working
- Keys like `rotate around x`, `rotate around y`, and `rotate around z` are not working
- Units are not working
- Most keys from the `3d` library are unsupported, e.g. all the `canvas is .. plane` keys.

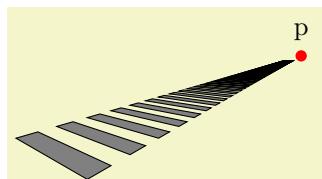
64.5 Examples

An `r` that lies ‘below’ your drawing can mimic a macro effect.



```
\usetikzlibrary {perspective}
\begin{tikzpicture}[
  isometric view,
  perspective={p = {(8,0,0)}, q = {(0,8,0)}, r = {(0,0,-8)}}
]
\simplecuboid{2}{2}{2}
\end{tikzpicture}
```

A peculiar phenomenon inherent to perspective drawing, is that however great your coordinate will become in the direction of the vanishing point, it will never reach it.

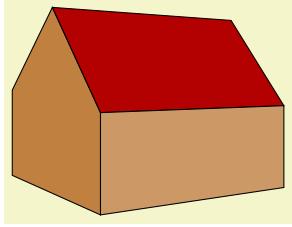


```
\usetikzlibrary {perspective}
\begin{tikzpicture}[
  isometric view,
  perspective={p = {(4,0,0)}, q = {(0,4,0)}}
]
\node[fill=red,circle,inner sep=1.5pt,label=above:p] at (4,0,0){};

\foreach \i in {0,...,100}{
  \filldraw[fill = gray] (tpp cs:x=\i,y=0,z=0)
    -- (tpp cs:x=\i+0.5,y=0,z=0)
    -- (tpp cs:x=\i+0.5,y=2,z=0)
    -- (tpp cs:x=\i,y=2,z=0)
    -- cycle;
}
\end{tikzpicture}
```

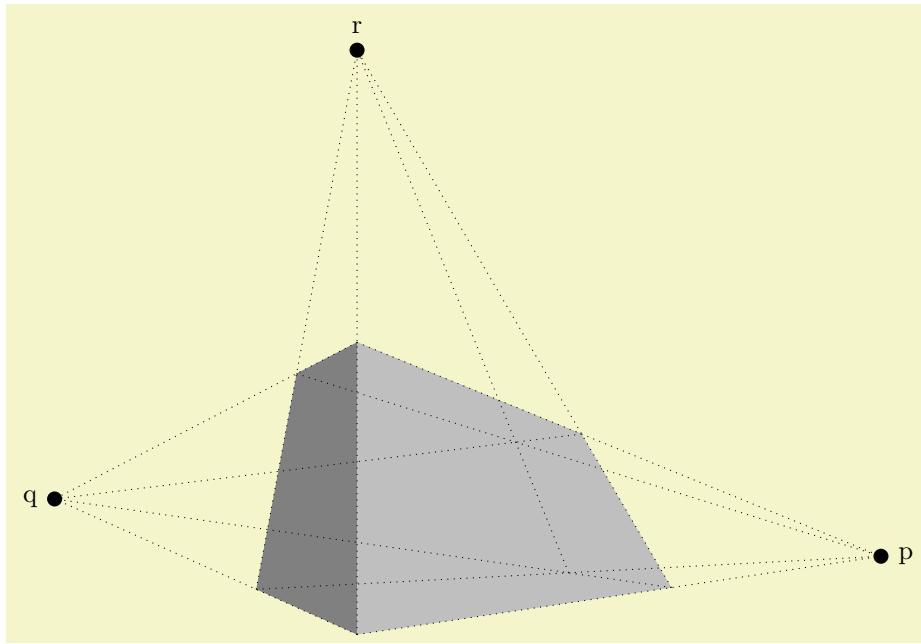
Even for simple examples, the added perspective might add another ‘dimension’ to your drawing. In

this case, two vanishing points give a more intuitive result than three would.



```
\usetikzlibrary {perspective}
\begin{tikzpicture}[
  scale=0.7,
  3d view,
  perspective={
    p = f(20,0,0),
    q = f(0,20,0)}]
\filldraw[fill=brown] (tpp cs:x=0,y=0,z=0)
  -- (tpp cs:x=0,y=4,z=0)
  -- (tpp cs:x=0,y=4,z=2)
  -- (tpp cs:x=0,y=2,z=4)
  -- (tpp cs:x=0,y=0,z=2) -- cycle;
\filldraw[fill=red!70!black] (tpp cs:x=0,y=0,z=2)
  -- (tpp cs:x=5,y=0,z=2)
  -- (tpp cs:x=5,y=2,z=4)
  -- (tpp cs:x=0,y=2,z=4) -- cycle;
\filldraw[fill=brown!80!white] (tpp cs:x=0,y=0,z=0)
  -- (tpp cs:x=0,y=0,z=2)
  -- (tpp cs:x=5,y=0,z=2)
  -- (tpp cs:x=5,y=0,z=0) -- cycle;
\end{tikzpicture}
```

With the vanishing points nearby, the distortion of parallel lines becomes very strong. This might lead to `Dimension too large` errors.



```

\usetikzlibrary {perspective}
\begin{tikzpicture}[
  3d view,
  perspective={%
    p = {(2,0,0)},
    q = {(0,2,0)},
    r = {(0,0,2)}},
  scale=4,
  vanishing point/.style={fill,circle,inner sep=2pt}]
\simplecuboid{3}{1}{2}

\node[vanishing point,label = right:p] (p) at (2,0,0){};
\node[vanishing point,label = left:q] (q) at (0,2,0){};
\node[vanishing point,label = above:r] (r) at (0,0,2){};

\begin{scope}[dotted]
\foreach \y in {0,1}{%
  \foreach \z in {0,2}{%
    \draw (tpp cs:x=0,y=\y,z=\z) -- (p.center);}}
\foreach \x in {0,3}{%
  \foreach \z in {0,2}{%
    \draw (tpp cs:x=\x,y=0,z=\z) -- (q.center);}}
\foreach \x in {0,3}{%
  \foreach \y in {0,1}{%
    \draw (tpp cs:x=\x,y=\y,z=0) -- (r.center);}}
\end{scope}
\end{tikzpicture}

```

65 Petri-Net Drawing Library

TikZ Library `petri`

```
\usetikzlibrary{petri} % LATEX and plain TEX
\usetikzlibrary[petri] % ConTeXt
```

This package provides shapes and styles for drawing Petri nets.

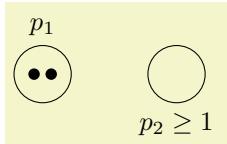
65.1 Places

The package defines a style for drawing places of Petri nets.

/tikz/place

(style, no value)

This style indicates that a node is a place of a Petri net. Usually, the text of the node should be empty since places do not contain any text. You should use the `label` option to add text outside the node like its name or its capacity. You should use the `tokens` options, explained in Section 65.3, to add tokens inside the place.

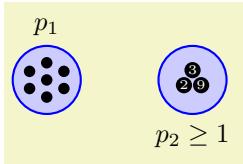


```
\usetikzlibrary {petri,positioning}
\begin{tikzpicture}
  \node[place,label=above:$p_1$,tokens=2] (p1) {};
  \node[place,label=below:$p_2\geq 1$,right=of p1] (p2) {};
\end{tikzpicture}
```

/tikz/every place

(style, no value)

This style is evoked by the style `place`. To change the appearance of places, you can change this style.



```
\usetikzlibrary {petri,positioning}
\begin{tikzpicture}
  [every place/.style={draw=blue,fill=blue!20,thick,minimum size=9mm}]
  \node[place,tokens=7,label=above:$p_1$] (p1) {};
  \node[place,structured tokens={3,2,9},label=below:$p_2\geq 1$,right=of p1] (p2) {};
\end{tikzpicture}
```

65.2 Transitions

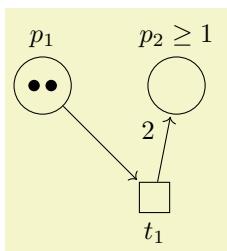
Transitions are also nodes. They should be drawn using the following style:

/tikz/transition

(style, no value)

This style indicates that a node is a transition. As for places, the text of a transition should be empty and the `label` option should be used for adding labels.

To connect a transition to places, you can use the `edge` command as in the following example:



```
\usetikzlibrary {petri,positioning}
\begin{tikzpicture}
  \node[place,tokens=2,label=above:$p_1$] (p1) {};
  \node[place,label=above:$p_2\geq 1$,right=of p1] (p2) {};
  \node[transition,below right=of p1,label=below:$t_1$] (t1) {};
  \edge[pre] (p1) -- (t1);
  \edge[post] (t1) -- (p2);
\end{tikzpicture}
```

/tikz/every transition

(style, no value)

This style is evoked by the style `transition`.

/tikz/pre

(style, no value)

This style can be used with paths leading *from* a transition *to* a place to indicate that the place is in the pre-set of the transition. By default, this style is `<-, shorten <=1pt`, but feel free to redefine it.

/tikz/post

(style, no value)

This style is also used with paths leading *from* a transition *to* a place, but this time the place is in the post-set of the transition. Again, feel free to redefine it.

/tikz/pre and post

(style, no value)

This style is to be used to indicate that a place is both in the pre- and post-set of a transition.

65.3 Tokens

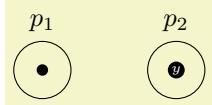
Interestingly, the most complicated aspect of drawing Petri nets in TikZ turns out to be the placement of tokens.

Let us start with a single token. They are also nodes and there is a simple style for typesetting them.

/tikz/token

(style, no value)

This style indicates that a node is a token. By default, this causes the node to be a small black circle. Unlike places and transitions, it *does* make sense to provide text for the token node. Such text will be typeset in a tiny font and in white on black (naturally, you can easily change this by setting the style `every token`).



```
\usetikzlibrary {petri,positioning}
\begin{tikzpicture}
  \node[place,label=above:$p_1$] (p1) {};
  \node[token] at (p1) {\tiny y};

  \node[place,label=above:$p_2$,right=of p1] (p2) {};
  \node[token] at (p2) {\tiny y};
\end{tikzpicture}
```

/tikz/every token

(style, no value)

Change this style to change the appearance of tokens.

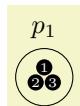
In the above example, it is bothersome that we need an extra command for the token node. Worse, when we have *two* tokens on a node, it is difficult to place both nodes inside the node without overlap.

The Petri library offers a solution to this problem: The `children are tokens` style.

/tikz/children are tokens

(style, no value)

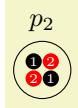
The idea behind this style is to use trees mechanism for placing tokens. Every token lying on a place is treated as a child of the node. Normally this would have the effect that the tokens are placed below the place and they would be connected to the place by an edge. The `children are tokens` style, however, redefines the growth function of trees such that it places the children next to each other inside (or, rather, on top) of the place node. Additionally, the edge from the parent node is not drawn.



```
\usetikzlibrary {petri}
\begin{tikzpicture}
  \node[place,label=above:$p_1$] {
    \begin{tikzpicture}[baseline]
      \node[token] {\tiny 1};
      \node[token] {\tiny 2};
      \node[token] {\tiny 3};
    \end{tikzpicture}
  };
\end{tikzpicture}
```

In detail, what happens is the following: Tree growth functions tell TikZ where it should place the children of nodes. These functions get passed the number of children that a node has and the number of the child that should be placed. The special tree growth function for tokens has a special mapping for each possible number of children up to nine children. This mapping decides for each child where it should be placed on top of the place. For example, a single child is placed directly on top of the

place. Two children are placed next to each other, separated by the `token distance`. Three children are placed in a triangle whose side lengths are `token distance`; and so on up to nine tokens. If you wish to place more than nice tokens on a place, you will have to write your own placement code.

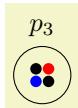


```
\usetikzlibrary {petri}
\begin{tikzpicture}
  \node [place,label=above:$p_2$] {}
    [children are tokens]
    child {node [token] {1}}
    child {node [token,fill=red] {2}}
    child {node [token,fill=red] {2}}
    child {node [token] {1}};
\end{tikzpicture}
```

`/tikz/token distance=<distance>`

(no default)

This specifies the distance between the centers of the tokens in the arrangements of the option `children are tokens`.



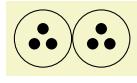
```
\usetikzlibrary {petri}
\begin{tikzpicture}
  \node [place,label=above:$p_3$] {}
    [children are tokens,token distance=1.1ex]
    child {node [token] {}}
    child {node [token,red] {}}
    child {node [token,blue] {}}
    child {node [token] {}};
\end{tikzpicture}
```

The `children are tokens` option gives you a lot of flexibility, but it is a bit cumbersome to use. For this reason there are some options that help in standard situations. They all use `children are tokens` internally, so any change to, say, the `every token` style will affect how these options depict tokens.

`/tikz/tokens=<number>`

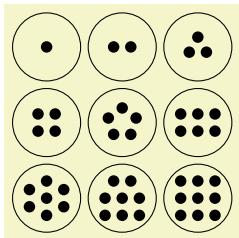
(no default)

This option is given to a `place` node, not to a `token` node. The effect of this option is to add `<number>` many child nodes to the place, each having the style `token`. Thus, the following two pieces of codes have the same effect:



```
\usetikzlibrary {petri}
\tikz
  \node [place] {}
    [children are tokens]
    child {node [token] {}}
    child {node [token] {}}
    child {node [token] {}};
\tikz
  \node [place,tokens=3] {};
```

It is legal to say `tokens=0`, no tokens are drawn in this case. This option does not handle ten or more tokens correctly. If you need this many tokens, you will have to program your own code.



```
\usetikzlibrary {petri}
\begin{tikzpicture}[every place/.style={minimum size=9mm}]
  \foreach \x/\y/\tokennumber in {0/2/1,1/2/2,2/2/3,
                                 0/1/4,1/1/5,2/1/6,
                                 0/0/7,1/0/8,2/0/9}
    \node [place,tokens=\tokennumber] at (\x,\y) {};
\end{tikzpicture}
```

`/tikz/colored tokens=<color list>`

(no default)

This option, which must also be given when a place node is being created, gets a list of colors as parameter. It will then add as many tokens to the place as there are colors in this list, each filled correspondingly.



```
\usetikzlibrary {petri}
\begin{tikzpicture}
\node[place,colored tokens={black,black,red,blue}] {};
\end{tikzpicture}
```

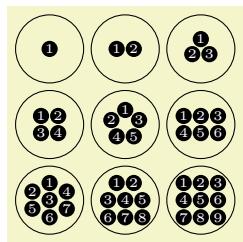
`/tikz/structured tokens=(token texts)`

(no default)

This option, which must again be passed to a place, gets a list of texts for tokens. For each text, a new token will be added to the place.



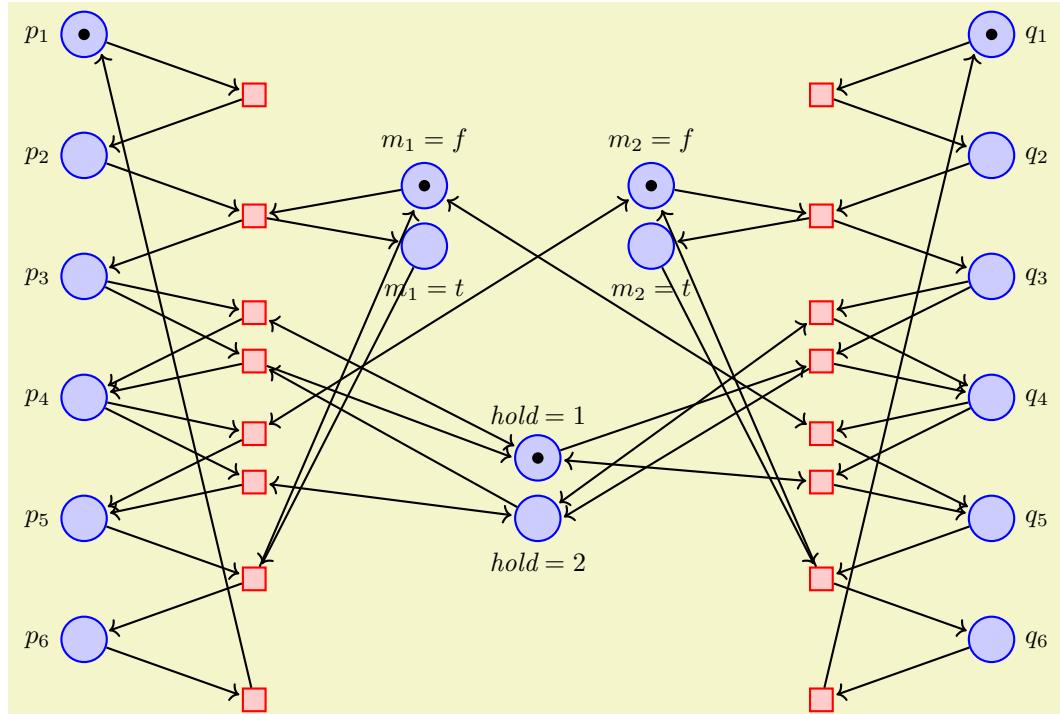
```
\usetikzlibrary {petri}
\begin{tikzpicture}
\node[place,structured tokens={$x$,$y$,$z$}] {};
\end{tikzpicture}
```



```
\usetikzlibrary {petri}
\begin{tikzpicture}[every place/.style={minimum size=9mm}]
\foreach \x/\y/\tokennumber in {0/2/1,1/2/2,2/2/3,
                                0/1/4,1/1/5,2/1/6,
                                0/0/7,1/0/8,2/0/9}
    \node [place,structured tokens={1,...,\tokennumber}] at (\x,\y) {};
\end{tikzpicture}
```

If you use lots of structured tokens, consider redefining the `every token` style so that the tokens are larger.

65.4 Examples



```

\usetikzlibrary {petri}
\begin{tikzpicture}[yscale=-1.6,xscale=1.5,thick,
  every transition/.style={draw=red,fill=red!20,minimum size=3mm},
  every place/.style={draw=blue,fill=blue!20,minimum size=6mm}]

\foreach \i in {1,\dots,6} {
  \node[place,label=left:$p_{\i}$] (p\i) at (0,\i) {};
  \node[place,label=right:$q_{\i}$] (q\i) at (8,\i) {};
}
\foreach \name/\var/\vala/\valb/\height/\x in
  {m1/m_1/f/t/2.25/3,m2/m_2/f/t/2.25/5,h/\mathit{hold}/1/2/4.5/4} {
  \node[place,label=above:$\var = \vala$] (\name\vala) at (\x,\height) {};
  \node[place,yshift=-8mm,label=below:$\var = \valb$] (\name\valb) at (\x,\height) {};
}
\node[token] at (p1) {};
\node[token] at (q1) {};
\node[token] at (m1f) {};
\node[token] at (m2f) {};
\node[token] at (h1) {};

\node[transition] at (1.5,1.5) {} edge [pre] (p1) edge [post] (p2);
\node[transition] at (1.5,2.5) {} edge [pre] (p2) edge [pre] (m1f);
edge [post] (p3) edge [post] (m1t);
\node[transition] at (1.5,3.3) {} edge [pre] (p3) edge [post] (p4);
edge [pre and post] (h1);
\node[transition] at (1.5,3.7) {} edge [pre] (p3) edge [pre] (h2);
edge [post] (p4) edge [post] (h1.west);
\node[transition] at (1.5,4.3) {} edge [pre] (p4) edge [post] (p5);
edge [pre and post] (m2f);
\node[transition] at (1.5,4.7) {} edge [pre] (p4) edge [post] (p5);
edge [pre and post] (h2);
\node[transition] at (1.5,5.5) {} edge [pre] (p5) edge [pre] (m1t);
edge [post] (p6) edge [post] (m1f);
\node[transition] at (1.5,6.5) {} edge [pre] (p6) edge [post] (p1.south east);
\node[transition] at (6.5,1.5) {} edge [pre] (q1) edge [post] (q2);
\node[transition] at (6.5,2.5) {} edge [pre] (q2) edge [pre] (m2f);
edge [post] (q3) edge [post] (m2t);
\node[transition] at (6.5,3.3) {} edge [pre] (q3) edge [post] (q4);
edge [pre and post] (h2);
\node[transition] at (6.5,3.7) {} edge [pre] (q3) edge [pre] (h1);
edge [post] (q4) edge [post] (h2.east);
\node[transition] at (6.5,4.3) {} edge [pre] (q4) edge [post] (q5);
edge [pre and post] (m1f);
\node[transition] at (6.5,4.7) {} edge [pre] (q4) edge [post] (q5);
edge [pre and post] (h1);
\node[transition] at (6.5,5.5) {} edge [pre] (q5) edge [pre] (m2t);
edge [post] (q6) edge [post] (m2f);
\node[transition] at (6.5,6.5) {} edge [pre] (q6) edge [post] (q1.south west);
\end{tikzpicture}

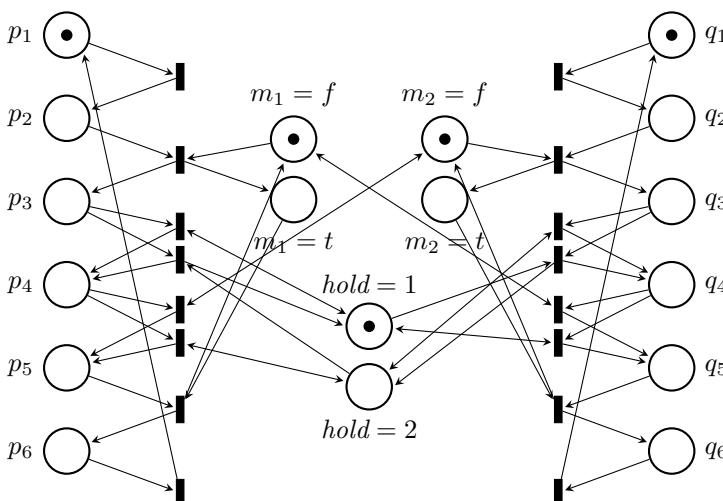
```

Here is the same net once more, but with these styles changes:

```

\begin{tikzpicture}[yscale=-1.1,thin,>=stealth,
  every transition/.style={fill,minimum width=1mm,minimum height=3.5mm},
  every place/.style={draw,thick,minimum size=6mm}]

```



66 Plot Handler Library

TikZ Library `plothandlers`

```
\usepgflibrary{plothandlers} % LATEX and plain TEX and pure pgf
\usepgflibrary[plothandlers] % ConTEXt and pure pgf
\usetikzlibrary{plothandlers} % LATEX and plain TEX when using TikZ
\usetikzlibrary[plothandlers] % ConTEXt when using TikZ
```

This library packages defines additional plot handlers, see Section ?? for an introduction to plot handlers. The additional handlers are described in the following.

This library is loaded automatically by TikZ.

66.1 Curve Plot Handlers

`\pgfplothandlercurveto`

This handler will issue a `\pgfpathcurveto` command for each point of the plot, *except* possibly for the first. As for the line-to handler, what happens with the first point can be specified using `\pgfsetmovetofirstplotpoint` or `\pgfsetlinetofirstplotpoint`.

Obviously, the `\pgfpathcurveto` command needs, in addition to the points on the path, some control points. These are generated automatically using a somewhat “dumb” algorithm: Suppose you have three points x , y , and z on the curve such that y is between x and z :

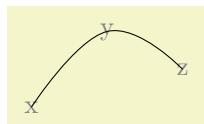


```
\begin{tikzpicture}
\draw[gray] (0,0) node {x} (1,1) node {y} (2,.5) node {z};
\pgfplothandlercurveto
\pgfplotstreamstart
\pgfplotstreampoint{\pgfpoint{0cm}{0cm}}
\pgfplotstreampoint{\pgfpoint{1cm}{1cm}}
\pgfplotstreampoint{\pgfpoint{2cm}{.5cm}}
\pgfplotstreamend
\pgfusepath{stroke}
\end{tikzpicture}
```

In order to determine the control points of the curve at the point y , the handler computes the vector $z - x$ and scales it by the tension factor (see below). Let us call the resulting vector s . Then $y + s$ and $y - s$ will be the control points around y . The first control point at the beginning of the curve will be the beginning itself, once more; likewise the last control point is the end itself.

`\pgfsetpottension{<value>}`

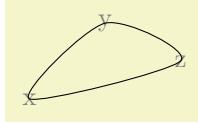
Sets the factor used by the curve plot handlers to determine the distance of the control points from the points they control. The higher the curvature of the curve points, the higher this value should be. A value of 1 will cause four points at quarter positions of a circle to be connected using a circle. The default is 0.5.



```
\begin{tikzpicture}
\draw[gray] (0,0) node {x} (1,1) node {y} (2,.5) node {z};
\pgfsetpottension{0.75}
\pgfplothandlercurveto
\pgfplotstreamstart
\pgfplotstreampoint{\pgfpoint{0cm}{0cm}}
\pgfplotstreampoint{\pgfpoint{1cm}{1cm}}
\pgfplotstreampoint{\pgfpoint{2cm}{0.5cm}}
\pgfplotstreamend
\pgfusepath{stroke}
\end{tikzpicture}
```

`\pgfplothandlerclosedcurve`

This handler works like the curve-to plot handler, only it will add a new part to the current path that is a closed curve through the plot points.



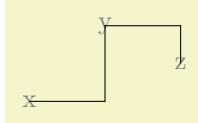
```
\begin{tikzpicture}
\draw[gray] (0,0) node {x} (1,1) node {y} (2,.5) node {z};
\pgfplotshandlerclosedcurve
\pgfplotstreamstart
\pgfplotstreampoint{\pgfpoint{0cm}{0cm}}
\pgfplotstreampoint{\pgfpoint{1cm}{1cm}}
\pgfplotstreampoint{\pgfpoint{2cm}{0.5cm}}
\pgfplotstreamend
\pgfusepath{stroke}
\end{tikzpicture}
```

66.2 Constant Plot Handlers

There are several plot handlers which produce piecewise constant interpolations between successive points:

`\pgfplotshandlerconstantlineto`

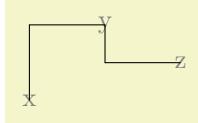
This handler works like the line-to plot handler, only it will produce a connected, piecewise constant path to connect the points.



```
\begin{tikzpicture}
\draw[gray] (0,0) node {x} (1,1) node {y} (2,.5) node {z};
\pgfplotshandlerconstantlineto
\pgfplotstreamstart
\pgfplotstreampoint{\pgfpoint{0cm}{0cm}}
\pgfplotstreampoint{\pgfpoint{1cm}{1cm}}
\pgfplotstreampoint{\pgfpoint{2cm}{0.5cm}}
\pgfplotstreamend
\pgfusepath{stroke}
\end{tikzpicture}
```

`\pgfplotshandlerconstantlinetomarkright`

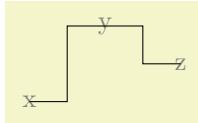
A variant of `\pgfplotshandlerconstantlineto` which places its mark on the right line ends.



```
\begin{tikzpicture}
\draw[gray] (0,0) node {x} (1,1) node {y} (2,.5) node {z};
\pgfplotshandlerconstantlinetomarkright
\pgfplotstreamstart
\pgfplotstreampoint{\pgfpoint{0cm}{0cm}}
\pgfplotstreampoint{\pgfpoint{1cm}{1cm}}
\pgfplotstreampoint{\pgfpoint{2cm}{0.5cm}}
\pgfplotstreamend
\pgfusepath{stroke}
\end{tikzpicture}
```

`\pgfplotshandlerconstantlinetomarkmid`

A variant of `\pgfplotshandlerconstantlineto` which places its mark on the center of the line.

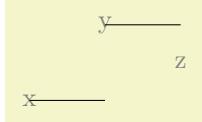


```
\begin{tikzpicture}
\draw[gray] (0,0) node {x} (1,1) node {y} (2,.5) node {z};
\pgfplotshandlerconstantlinetomarkmid
\pgfplotstreamstart
\pgfplotstreampoint{\pgfpoint{0cm}{0cm}}
\pgfplotstreampoint{\pgfpoint{1cm}{1cm}}
\pgfplotstreampoint{\pgfpoint{2cm}{0.5cm}}
\pgfplotstreamend
\pgfusepath{stroke}
\end{tikzpicture}
```

The plot handler always connects two data points by a horizontal line starting from the previous data points, followed by a vertical line in the middle between the two data points, followed by a horizontal line between the middle and the current data point. This results in a symmetric constant plot handler for constant mesh width.

`\pgfplotshandlerjumpmarkleft`

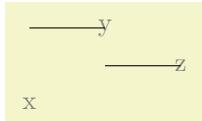
This handler works like the line-to plot handler, only it will produce a non-connected, piecewise constant path to connect the points. If there are any plot marks, they will be placed on the left open pieces.



```
\begin{tikzpicture}
\draw[gray] (0,0) node {x} (1,1) node {y} (2,.5) node {z};
\pgfplotshandlerjumpmarkleft
\pgfplotstreamstart
\pgfplotstreampoint{\pgfpoint{0cm}{0cm}}
\pgfplotstreampoint{\pgfpoint{1cm}{1cm}}
\pgfplotstreampoint{\pgfpoint{2cm}{0.5cm}}
\pgfplotstreamend
\pgfusepath{stroke}
\end{tikzpicture}
```

\pgfplotshandlerjumpmarkright

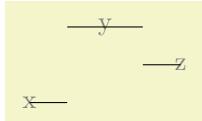
This handler works like the line-to plot handler, only it will produce a non-connected, piecewise constant path to connect the points. If there are any plot marks, they will be placed on the right open pieces.



```
\begin{tikzpicture}
\draw[gray] (0,0) node {x} (1,1) node {y} (2,.5) node {z};
\pgfplotshandlerjumpmarkright
\pgfplotstreamstart
\pgfplotstreampoint{\pgfpoint{0cm}{0cm}}
\pgfplotstreampoint{\pgfpoint{1cm}{1cm}}
\pgfplotstreampoint{\pgfpoint{2cm}{0.5cm}}
\pgfplotstreamend
\pgfusepath{stroke}
\end{tikzpicture}
```

\pgfplotshandlerjumpmarkmid

This handler works like the \pgfplotshandlerconstantlinetomarkmid, but it will produce a non-connected, piecewise constant path to connect the points. If there are any plot marks, they will be placed in the center of the horizontal line segment..



```
\begin{tikzpicture}
\draw[gray] (0,0) node {x} (1,1) node {y} (2,.5) node {z};
\pgfplotshandlerjumpmarkmid
\pgfplotstreamstart
\pgfplotstreampoint{\pgfpoint{0cm}{0cm}}
\pgfplotstreampoint{\pgfpoint{1cm}{1cm}}
\pgfplotstreampoint{\pgfpoint{2cm}{0.5cm}}
\pgfplotstreamend
\pgfusepath{stroke}
\end{tikzpicture}
```

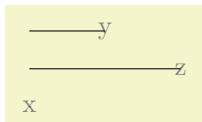
See \pgfplotshandlerconstantlinetomarkmid for details.

66.3 Comb Plot Handlers

There are three “comb” plot handlers. Their name stems from the fact that the plots they produce look like “combs” (more or less).

\pgfplotshandlerxcomb

This handler converts each point in the plot stream into a line from the *y*-axis to the point’s coordinate, resulting in a “horizontal comb”.



```
\begin{tikzpicture}
\draw[gray] (0,0) node {x} (1,1) node {y} (2,.5) node {z};
\pgfplotshandlerxcomb
\pgfplotstreamstart
\pgfplotstreampoint{\pgfpoint{0cm}{0cm}}
\pgfplotstreampoint{\pgfpoint{1cm}{1cm}}
\pgfplotstreampoint{\pgfpoint{2cm}{0.5cm}}
\pgfplotstreamend
\pgfusepath{stroke}
\end{tikzpicture}
```

\pgfplotxhandlercomb

This handler converts each point in the plot stream into a line from the x -axis to the point's coordinate, resulting in a “vertical comb”.

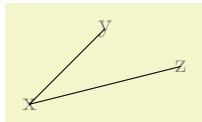
This handler is useful for creating “bar diagrams”.



```
\begin{tikzpicture}
\draw[gray] (0,0) node {x} (1,1) node {y} (2,.5) node {z};
\pgfplotxhandlercomb
\pgfplotstreamstart
\pgfplotstreampoint{\pgfpoint{0cm}{0cm}}
\pgfplotstreampoint{\pgfpoint{1cm}{1cm}}
\pgfplotstreampoint{\pgfpoint{2cm}{0.5cm}}
\pgfplotstreamend
\pgfusepath{stroke}
\end{tikzpicture}
```

\pgfplotxhandlerpolarcomb

This handler converts each point in the plot stream into a line from the origin to the point's coordinate.



```
\begin{tikzpicture}
\draw[gray] (0,0) node {x} (1,1) node {y} (2,.5) node {z};
\pgfplotxhandlerpolarcomb
\pgfplotstreamstart
\pgfplotstreampoint{\pgfpoint{0cm}{0cm}}
\pgfplotstreampoint{\pgfpoint{1cm}{1cm}}
\pgfplotstreampoint{\pgfpoint{2cm}{0.5cm}}
\pgfplotstreamend
\pgfusepath{stroke}
\end{tikzpicture}
```

PGF bar or comb plots usually draw something from zero to the current plot's coordinate. The “zero” offset can be changed using an input stream which returns the desired offset successively for each processed coordinate.

There are two such streams, which can be configured independently. The first one returns “zeros” for coordinate x , the second one returns “zeros” for coordinate y . They are used as follows.

```
\pgfplotxzerolevelstreamstart
\pgfplotxzerolevelstreamnext % assigns \pgf@x
\pgfplotxzerolevelstreamnext
\pgfplotxzerolevelstreamnext
\pgfplotxzerolevelstreamend
```

```
\pgfplotyzerolevelstreamstart
\pgfplotyzerolevelstreamnext % assigns \pgf@x
\pgfplotyzerolevelstreamend
```

Different zero level streams can be implemented by overwriting these macros.

\pgfplotxzerolevelstreamconstant{<dimension>}

This zero level stream always returns $\{<\text{dimension}>\}$ instead of $x = 0\text{pt}$.

It is used for `xcomb` and `xbar`.

\pgfplotyzerolevelstreamconstant{<dimension>}

This zero level stream always returns $\{<\text{dimension}>\}$ instead of $y = 0\text{pt}$.

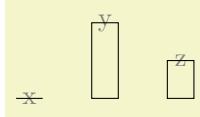
It is used for `ycomb` and `ybar`.

66.4 Bar Plot Handlers

While comb plot handlers produce a line-to operation to generate combs, bar plot handlers employ rectangular shapes, allowing filled bars (or pattern bars).

\pgfplotshandlerybar

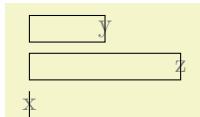
This handler converts each point in the plot stream into a rectangle from the x -axis to the point's coordinate. The rectangle is placed centered at the x -axis.



```
\begin{tikzpicture}
\draw[gray] (0,0) node {x} (1,1) node {y} (2,.5) node {z};
\pgfplotshandlerybar
\pgfplotstreamstart
\pgfplotstreampoint{\pgfpoint{0cm}{0cm}}
\pgfplotstreampoint{\pgfpoint{1cm}{1cm}}
\pgfplotstreampoint{\pgfpoint{2cm}{0.5cm}}
\pgfplotstreamend
\pgfusepath{stroke}
\end{tikzpicture}
```

\pgfplotshandlerxbar

This handler converts each point in the plot stream into a rectangle from the y -axis to the point's coordinate. The rectangle is placed centered at the y -axis.



```
\begin{tikzpicture}
\draw[gray] (0,0) node {x} (1,1) node {y} (2,.5) node {z};
\pgfplotshandlerxbar
\pgfplotstreamstart
\pgfplotstreampoint{\pgfpoint{0cm}{0cm}}
\pgfplotstreampoint{\pgfpoint{1cm}{1cm}}
\pgfplotstreampoint{\pgfpoint{2cm}{0.5cm}}
\pgfplotstreamend
\pgfusepath{stroke}
\end{tikzpicture}
```

/pgf/bar width={*dimension*}

(no default, initially 10pt)

alias **/tikz/bar width**

Sets the width of **\pgfplotshandlerxbar** and **\pgfplotshandlerybar** to **{*dimension*}**. The argument **{*dimension*}** will be evaluated using the math parser.

/pgf/bar shift={*dimension*}

(no default, initially 0pt)

alias **/tikz/bar shift**

Sets a shift used by **\pgfplotshandlerxbar** and **\pgfplotshandlerybar** to **{*dimension*}**. It has the same effect as **xshift**, but it applies only to those bar plots. The argument **{*dimension*}** will be evaluated using the math parser.

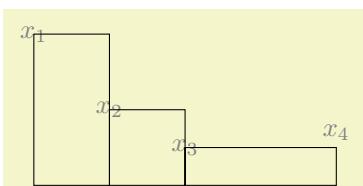
\pgfplotbarwidth

Expands to the value of **/pgf/bar width**.

\pgfplotshandlerybarinterval

This handler is a variant of **\pgfplotshandlerybar** which works with intervals instead of points.

Bars are drawn between successive input coordinates and the width is determined relatively to the interval length.



```
\begin{tikzpicture}
\draw[gray] (0,2) node {$x_1$} (1,1) node {$x_2$} (2,.5) node {$x_3$} (4,0.7) node {$x_4$};
\pgfplotshandlerbarinterval
\pgfplotstreamstart
\pgfplotstreampoint{\pgfpoint{0cm}{2cm}}
\pgfplotstreampoint{\pgfpoint{1cm}{1cm}}
\pgfplotstreampoint{\pgfpoint{2cm}{0.5cm}}
\pgfplotstreampoint{\pgfpoint{4cm}{0.7cm}}
\pgfplotstreamend
\pgfusepath{stroke}
\end{tikzpicture}
```

In more detail, if (x_i, y_i) and (x_{i+1}, y_{i+1}) denote successive input coordinates, the bar will be placed above the interval $[x_i, x_{i+1}]$, centered at

$$x_i + \langle \text{bar interval shift} \rangle \cdot (x_{i+1} - x_i)$$

with width

$$\langle \text{bar interval width} \rangle \cdot (x_{i+1} - x_i).$$

Here, $\langle \text{bar interval shift} \rangle$ and $\langle \text{bar interval width} \rangle$ denote the current values of the associated options.

If you have $N + 1$ input points, you will get N bars (one for each interval). The y value of the last point will be ignored.

\pgfplotshandlerxbarinterval

As `\pgfplotshandlerbarinterval`, this handler provides bar plots with relative bar sizes and offsets, one bar for each y coordinate interval.

`/pgf/bar interval shift={⟨factor⟩}` (no default, initially 0.5)
alias `/tikz/bar interval shift`

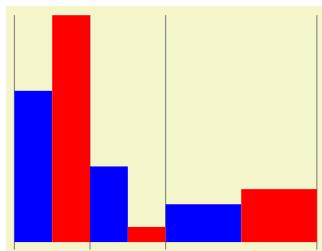
Sets the *relative* shift of `\pgfplotshandlerxbarinterval` and `\pgfplotshandlerbarinterval` to $\langle \text{factor} \rangle$. As `/pgf/bar interval width`, the argument is relative to the interval length of the input coordinates.

The argument $\{⟨scale⟩\}$ will be evaluated using the math parser.

`/pgf/bar interval width={⟨scale⟩}` (no default, initially 1)
alias `/tikz/bar interval width`

Sets the *relative* width of `\pgfplotshandlerxbarinterval` and `\pgfplotshandlerbarinterval` to $\{⟨scale⟩\}$. The argument is relative to $(x_{i+1} - x_i)$ for y bar plots and relative to $(y_{i+1} - y_i)$ for x bar plots.

The argument $\{⟨scale⟩\}$ will be evaluated using the math parser.



```
\begin{tikzpicture}[bar interval width=0.5]
\draw[gray]
(0,3) -- (0,-0.1)
(1,3) -- (1,-0.1)
(2,3) -- (2,-0.1)
(4,3) -- (4,-0.1);
\pgfplotshandlerbarinterval
\begin{scope}[bar interval shift=0.25,fill=blue]
\pgfplotstreamstart
\pgfplotstreampoint{\pgfpoint{0cm}{2cm}}
\pgfplotstreampoint{\pgfpoint{1cm}{1cm}}
\pgfplotstreampoint{\pgfpoint{2cm}{0.5cm}}
\pgfplotstreampoint{\pgfpoint{4cm}{0.7cm}}
\pgfplotstreamend
\pgfusepath{fill}
\end{scope}
\begin{scope}[bar interval shift=0.75,fill=red]
\pgfplotstreamstart
\pgfplotstreampoint{\pgfpoint{0cm}{3cm}}
\pgfplotstreampoint{\pgfpoint{1cm}{0.2cm}}
\pgfplotstreampoint{\pgfpoint{2cm}{0.7cm}}
\pgfplotstreampoint{\pgfpoint{4cm}{0.2cm}}
\pgfplotstreamend
\pgfusepath{fill}
\end{scope}
\end{tikzpicture}
```

Please note that bars are always centered, so we have to use shifts 0.25 and 0.75 instead of 0 and 0.5.

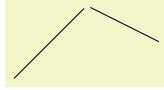
66.5 Gapped Plot Handlers

\pgfplothandlergaplineto

This handler will connect the points of the plots by straight line segments. However, at the start and the end of the lines there will be a small gap, given by the following key:

`/pgf/gap around stream point=<dimension>` (no default, initially 1.5pt)

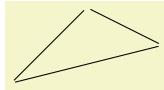
The `<dimension>` by which the lines between consecutive stream points are shortened at the beginning and end.



```
\begin{tikzpicture}
\pgfplothandlergaplineto
\pgfplotstreamstart
\pgfplotstreampoint{\pgfpoint{0cm}{0cm}}
\pgfplotstreampoint{\pgfpoint{1cm}{1cm}}
\pgfplotstreampoint{\pgfpoint{2cm}{0.5cm}}
\pgfplotstreamend
\pgfusepath{stroke}
\end{tikzpicture}
```

\pgfplothandlergapcycle

Works like `\pgfplothandlergaplineto`, but the last point is connected to the first in the same fashion:



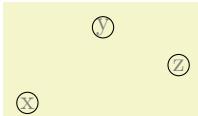
```
\begin{tikzpicture}
\pgfplothandlergapcycle
\pgfplotstreamstart
\pgfplotstreampoint{\pgfpoint{0cm}{0cm}}
\pgfplotstreampoint{\pgfpoint{1cm}{1cm}}
\pgfplotstreampoint{\pgfpoint{2cm}{0.5cm}}
\pgfplotstreamend
\pgfusepath{stroke}
\end{tikzpicture}
```

66.6 Mark Plot Handler

\pgfplothandlermark{<mark code>}

This command will execute the `<mark code>` for some points of the plot, but each time the coordinate transformation matrix will be set up such that the origin is at the position of the point to be plotted. This way, if the `<mark code>` draws a little circle around the origin, little circles will be drawn at some point of the plot.

By default, a mark is drawn at all points of the plot. However, two parameters r and p influence this. First, only every r th mark is drawn. Second, the first mark drawn is the p th. These parameters can be influenced using the commands below.



```
\begin{tikzpicture}
\draw[gray] (0,0) node {x} (1,1) node {y} (2,.5) node {z};
\pgfplothandlermark{\pgfpathcircle{\pgfpointorigin}{4pt}\pgfusepath{stroke}}
\pgfplotstreamstart
\pgfplotstreampoint{\pgfpoint{0cm}{0cm}}
\pgfplotstreampoint{\pgfpoint{1cm}{1cm}}
\pgfplotstreampoint{\pgfpoint{2cm}{0.5cm}}
\pgfplotstreamend
\pgfusepath{stroke}
\end{tikzpicture}
```

Typically, the `<code>` will be `\pgfuseplotmark{<plot mark name>}`, where `<plot mark name>` is the name of a predefined plot mark.

\pgfsetplotmarkrepeat{<repeat>}

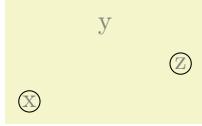
Sets the r parameter to `<repeat>`, that is, only every r th mark will be drawn.

`\pgfsetplotmarkphase{<phase>}`

Sets the p parameter to $\langle phase \rangle$, that is, the first mark to be drawn is the p th, followed by the $(p+r)$ th, then the $(p+2r)$ th, and so on.

`\pgfplotmarklisted{<mark code>}{<index list>}`

This command works similar to the previous one. However, marks will only be placed at those indices in the given $\langle index\ list\rangle$. The syntax for the list is the same as for the `\foreach` statement. For example, if you provide the list $1, 3, \dots, 25$, a mark will be placed only at every second point. Similarly, $1, 2, 4, 8, 16, 32$ yields marks only at those points that are powers of two.



```
\begin{tikzpicture}
\draw[gray] (0,0) node {x} (1,1) node {y} (2,.5) node {z};
\pgfplotmarklisted
{\pgfpathcircle{\pgfpointorigin}{4pt}\pgfusepath{stroke}}
{1,3}
\pgfplotstreamstart
\pgfplotstreampoint{\pgfpoint{0cm}{0cm}}
\pgfplotstreampoint{\pgfpoint{1cm}{1cm}}
\pgfplotstreampoint{\pgfpoint{2cm}{0.5cm}}
\pgfplotstreamend
\pgfusepath{stroke}
\end{tikzpicture}
```

`\pgfuseplotmark{<plot mark name>}`

Draws the given $\langle plot\ mark\ name \rangle$ at the origin. The $\langle plot\ mark\ name \rangle$ must have been previously declared using `\pgfdeclareplotmark`.



```
\begin{tikzpicture}
\draw[gray] (0,0) node {x} (1,1) node {y} (2,.5) node {z};
\pgfplotmark{\pgfuseplotmark{pentagon}}
\pgfplotstreamstart
\pgfplotstreampoint{\pgfpoint{0cm}{0cm}}
\pgfplotstreampoint{\pgfpoint{1cm}{1cm}}
\pgfplotstreampoint{\pgfpoint{2cm}{0.5cm}}
\pgfplotstreamend
\pgfusepath{stroke}
\end{tikzpicture}
```

`\pgfdeclareplotmark{<plot mark name>}{<code>}`

Declares a plot mark for later used with the `\pgfuseplotmark` command.



```
\pgfdeclareplotmark{my plot mark}
{\pgfpathcircle{\pgfpoint{0cm}{1ex}}{1ex}\pgfusepathqstroke}
\begin{tikzpicture}
\draw[gray] (0,0) node {x} (1,1) node {y} (2,.5) node {z};
\pgfplotmark{\pgfuseplotmark{my plot mark}}
\pgfplotstreamstart
\pgfplotstreampoint{\pgfpoint{0cm}{0cm}}
\pgfplotstreampoint{\pgfpoint{1cm}{1cm}}
\pgfplotstreampoint{\pgfpoint{2cm}{0.5cm}}
\pgfplotstreamend
\pgfusepath{stroke}
\end{tikzpicture}
```

`\pgfsetplotmarksizes{<dimension>}`

This command sets the TeX dimension `\pgfplotmarksizes` to $\langle dimension \rangle$. This dimension is a “recommendation” for plot mark code at which size the plot mark should be drawn; plot mark code may choose to ignore this $\langle dimension \rangle$ altogether. For circles, $\langle dimension \rangle$ should be the radius, for other shapes it should be about half the width/height.

The predefined plot marks all take this dimension into account.



```
\begin{tikzpicture}
\draw[gray] (0,0) node {x} (1,1) node {y} (2,.5) node {z};
\pgfsetplotmarksize{1ex}
\pgfplothandlermark{\pgfuseplotmark{*}}
\pgfplotstreamstart
\pgfplotstreampoint{\pgfpoint{0cm}{0cm}}
\pgfplotstreampoint{\pgfpoint{1cm}{1cm}}
\pgfplotstreampoint{\pgfpoint{2cm}{0.5cm}}
\pgfplotstreamend
\pgfusepath{stroke}
\end{tikzpicture}
```

\pgfplotmarksize

A TeX dimension that is a “recommendation” for the size of plot marks.

The following plot marks are predefined (the filling color has been set to yellow):

\pgfuseplotmark{*}	
\pgfuseplotmark{x}	
\pgfuseplotmark{+}	

67 Plot Mark Library

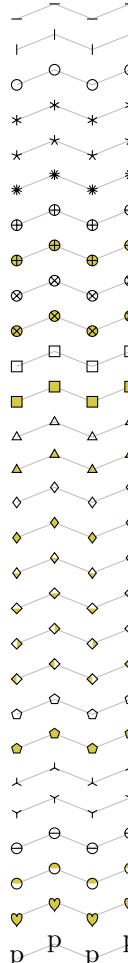
TikZ Library `plotmarks`

```
\usepgflibrary{plotmarks} % LATEX and plain TEX and pure pgf
\usepgflibrary[plotmarks] % ConTEXt and pure pgf
\usetikzlibrary{plotmarks} % LATEX and plain TEX when using TikZ
\usetikzlibrary[plotmarks] % ConTEXt when using TikZ
```

This library defines a number of plot marks.

This library defines the following plot marks in addition to *, x, and + (the filling color has been set to a dark yellow):

```
\pgfuseplotmark{-}
\pgfuseplotmark{|}
\pgfuseplotmark{o}
\pgfuseplotmark{asterisk}
\pgfuseplotmark{star}
\pgfuseplotmark{10-pointed star}
\pgfuseplotmark{oplus}
\pgfuseplotmark{oplus*}
\pgfuseplotmark{otimes}
\pgfuseplotmark{otimes*}
\pgfuseplotmark{square}
\pgfuseplotmark{square*}
\pgfuseplotmark{triangle}
\pgfuseplotmark{triangle*}
\pgfuseplotmark{diamond}
\pgfuseplotmark{diamond*}
\pgfuseplotmark{halfdiamond*}
\pgfuseplotmark{halfsquare*}
\pgfuseplotmark{halfsquare right*}
\pgfuseplotmark{halfsquare left*}
\pgfuseplotmark{pentagon}
\pgfuseplotmark{pentagon*}
\pgfuseplotmark{Mercedes star}
\pgfuseplotmark{Mercedes star flipped}
\pgfuseplotmark{halfcircle}
\pgfuseplotmark{halfcircle*}
\pgfuseplotmark{heart}
\pgfuseplotmark{text}
```



Note that each of the provided marks can be rotated freely by means of `mark options={rotate=90}` or every `mark/.append style={rotate=90}`.

`/pgf/mark color={<color>}`

(no default, initially empty)

Defines the additional fill color for the `halfcircle`, `halfcircle*`, `halfdiamond*` and `halfsquare*` markers. An empty value uses `white` (which is the initial configuration). The special value `none` disables filling of the respective parts.

Note that `halfsquare` will be filled with `mark color`, and the starred variant `halfsquare*` will be filled half with `mark color` and half with the actual `fill` color.

`/pgf/text mark={<text>}`

(no default, initially p)

Changes the text shown by `mark=text`.

With `/pgf/text mark=m:` m 

With `/pgf/text mark=A:` A 

There is no limitation about the number of characters or whatever. In fact, any T_EX material can be inserted as `{<text>}`, including images.

`/pgf/text mark as node={⟨boolean⟩}` (no default, initially `false`)

Configures how `mark=⟨text⟩` will be drawn: either as `\node` or as `\pgftext`.

The first choice is highly flexible and possibly slow, the second is very fast and usually enough.

`/pgf/text mark style={⟨options for mark=⟨text⟩⟩}` (no default)

Defines a set of options which control the appearance of `mark=⟨text⟩`.

If `/pgf/text mark as node=false` (the default), {⟨options⟩} is provided as argument to `\pgftext` – which provides only some basic keys like `left`, `right`, `top`, `bottom`, `base` and `rotate`.

If `/pgf/text mark as node=true`, {⟨options⟩} is provided as argument to `\node`. This means you can provide a very powerful set of options including `anchor`, `scale`, `fill`, `draw`, `rounded corners` etc.

68 Profiler Library

by Christian Feuersänger

TikZ Library `profiler`

```
\usepgflibrary{profiler} % LATEX and plain TEX and pure pgf
\usepgflibrary[profiler] % ConTeXt and pure pgf
\usetikzlibrary{profiler} % LATEX and plain TEX when using TikZ
\usetikzlibrary[profiler] % ConTeXt when using TikZ
```

A library to simplify the optimization of runtime speed of T_EX programs.

It relies on the pdftex primitive `\pdfelapsedtime`¹³ to count (fractional) seconds and counts total time and self time for macro invocations.

68.1 Overview

The intended audience for this library are people writing T_EX code which should be optimized. It is certainly *not* useful for the end-user.

The work flow for the optimization is simple: the preamble contains configuration commands like

```
\usepgflibrary{profiler}
\pgfprofilenewforenvironment{tikzpicture}
\pgfprofilenewforcommand{\pgfkeys}{1}
```

and then, the time between `\begin{tikzpicture}` and `\end{tikzpicture}` and the time required to call `\pgfkeys` will be collected.

At the end, a short usage summary like

```
pgflibraryprofiler(main job) {total time=1.07378sec; (100.0122%) self time=0.034sec; (3.1662%)}
pgflibraryprofiler(<ENV>tikzpicture) {total time=1.03978sec; (96.84601%) self time=1.00415sec; (93.52722%)}
pgflibraryprofiler(<CS>pgfkeys) {total time=0.03563sec; (3.31726%) self time=0.03563sec; (3.31726%)}
```

will be provided in the log file, furthermore, the same information is available in a text table called `\jobname.profiler.<datetime>.dat` which is of the form:

profilerentry	totaltime [s]	totaltime [percent]	selftime [s]	selftime [percent]
main job	1.07378	100.0122	0.034	3.1662
<ENV>tikzpicture	1.03978	96.84601	1.00415	93.52722
<CS>pgfkeys	0.03563	3.31726	0.03563	3.31726

Here, the `totaltime` means the time used for all invocations of the respective profiler entry (one row in the table). The `selftime` measures time which is not already counted for in another profiler entry which has been invoked within the current one. The example above is not very exciting: the main job consists only of several (quite complex) pictures and nothing else. Thus, its total time is large. However, the self time is very small because the `tikzpictures` are counted separately, and they have been invoked within the `main job`. The `\pgfkeys` control sequence has been invoked within the `tikzpicture`, that's why the `selftime` for the `tikzpicture` is a little bit smaller than its `totaltime`.

68.2 Requirements

The library works with pdftex and luatex. Furthermore, it requires a more or less recent version of pdftex which supports the `\pdfelapsedtime` directive.

68.3 Defining Profiler Entries

Unlike profilers for C/C++ or java, this library doesn't extract information about every T_EX macro automatically, nor does it collect information for each of them. Instead, every profiler entry needs to be defined explicitly. Only defined profiler entries will be processed.

```
\pgfprofilenew{<name>}
```

Defines a new profiler entry named `<name>`.

¹³The primitive is emulated in luatex.

This updates a set of internal registers used to track the profiler entry. The *<name>* can be arbitrary, it doesn't need to be related to any TeX macro.

The actual job of counting seconds is accomplished using `\pgfprofilestart{<name>}` followed eventually by the command `\pgfprofileend{<name>}`.

It doesn't hurt if `\pgfprofilenew` is called multiple times with the same name.

`\pgfprofilenewforcommand[<profiler entry name>]{<|macro>}{<arguments>}`

Defines a new profiler entry which will measure the time spent in *<|macro>*. This calls `\pgfprofilenew` and replaces the current definition of *<|macro>* with a new one.

If `[<profiler entry name>]` has been provided, this defines the argument for `\pgfprofilenew`. It is allowed to use the same name for multiple commands; in this case, they are treated as if it were the same command. If the optional argument is not used, the profiler entry will be called '`\pgfprofilecs{macro}' (<macro> without backslash)` where `\pgfprofilecs` is predefined to be `<CS>`.

The replacement macro will collect all required arguments, start counting, invoke the original macro definition and stop counting.

The following macro types are supported within `\pgfprofilenewforcommand`:

- commands which take one (optional) argument in square brackets followed by one optional argument which has to be delimited by curly braces (use an empty argument for *<arguments>* in this case),
- commands which take one (optional) argument in square brackets and *exactly* *<arguments>* arguments afterwards.

Take a look at `\pgfprofilenewforcommandpattern` in case you have more complicated commands.

Note that the library can't detect if a command has been redefined somewhere.

`\pgfprofilenewforcommandpattern[<profiler entry name>]{<|macro>}{<argument pattern>}{<invocation pattern>}`

A variant of `\pgfprofilenewforcommand` which can be used with arbitrary *<argument patterns>*. Example:

```
\def\mymacro#1#2#3{ ... }
\pgfprofilenewforcommandpattern{\mymacro}{#1#2#3}{#1}{#2}{#3}
```

Note that `\pgfprofilenewforcommand` is a special case of `\pgfprofilenewforcommandpattern`:

```
\def\mymacro#1#2{ ... }
\pgfprofilenewforcommand\macro{2}
\pgfprofilenewforcommandpattern{\mymacro}{#1#2}{#1}{#2}
```

Thus, *<argument pattern>* is a copy-paste from the definition of your command. The *<invocation pattern>* is used by the `profiler` library to invoke the *original* command, so it is closely related to *<argument pattern>*, but it needs extra curly braces around each argument.

The behavior of `\pgfprofilenewforcommandpattern` is the same as discussed above: it defines a new profiler entry which will measure the time spent in *<|macro>*. The details about this definition has already been described. Note that up to one optional argument in square brackets is also checked automatically.

If you like to profile a command which doesn't match here for whatever reasons, you'll have to redefine it manually and insert `\pgfprofilestart` and `\pgfprofileend` in appropriate places.

`\pgfprofileshowinvocationsfor{<profiler entry name>}`

Enables verbose output for *every* invocation of *<profiler entry name>*.

This is only available for profiler entries for commands (those created by `\pgfprofilenewforcommand` for example). It will also show all given arguments.

`\pgfprofileshowinvocationsexpandedfor{<profiler entry name>}`

A variant of `\pgfprofileshowinvocationsfor` which will expand all arguments for *<profiler entry name>* before showing them. The invocation as such is not affected by this expansion.

This expansion (with `\edef`) might yield unrecoverable errors for some commands. Handle with care.

`\pgfprofilenew[profiler entry name]{environment name}`

Defines a new profiler entry which measures time spent in the environment *environment name*.

This calls `\pgfprofilenew` and handles the begin/end of the environment automatically.

The argument for `\pgfprofilenew` is *profiler entry name*, or, if this optional argument is not used, it is ‘`\pgfprofileenv{environment name}`’ where `\pgfprofileenv` is predefined as `<ENV>`. Again, it is permitted to use the same *profiler entry name* multiple times to merge different commands into one output section.

`\pgfprofilestart{profiler entry name}`

Starts (or resumes) timing of *profiler entry name*. The argument must have been declared in the preamble using `\pgfprofilenew`.

Nested calls of `\pgfprofilestart` with the same argument will be ignored.

The invocation of this command doesn’t change the environment: it doesn’t introduce any `\TeX` groups nor does it modify the token list.

`\pgfprofileend{profiler entry name}`

Stops (or interrupts) timing of *profiler entry name*.

This command finishes a preceding call to `\pgfprofilestart`.

`\pgfprofilepostprocess`

For `\LaTeX`, this command is installed automatically in `\end{document}`. It stops all running timings, evaluates them and returns the result into the logfile. Furthermore, it generates a text table called `\jobname.profiler.<YYYY>-<MM>-<DD>_<HH>\h_<MM>\m.dat` with the same information.

Note that the `profiler` library predefines two profiler entries, namely `main` job which counts time from the beginning of the document until `\pgfprofilepostprocess` and `preamble` which counts time from the beginning of the document until `\begin{document}`.

`\pgfprofilesetrel{profiler entry name}` (initially main job)

Sets the profiler entry whose total time will be used to compute all other relative times. Thus, *profiler entry name* will use 100% of the total time per definition, all other relative times are relative to this one.

`\pgfprofileifisrunning{profiler entry name}{{true code}}{{false code}}`

Invokes {{*true code*}} if {{*profiler entry name*}} is currently running and {{*false code*}} otherwise.

69 Resource Description Framework Library

With certain output formats (in particular, with SVG), TikZ can add *semantic annotations* to an output file. Consider as an example the drawing of a finite automaton. In your \TeX code, you might have a nice description of the automaton like the following:

```
\tikz[automaton] \graph { a[state, initial] ->[transition] b [state] ->[transition] c[state, final] };
```

This description of the automaton carries a lot of “semantic information” like the information that the node `a` is not just some node, but actually the initial state of the automaton, while `c` is a final state. Unfortunately, in the output produced TikZ, this information is normally “lost”: In the output, `a` is only a short text, possibly with a circle drawn around it; but there is no information that *this* text and *this* circle together form the state of an automaton.

As a human (more precisely, as a computer scientist), you might “see” that the text and the circle form a state, but most software will have a very hard time retrieving this semantic information from the output. In particular, it is more or less impossible to design a search engine that you can query to find, say, “all automata with three states” in a document.

This is the point where *semantic annotations* come in. These are small labels or “hints” in the *output* that tell you (and, more importantly, a program) that the text and the circle together form a state of an automaton. There is a standard for specifying such annotations (“resource description framework annotations”, abbreviated RDFa) and TikZ provides a way of adding such annotations to an output file using the `rdf engine` key, explained in a moment. Note, however, that the output format must support such annotations; currently TikZ only supports SVG.

69.1 Starting the RDF Engine

TikZ Library `rdf`

```
\usetikzlibrary{rdf} % \TeX and plain \TeX  
\usetikzlibrary[rdf] % Con\TeX t
```

You need to load this library for the keys described in the following. However, even when this library is loaded, RDF information is only written to the output inside scopes where the following key is set:

`/tikz/rdf engine on` (no value)

Switches “on” the generation of RDF information for the current \TeX scope. The idea is that libraries can internally use the `rdf engine` key (explained below) a lot in order to provide good semantic information in the output when desired, but need not worry that this will bloat output files since users have to use this key explicitly to include semantic information in the output.

`/tikz/rdf engine=<rdf keys>` (no default)

This key only has an effect when `rdf engine on` is called, otherwise the argument is silently ignored. The `<rdf keys>` get executed with the path prefix `/tikz/rdf engine` at the beginning of the current scope (for a node, at the beginning of the node’s scope). Depending on which keys are used, semantic information gets to be added to the output.

Note that you cannot simply the keys with path prefix `/tikz/rdf engine` directly since they need to be executed at very specific times during TikZ’s processing of scopes. Always call those keys via this key.

The following key is useful for generally setting the prefix for a larger number of annotations:

`/tikz/rdf engine/prefix=<prefix: iri>` (no default)

Inside the current scope, you can use `<prefix>`: inside curies (compact universal resource identifier expressions, see the RDFa specification) as an abbreviation for the `<iri>`. (It has the same effect as the `prefix` attribute in RDFa.) You can use this key several times for a given scope.

```
\scoped [rdf engine = {  
    prefix = {rdf: http://www.w3.org/1999/02/22-rdf-syntax-ns\,tikzrdfhashmark},  
    prefix = {automata: http://www.tcs.uni-luebeck.de/ontologies/2016/04/28/automata/},  
    statement = { ...., predicate = rdf:type, object = automata:state },  
    statement = { ...., predicate = rdf:type, object = automata:final },  
}] ...
```

The above could also be written more verbosely as

```
\scoped [rdf engine = {
    statement = { ...
        predicate = http://www.w3.org/1999/02/22-rdf-syntax-ns\!tikzrdfhashmark type,
        object = http://www.tcs.uni-luebeck.de/ontologies/2016/04/28/automata/state }
    },
    statement = { ...
        predicate = http://www.w3.org/1999/02/22-rdf-syntax-ns\!tikzrdfhashmark type,
        object = http://www.tcs.uni-luebeck.de/ontologies/2016/04/28/automata/final }
}] ...
```

The use of the command `\tikzrdfhashmark` is necessary since TeX assigns a special meaning to hash marks. The command simply expands to a “normal” hash mark for use in texts.

`\tikzrdfhashmark`

Expands to # with catcode 11.

69.2 Creating Statements

TikZ’s method of adding semantic information to an output is based on the principles underlying the *resource description framework* (RDF). In this framework, all semantic information is encoded using a large graph consisting of nodes and connecting directed edges, but the nodes are called *resources* and the edges are called *statements*. A resource is identified by an IRI, an *internationalized resource identifier*, which basically looks like the well-known URLs, but allows additional Unicode characters. Note that these IRIs do not need to point to “real” webpages, they are just a way of conceptually identifying resources uniquely and permanently. Similarly, each edge (statement) of the RDF graph has such an IRI attached to it, which identifies the “flavour” of the arc.

In a “mathematical” graph, each edge has a “tail” and a “head” vertex and a label, but in the context of the resource description framework these notions are called differently: As mentioned before, an edge is called a *statement*, the tail of this edge is called the *subject*, the head is called the *object* (in the linguistic sense), and the label is called the *predicate*. Thus, a statement is – quite fittingly – a triple “subject predicate object”.

Note that in the RDF framework *all* semantic information must be encoded using statements of this fixed kind. Many semantic notions are easy to store in this way such as “Albert Einstein was a physicist” (“Albert Einstein” is the subject, “was” is the predicate, “a physicist” is the object), but other notions do not fit well like “The automaton has states q_1, q_2, q_a , and q_b ” since there are several objects in the statement. Nevertheless, all information must be encoded as simple statements with a single subject, a single predicate, and a single object.

You add an RDF statement to the output file using the following key:

`/tikz/rdf engine/statement={⟨options⟩}` (no default)

Each use of this key will add one RDF statement to the output file. The `⟨options⟩` will be executed with the path prefix `/tikz/rdf engine/statements` and must use the three keys `subject`, `predicate`, and `object` to specify the three components of the statement (these keys can, however, be called by styles internally, so not all statements will explicitly set these three keys). Note that *all three must always be set*, it is *not* possible to setup, say, just a subject for a scope and then omit the subject for statements inside the scope. (However, using styles you can setup things in such a way that a certain subject is used for several statements.)

```
\tikz [rdf engine = {
    statement = {
        subject = http://www.example.org/persons/Einstein,
        predicate = http://www.example.org/predicates/isA,
        object = http://www.example.org/professions/physicist
    },
    statement = {
        subject = http://www.example.org/persons/Curie,
        predicate = http://www.example.org/predicates/isA,
        object = http://www.example.org/professions/physicist
    } ] { ... }
```

The statements are normally added at the beginning of the scope where the `rdf engine` command is used (except when the `object` is `scope content`, which is explained later). This means that when you use `prefix` inside an `rdf engine` command, it will apply to all statements, regardless of the order.

`/tikz/rdf engine/statements/subject=<subject>` (no default)

Sets the subject of the to-be-created statement. The `<subject>` can be in one of two possible formats:

1. A curie (a *compact universal resource identifier expression*, see the RDFA specification for details). Examples are standard URLs like `http://www.example.org`, but also text like `#my_automaton`. Note that in order to include a hashmark in a curie you should use the command `\tikzrdfhashmark`, which expands to a hash mark (TeX treats hash marks in a special way, which is why this command is used here).
2. When the `<subject>` starts with an opening parenthesis, that is, with “(”, the `<subject>` must have the form `(<node or scope name>)`. In this case, the `<node or scope name>` must be the name of an already existing node (the current node or scope is considered as “existing” here). Then, the curie `#<id>` is used as subject, where the `<id>` is a unique internal identifier for the node.

As an example, suppose you wish to specify that a node has some other node as child, you could write the following:

```
\tikz [ rdf engine = { prefix = { rels: http://www.example.org/relations/ } } ] {
    \node (fritz)           { Fritz };
    \node (heinz) at (2,0) { Heinz };
    \draw [->] (fritz) -- (heinz)
        [rdf engine = {
            statement = {
                subject = (fritz),
                predicate = rels:isSonOf,
                object   = (heinz)
            } } ];
}
```

You can use a macro as `<subject>`, it will be expanded before the above syntax check is done.

If you use the `subject` key several times inside a single `statement` command, (only) the last subject is used.

`/tikz/rdf engine/statements/predicate=<predicate>` (no default)

Sets the predicate for the statement. The syntax is exactly the same as for the subject. Unlike for subjects, you can use the predicate key several times inside a single statement and the uses will “accumulate” and several statements are created, namely one statement for each use of `predicate` for the subject and object specified inside the use of `statement`. This behavior is not very systematic (it violates the rule “one statement per `statement`”) and you should normally use the `statement` once for each use of the `predicate` key. However, in conjunction with the object `scope content` it is necessary to allow this behavior.

`/tikz/rdf engine/statements/object=<object>` (no default)

Sets the object for the statement. The syntax allowed for the `<object>` is as follows:

1. As for `subject` and `predicate` you can use a curie here. This is the default unless one of the following special cases is used:
2. As for `subject` and `predicate`, you can use the syntax `(<name of node or scope>)` to create and use a curie for the node or scope.
3. If the `<object>` starts with “”, it must have the syntax “`<literals>`”. In this case, the object of the statement is not a curie (not a normal “resource”) but the string of `<literals>` given.
4. If the `<object>` is the text “`scope content`”, the object of the statement is actually the whole contents of the scope to which this statement is attached.
5. The two previous cases can be combined in the form of an object of the form “`<literals>`” and `scope content`. In this case, the contents of the scope is “normally” the object, but this gets “overruled” by the `<literals>`. Formally, this means that the object is the `<literals>`, but the intended semantics is that the object is the scope content, only for further processing it should be considered to be `<literals>`. A typical example is the case where the scope content is, say, the text “January 1st, 2000” but the `<literals>` are set to `2000-01-01`, which is easier for software to process:

```
\node [rdf engine = {
    statement = {
        subject = ...,
        predicate = dc:Date,
        object = "2000-01-01" and scope content
    } } ] { January 1st, 2000 };
```

For the last two cases, only one statement may be given per scope that has the `scope content` as its object; if more than one is given, the last one wins. This is the reason why several uses of `predicate` are allowed in a `statement`.

`/tikz/rdf engine/statements/has type=<type>`

(no default)

This style is a shorthand for `predicate=rdf:type` and `object=<type>`.

69.3 Creating Resources

In RDF statements, when you can use the name of a TikZ scope or node surrounded by parenthesis, a curie is inserted into the output that references this scope or node. While this makes it easy to describe relationships between existing nodes, in library code (RDF generation code added to library styles and gets executed as a “byproduct”) there are two situations where this method is insufficient:

1. You may sometimes wish to create additional resources in the RDF graph that are not represented by any concrete node or scope in the TikZ picture. For instance, a finite automaton has a set of states and a set of transitions, but neither of these “containers” has any concrete representation in the output. One could, of course, create “dummy scopes” for this purpose, but this is rather hard to do inside styles of a library.
2. You may not know the name of the current scope in library code. For instance, a style like `state`, which can be added to a node to indicate that the node is supposed to be a state in a finite automaton, might contain something like the following code:

```
state/.style = {
    draw, circle, minimum size = ...,
    rdf engine = {
        subject = ???,
        predicate = rdf:type,
        object = automata:state
    }
}
```

The problem is, of course, what should be passed to the `subject`. We cannot even write something like `(\tikz@fig@name)` since no name may have been set for the state.

Each of the above problems is solved by a special keys:

`/tikz/rdf engine/get new resource curie=<macro>`

(no default)

The `<macro>` will be set to a new unique curie that can be used anywhere where a curie is allowed. Here is an example how we can add a state and a transition container to an automaton, both of which have no corresponding scope in TikZ.

```
\tikz [ name = my automaton,
    rdf engine = {
        get new resource curie = \statecurie,
        get new resource curie = \transitiocurie,
        statement = {
            subject = (my automaton),
            predicate = automata:hasStateSet,
            object = \statecurie },
        statement = {
            subject = \statecurie,
            hat type = automata:stateSet },
        statement = {
            subject = (my automaton),
            predicate = automata:hasTransitionSet,
            object = \transitiocurie },
        statement = {
            subject = \transitiocurie,
            hat type = automata:transitionSet } } ] { ... }
```

The `<macro>` will be valid for the whole scope.

`/tikz/rdf engine/get scope curie=<macro>` (no default)

The `<macro>` will be set to a unique curie that represents the scope or node. If the scope is named (using the `name` key or the special parenthesis syntax for nodes) and this name is later referenced in another statement, the same curie will be generated. Note how in the following code no name is given for the automaton, which means that the whole RDF code could be moved inside a style like `finite automaton` or something similar.

```
\tikz [ rdf engine = {
    get new resource curie = \statecurie,
    get new resource curie = \transitiocurie,
    get scope curie = \automatoncurie,
    statement = {
        subject = \automatoncurie,
        predicate = automata:hasStateSet,
        object = \statecurie },
    statement = {
        subject = \statecurie,
        hat type = automata:stateSet },
    statement = {
        subject = \automatoncurie,
        predicate = automata:hasTransitionSet,
        object = \transitiocurie },
    statement = {
        subject = \transitiocurie,
        hat type = automata:transitionSet } } ] { ... }
```

The `<macro>` will be valid for the whole scope.

The following key builds on the above keys:

`/tikz/rdf engine/scope is new context` (no value)

This key executes `get scope curie=\tikzrdfcontext`, thereby setting the macro `\tikzrdfcontext` to the current scope. The idea is the key is used with “major resources” and that keys can use this macro as the `subject` of statements if no subject is given explicitly. For instance, a `title` key might be defined as follows:

```
title/.style = {
    rdf engine = { statement = {
        subject = \tikzrdfcontext,
        predicate = dc>Title,
        object = "#1"
    } } }
```

69.4 Creating Containers

A *container* is a resource that represents a set or a sequence of elements. In RDF this is modeled by having a statement say that the type of the resource is something special like `rdf:Seq` and then for each member resource of the container add a statement saying that the container has as its *i*th member the member resource. Here is an example of a container with two elements:

```
\tikz {
  \node (safe) { Safe }
  child { node (coins) {Coins} }
  child { node (gold) {Gold} };

  \scoped [rdf engine = {
    statement = {
      subject = (safe),
      has type = rdf:Seq
    },
    statement = {
      subject = (safe),
      predicate = rdf:_1,
      object = (coins)
    },
    statement = {
      subject = (safe),
      predicate = rdf:_2,
      object = (gold)
    }
  }];
}
```

However, the above code is error-prone and does not integrate well with styles and libraries. For this reason, TikZ offers some styles that may help in creating containers:

/tikz/rdf engine/statements/is a container

(no value)

Add this key to a statement in order to tell TikZ that it should setup a special counter for the subject of the statement that keeps track of the container's children.

/tikz/rdf engine/statements/has as member

(no value)

This key may only be added to statements whose subject was previously used as a subject in a statement containing the `is a container` key. In this case, the internal counter will be increased and the predicate will be set to `rdf:_<count>`. This means that we can write the above code as:

```
\tikz {
  ...
  \scoped [rdf engine = {
    statement = {
      subject = (safe),
      has type = rdf:Seq,
      is a container,
    },
    statement = {
      subject = (safe),
      has as member,
      object = (coins)
    },
    statement = {
      subject = (safe),
      has as member,
      object = (gold)
    }
  }];
}
```

/tikz/rdf engine/statements/is a sequence

(no value)

This is a shorthand for `predicate = rdf:Seq, is a container`. In the above example we could say:

```
\tikz {
  ...
  \scoped [rdf engine = {
    statement = {
      subject = (safe),
      is a sequence
    },
    ...
  }];
}
```

/tikz/rdf engine/statements/is a bag

(no value)

This is a shorthand for `predicate = rdf:Bag, is a container`.

`/tikz/rdf engine/statements/is an alternative`

(no value)

This is a shorthand for predicate = `rdf:Alt`, is a container.

69.5 Creating Semantic Information Inside Styles and Libraries

The RDF library was designed in such a way that normal document authors do not need to use the keys of the library explicitly, except possibly for saying `rdf engine` on somewhere at the beginning. Instead, library authors should include the necessary commands to generate RDF information that is then automatically included in the output. Furthermore, if the author does not “switch on” the generation of RDF information, all uses of `rdf engine` will simply be ignored silently and neither the speed of compilation nor the size of the generated files is impacted.

69.5.1 An Example Library for Drawing Finite Automata

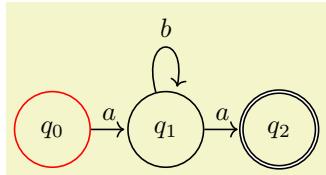
In the following, we have a look at how a library might be augmented by RDF generation keys. The library we augment is a (fictitious) library for drawing finite automata. The library offers the following styles:

1. `dfa` and `nfa` can be added to a scope to indicate that the scope contains a deterministic or a nondeterministic finite automaton.
2. `state` can be added to a node to indicate that the node is a state in the automaton.
3. `initial` and `final` are used to indicate that a state is an initial or final state, which should be rendered in a special way.
4. `transition` can be added to an edge to indicate that there is a transition in the automaton from the first state to the second state. The style takes a parameter which is the symbol read by the automaton.

Here are some possible definitions of these keys that do not (yet) generate RDF information:

```
\tikzset{
  dfa/.style      = { semithick, > = To [sep] },
  nfa/.style      = { semithick, > = To [sep] },
  state/.style    = { circle, draw, minimum size = 1cm },
  final/.style    = { double },
  initial/.style  = { draw = red }, % to keep things simple
  transition/.style = { edge label = {$\#1$} } }
```

The library could be used as follows:



```
\usetikzlibrary {graphs,rdf}
\tikz [dfa]
\graph [math nodes, grow right = 1.5cm] {
  q_0 [state, initial] -- [transition = a] q_1 [state]
  q_1 -- [transition = b, loop above] q_1
  q_1 -- [transition = a] q_2 [state, final]
};
```

69.5.2 Adding Semantic Information About the Automata as a Whole

Let us change the different keys so that they add RDF information to the output. For this, we first need an ontology that defines notions like “state” or “deterministic finite automaton”. For the purposes of this example, we just assume that such an ontology exists at <http://www.tcs.uni-luebeck.de/ontologies/automata/>. The new definition of the `dfa` key might start as follows (we will extend these later on):

```

dfa/.style = {
    semithick, > = To [sep], % as before,
    rdf engine = {
        %
        % Setup prefix:
        prefix = { automata: http://www.tcs.uni-luebeck.de/ontologies/automata/ },
        %
        % Get the curie of the automaton and store it in a macro for later use:
        get scope curie = \mylibAutomatonCurie,
        %
        % Make a statement that the resource is, indeed, an automaton:
        statement = {
            subject      = \mylibAutomatonCurie,
            has type    = automata:types/automaton },
        %
        % Make a statement that the automaton is deterministic:
        statement = {
            subject      = \mylibAutomatonCurie,
            predicate   = automata:properties/deterministic,
            object      = "yes" } } }

```

The definition of the style `nfa` would be exactly the same as for `dfa`, except, of course, that the last statement would have "no" as object. Note that the original styles `dfa` and `nfa` has identical definitions since, indeed, there is no "visual" difference between the two. In contrast, the RDF information stores this information in the output.

69.5.3 Adding Semantic Information About the States

We next augment the styles for creating states and marking them as final or initial. We could do the following (note that we do not setup the prefix since this has been done by the surrounding `dfa` or `nfa` key):

```

state/.style = {
    circle, draw, minimum size = 1cm, % as before,
    rdf engine = {
        get scope curie = \mylibStateCurie,
        statement = {
            subject      = \mylibStateCurie,
            has type    = automata:types/state },
        statement = {
            subject      = \mylibStateCurie,
            predicate   = rdf:value,
            object      = scope content } } }

```

The first statement tells us that the circle with its contents is a state (and not just "any" fancy circle). The second statement tells us that the "value" of this state is the content. One might argue that, instead, only the number itself (like " q_0 " or perhaps only "0") should be the "value" or, perhaps, a different property should actually be used (like `automata:stateNumber` or something like that). However, these are questions of ontological modeling, not of the use of the RDF engine.

What is definitely missing from the above definition is a link between the automaton resource and the state resource. Note that the state's rendering code will be inside the scope of the automaton, so, in a sense, the state is "inside" the automaton in the output. However, the nesting of scopes is *not* part of the RDF graph; we must make these relationships explicit using statements. One way to achieve this would be to add the following to the `state` style:

```

statement = {
    subject      = \mylibAutomatonCurie,
    predicate   = automata:hasAsAState,
    object      = \mylibStateCurie }

```

Note that we can access the macro `\mylibAutomatonCurie` here since this will have been setup by the surrounding `dfa` or `nfa` key. While the above is possible and legitimate, we will see a better solution using containers in a moment ("better" in the sense that the ontological model is easier to process by software).

The two styles `final` and `initial` are easy to augment:

```

final/.style = {
  double, % as before,
  rdf engine = {
    get scope curie = \mylibStateCurie,
    statement = {
      subject      = \mylibStateCurie,
      has type     = automata:properties/final } } }

```

Note that when we write `node [state, final] ...` the state now has two types: It has type `automata:types/state` and also `automata:properties/final`. This is perfectly legitimate. Also note that I added `get scope curie` to the above definition, which may seem superfluous since the `state` style already executes this key to get a curie for the state resource. However, users should be free to write `node [final, state] ...` and, now, the `final` key will be executed first.

The style for `initial` is the same as for `final` only with a different type.

69.5.4 Adding Semantic Information About the Transitions

A transition is, essentially, a labeled edge from a state to another state. It may seem tempting to model them as a statement with the first state as its subject and the second state as the object and the transition's symbol as the label (turned into a curie in some appropriate way). However, closer inspection shows that this is not a good way of modeling transitions: In essence, it is just coincidence that the RDF graph happens to be a directed graph and, at the same time, the thing we describe by it (the automaton) can also be viewed as a directed graph. If, for instance, we consider alternating automata where a transition can involve more than two states, the simple model breaks down.

The “right” way of modeling a transition is to treat the transition as a resource of its own and then make statements like “the transition has this state as its old state”. This turns out to be relatively easy to achieve:

```

transition/.style = {
  edge label = {#1}, % as before,
  rdf engine = {
    get scope curie = \mylibTransitionCurie,
    statement = {
      subject      = \mylibTransitionCurie,
      has type     = automata:types/transition },
    statement = {
      subject      = \mylibTransitionCurie,
      predicate   = automata:properties/symbolReadFromTape,
      object       = "#1" },
    statement = {
      subject      = \mylibTransitionCurie,
      predicate   = automata:relations/oldState,
      object       = (\tikztostart) },
    statement = {
      subject      = \mylibTransitionCurie,
      predicate   = automata:relations/newState,
      object       = (\tikztotarget) } } }

```

69.5.5 Using Containers

As a last step we wish to organize the states and transitions using containers. As explained earlier, we can easily add statements linking our automaton to the states and to the transitions, but the RDF standard has standard way of specifying that a set of resources form a logical sequence: containers.

In case automata contained *only* states, we could setup the automaton itself to be the container and the states to be its elements. However, the automaton has states and transitions and in this example I would like to keep these in separate containers. Thus, we must create two containers and then make statements that the automaton contains these two containers. Since these containers do not have any accompanying visual representation, we use the `get new resource curie` key to create new resources that are purely for descriptive purposes inside the RDF graph:

```

dfa/.style = {
    semithick, > = To [sep], % as before,
    rdf engine = {
        prefix = { automata: http://www.tcs.uni-luebeck.de/ontologies/automata/ },
        get scope curie = \mylibAutomatonCurie,
        statement = { ... as before that automaton has type automata:types/automaton ... },
        statement = { ... as before that automaton is deterministic ... },
        get new resource curie = \mylibStateContainerCurie,
        statement = {
            subject      = \mylibStateContainerCurie,
            is a sequence },
        statement = {
            subject      = \mylibAutomatonCurie,
            predicate   = automata:relations/hasAsStateContainer,
            object       = \mylibStateContainerCurie },
        get new resource curie = \mylibTransitionContainerCurie,
        statement = {
            subject      = \mylibTransitionContainerCurie,
            is a sequence },
        statement = {
            subject      = \mylibAutomatonCurie,
            predicate   = automata:relations/hasAsTransitionContainer,
            object       = \mylibTransitionContainerCurie } } }

```

We can now modify the `state` style as follows:

```

state/.style = {
    circle, draw, minimum size = 1cm, % as before,
    rdf engine = {
        get scope curie = \mylibStateCurie,
        statement = { ... as before ... },
        statement = { ... as before ... },
        statement = {
            subject      = \mylibStateContainerCurie,
            has as member,
            object       = \mylibStateCurie } } }

```

The modification for the `transition` style is similar:

```

transition/.style = {
    edge label = {\#1}, % as before,
    rdf engine = {
        get scope curie = \mylibTransitionCurie,
        statement = { ... as before ... },
        statement = {
            subject      = \mylibTransitionContainerCurie,
            has as member,
            object       = \mylibTransitionCurie } } }

```

69.5.6 The Resulting RDF Graph

Putting it all together, we now get the following library code:

```

\tikzset{
    dfa/.style = {
        semithick, > = To [sep], % as before,
        rdf engine = {
            prefix = { automata: http://www.tcs.uni-luebeck.de/ontologies/automata/ },
            get scope curie = \mylibAutomatonCurie,
            statement = {
                subject      = \mylibAutomatonCurie,
                has type    = automata:types/automaton },
            statement = {
                subject      = \mylibAutomatonCurie,
                predicate   = automata:properties/deterministic,
                object      = "yes" },
            get new resource curie = \mylibStateContainerCurie,
            statement = {
                subject      = \mylibStateContainerCurie,
                is a sequence },
            statement = {
                subject      = \mylibTransitionContainerCurie,
                has as member,
                object       = \mylibTransitionCurie } } }

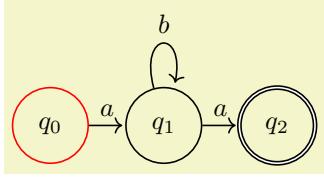
```

```

statement = {
    subject      = \mylibAutomatonCurie,
    predicate    = automata:relations/hasAsStateContainer,
    object       = \mylibStateContainerCurie },
get new resource curie = \mylibTransitionContainerCurie,
statement = {
    subject      = \mylibTransitionContainerCurie,
    is a sequence },
statement = {
    subject      = \mylibAutomatonCurie,
    predicate    = automata:relations/hasAsTransitionContainer,
    object       = \mylibTransitionContainerCurie } } },
state/.style = {
    circle, draw, minimum size = 1cm, % as before,
    rdf engine = {
        get scope curie = \mylibStateCurie,
        statement = {
            subject      = \mylibStateCurie,
            has type     = automata:types/state },
        statement = {
            subject      = \mylibStateCurie,
            predicate    = rdf:value,
            object       = scope content },
        statement = {
            subject      = \mylibStateContainerCurie,
            has as member,
            object       = \mylibStateCurie } } },
initial/.style = {
    draw = red, % as before,
    rdf engine = {
        get scope curie = \mylibStateCurie,
        statement = {
            subject      = \mylibStateCurie,
            has type     = automata:properties/initial } } },
final/.style = {
    double, % as before,
    rdf engine = {
        get scope curie = \mylibStateCurie,
        statement = {
            subject      = \mylibStateCurie,
            has type     = automata:properties/final } } },
transition/.style = {
    edge label = {$#1$}, % as before,
    rdf engine = {
        get scope curie = \mylibTransitionCurie,
        statement = {
            subject      = \mylibTransitionCurie,
            has type     = automata:types/transition },
        statement = {
            subject      = \mylibTransitionCurie,
            predicate    = automata:properties/symbolReadFromTape,
            object       = "#1" },
        statement = {
            subject      = \mylibTransitionCurie,
            predicate    = automata:relations/oldState,
            object       = (\tikz{start}) },
        statement = {
            subject      = \mylibTransitionCurie,
            predicate    = automata:relations/newState,
            object       = (\tikz{target}) },
        statement = {
            subject      = \mylibTransitionContainerCurie,
            has as member,
            object       = \mylibTransitionCurie } } }
}

```

Using this code is still “as easy as before”, indeed, the code for creating the automaton is perfectly unchanged:



```
\usetikzlibrary {graphs,rdf}
\begin{tikzpicture} [math nodes, grow right = 1.5cm]
\graph [transition = a] {
q_0 [state, initial] --> q_1 [state]
q_1 --> q_2 [state, final]
q_1 --> q_1
}
\graph [transition = b, loop above] {
q_1 --> q_1
}
\graph [transition = a] {
q_1 --> q_2
}
\end{tikzpicture}
```

Let us now have a look at the result. If the above is processed using TeX and transformed to SVG code, the following results (reformatted and slightly simplified):

```
<g id="pgf3" prefix="automata: http://www.tcs.uni-luebeck.de/ontologies/automata/">
<!-- The automaton -->
<g about="#pgf3" property="rdf:type" resource="automata:types/automaton" />
<g about="#pgf3" property="automata:properties/deterministic" content="yes" />
<g about="#pgf4" property="rdf:type" resource="rdf:Seq" />
<g about="#pgf3" property="automata:relations/hasAsStateContainer" resource="#pgf4" />
<g about="#pgf5" property="rdf:type" resource="rdf:Seq" />
<g about="#pgf3" property="automata:relations/hasAsTransitionContainer" resource="#pgf5" />
<g id="pgf6" about="#pgf6" property="rdf:value">
<!-- State $q_0$ -->
<g about="#pgf6" property="rdf:type" resource="automata:types/state" />
<g about="#pgf4" property="rdf:_1" resource="#pgf6" />
<g about="#pgf6" property="rdf:type" resource="automata:properties/initial" />
<g stroke="#f00"> <!-- Red Line -->
<path id="pgf6bp" d="M 14.22636 0.0 C 14.22636 7.8571 7.8571 14.22636 0.0 14.22636 ..." />
...
</g>
</g>
<g id="pgf7" about="#pgf7" property="rdf:value">
<!-- State $q_1$ -->
<g about="#pgf7" property="rdf:type" resource="automata:types/state" />
<g about="#pgf4" property="rdf:_2" resource="#pgf7" />
<path id="pgf7bp" d="M 56.90549 0.0 C 56.90549 7.8571 50.53622 14.22636 42.67912 14.22636 ..." />
...
</g>
<g id="pgf8" >
<!-- Transition from $q_0$ to $q_1$ -->
<g about="#pgf8" property="rdf:type" resource="automata:types/transition" />
<g about="#pgf8" property="automata:properties/symbolReadFromTape" content="a" />
<g about="#pgf8" property="automata:relations/oldState" resource="#pgf6" />
<g about="#pgf8" property="automata:relations/newState" resource="#pgf7" />
<g about="#pgf5" property="rdf:_1" resource="#pgf8" />
<path id="pgf8p" d="M 14.52637 0.0 L 26.49275 0.0"/>
...
</g>
<g id="pgf11" >
<!-- Transition loop at $q_1$ -->
<g about="#pgf11" property="rdf:type" resource="automata:types/transition" />
<g about="#pgf11" property="automata:properties/symbolReadFromTape" content="b" />
<g about="#pgf11" property="automata:relations/oldState" resource="#pgf7" />
<g about="#pgf11" property="automata:relations/newState" resource="#pgf7" />
<g about="#pgf5" property="rdf:_2" resource="#pgf11" />
<path id="pgf11p" d="M 38.91211 14.05888 C 33.07051 35.8591 51.00113 36.72765 47.05392 16.66444"/>
...
</g>
<g id="pgf12" about="#pgf12" property="rdf:value">
<!-- State $q_2$ -->
<g about="#pgf12" property="rdf:type" resource="automata:types/state" />
<g about="#pgf4" property="rdf:_3" resource="#pgf12" />
<g about="#pgf12" property="rdf:type" resource="automata:properties/final" />
<g stroke-width="1.80002"> <!-- Double Line -->
<path id="pgf12bp" d="M 99.58461 0.0 C 99.58461 7.8571 93.21535 14.22636 85.35825 14.22636 ..." />
...
</g>
</g>
<g id="pgf13" >
<!-- Transition from $q_1$ to $q_2$ -->
<g about="#pgf13" property="rdf:type" resource="automata:types/transition" />
<g about="#pgf13" property="automata:properties/symbolReadFromTape" content="a" />
<g about="#pgf13" property="automata:relations/oldState" resource="#pgf7" />
<g about="#pgf13" property="automata:relations/newState" resource="#pgf12" />
```

```
<g about="#pgf5" property="rdf:_3" resource="#pgf13" />
<path id="pgf13p" d="M 57.20549 0.0 L 69.17188 0.0"/>
...
</g>
</g>
```

When this code is processed by some RDFa tool, the following graph will result where the blue nodes represent resources:

This example can only be typeset using \LaTeX .

70 Shadings Library

TikZ Library `shadings`

```
\usepgflibrary{shadings} % LATEX and plain TEX and pure pgf
\usepgflibrary[shadings] % ConTEXt and pure pgf
\usetikzlibrary{shadings} % LATEX and plain TEX when using TikZ
\usetikzlibrary[shadings] % ConTEXt when using TikZ
```

The package defines a number of shadings in addition to the ball and axis shadings that are available by default.

In the following, the shadings defined in the library are listed in alphabetical order. The colors of some of these shadings can be configured using special options (like `left color`). These options implicitly select the shading.

The three shadings `axis`, `ball`, and `radial` are always defined, even when this library is not used.

Shading `axis`

In this always-defined shading the colors change gradually between three horizontal lines. The top line is at the top (uppermost) point of the path, the middle is in the middle, the bottom line is at the bottom of the path.

`/tikz/top color=(color)` (no default)

This option sets the color to be used at the top in an `axis` shading. When this option is given, several things happen:

1. The `shade` option is selected.
2. The `shading=axis` option is selected.
3. The middle color of the axis shading is set to the average of the given top color `(color)` and of whatever color is currently selected for the bottom.
4. The rotation angle of the shading is set to 0.



```
\usepgflibrary {shadings}
\tikz \draw[top color=red] (0,0) rectangle (2,1);
```

`/tikz/bottom color=(color)` (no default)

This option works like `top color`, only for the bottom color.

`/tikz/middle color=(color)` (no default)

This option specifies the color for the middle of an axis shading. It also sets the `shade` and `shading=axis` options, but it does not change the rotation angle.

Note: Since both `top color` and `bottom color` change the middle color, this option should be given *last* if all of these options need to be given:



```
\usepgflibrary {shadings}
\tikz \draw[top color=white,bottom color=black,middle color=red]
(0,0) rectangle (2,1);
```

`/tikz/left color=(color)` (no default)

This option does exactly the same as `top color`, except that the shading angle is set to 90°.

`/tikz/right color=(color)` (no default)

Works like `left color`.

Shading `ball`

This always-defined shading fills the path with a shading that “looks like a ball”. The default “color” of the ball is blue (for no particular reason).

`/tikz/ball color=<color>`

(no default)

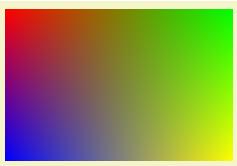
This option sets the color used for the ball shading. It sets the `shade` and `shading=ball` options. Note that the ball will never “completely” have the color `<color>`. At its “highlight” spot a certain amount of white is mixed in, at the border a certain amount of black. Because of this, it also makes sense to say `ball color=white` or `ball color=black`



```
\usepgflibrary {shadings}
\begin{tikzpicture}
  \shade[ball color=white] (0,0) circle (2ex);
  \shade[ball color=red] (1,0) circle (2ex);
  \shade[ball color=black] (2,0) circle (2ex);
\end{tikzpicture}
```

Shading bilinear interpolation

This shading fills a rectangle with colors that are bilinearly interpolated between the colors in the four corners of the rectangle. These four colors are called `lower left`, `lower right`, `upper left`, and `upper right`. By changing these colors, you can change the way the shading looks. The library also defines four options, called the same way, that can be used to set these colors and select the shading implicitly.



```
\usepgflibrary {shadings}
\begin{tikzpicture}
  \shade[upper left=red,upper right=green,
         lower left=blue,lower right=yellow]
    (0,0) rectangle (3,2);
\end{tikzpicture}
```

`/tikz/lower left=<color>`

(no default, initially `white`)

Sets the color to be used in a `bilinear interpolation` shading for the lower left corner. Also, this option selects this shading and sets the `shade` option.

`/tikz/upper left=<color>`

(no default, initially `white`)

Works like `lower left`.

`/tikz/upper right=<color>`

(no default, initially `white`)

Works like `lower left`.

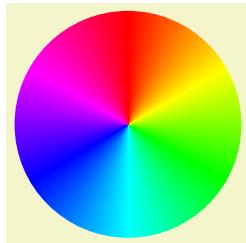
`/tikz/lower right=<color>`

(no default, initially `white`)

Works like `lower left`.

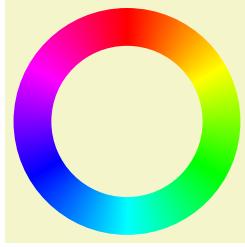
Shading color wheel

This shading fills the path with a color wheel.



```
\usepgflibrary {shadings}
\begin{tikzpicture}
  \shade[shading=color wheel] (0,0) circle (1.5);
\end{tikzpicture}
```

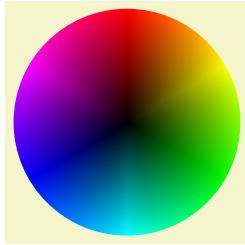
To produce a color ring, cut out a circle from the color wheel:



```
\usepgflibrary {shadings}
\begin{tikzpicture}
\shade [shading=color wheel] (0,0) circle (1.5);
\end{tikzpicture}
```

Shading `color wheel black center`

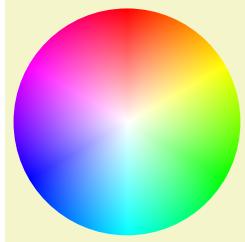
This shading looks like a color wheel, but the brightness drops to zero in the center.



```
\usepgflibrary {shadings}
\begin{tikzpicture}
\shade [shading=color wheel black center] (0,0) circle (1.5);
\end{tikzpicture}
```

Shading `color wheel white center`

This shading looks like a color wheel, but the saturation drops to zero in the center.



```
\usepgflibrary {shadings}
\begin{tikzpicture}
\shade [shading=color wheel white center] (0,0) circle (1.5);
\end{tikzpicture}
```

Shading `Mandelbrot set`

This shading is just for fun. It fills the path with a zoomable Mandelbrot set. Note that this is *not* a bitmap graphic. Rather, the Mandelbrot set is *computed by the PDF renderer* and can be zoomed arbitrarily (give it a try, if you have a fast computer). [Compilation disabled. Please use the lua manual.]

```
\begin{tikzpicture}
\draw [shading=Mandelbrot set] (0,0) rectangle (2,2);
\end{tikzpicture}
```

Shading `radial`

This always-defined shading fills the path with a gradual sweep from a certain color in the middle to another color at the border. If the path is a circle, the outer color will be reached exactly at the border. If the shading is not a circle, the outer color will continue a bit towards the corners. The default inner color is gray, the default outer color is white.

`/tikz/inner color=(color)` (no default)

This option sets the color used at the center of a `radial` shading. When this option is used, the `shade` and `shading=radial` options are set.



```
\usepgflibrary {shadings}
\begin{tikzpicture}
\draw [inner color=red] (0,0) rectangle (2,1);
\end{tikzpicture}
```

`/tikz/outer color=<color>`

(no default)

This option sets the color used at the border and outside of a `radial` shading.



```
\usepgflibrary {shadings}
\tikz \draw[outer color=red,inner color=white]
(0,0) rectangle (2,1);
```

71 Shadows Library

TikZ Library `shadows`

```
\usetikzlibrary{shadows} % LATEX and plain TEX  
\usetikzlibrary[shadows] % ConTEX
```

This library defines styles that help adding a (partly) transparent shadow to a path or node.

71.1 Overview

A *shadow* is usually a black or gray area that is drawn behind a path or a node, thereby adding visual depth to a picture. The `shadows` library defines options that make it easy to add shadows to paths. Internally, these options are based on using the `preaction` option to use a path twice: Once for drawing the shadow (slightly shifted) and once for actually using the path.

Note that you can only add shadows to *paths*, not to whole scopes.

In addition to the general `shadow` option, there exist special options like `circular shadow`. These can only (sensibly) be used with a special kind of path (for `circular shadow`, a circle) and, thus, they are not as general. The advantage is, however, that they are more visually pleasing since these shadows blend smoothly with the background. Note that these special shadows use fadings, which few printers will support.

71.2 The General Shadow Option

The shadows are internally created by using a single option called `general shadow`. The different options like `drop shadow` or `copy shadow` only differ in the commands that they preset.

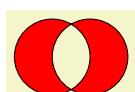
You will not need to use this option directly under normal circumstances.

`/tikz/general shadow=<shadow options>` (default empty)

This option should be given to a `\path` or a `node`. It has the following effect: Before the path is used normally, it is used once with the `<shadow options>` in force. Furthermore, when the path is “preused” in this way, it is shifted and scaled a little bit.

In detail, the following happens: A `preaction` is used to paint the path in a special manner before it is actually painted. This “special” manner is as follows: The options in `<shadow options>` are used for painting this path. Typically, the `<shadow options>` will contain options like `fill=black` to create, say, a black shadow. Furthermore, after the `<shadow options>` have been set up, the following extra canvas transformations are applied to the path: It is scaled by `shadow scale` (with the origin of scaling at the path’s center) and it is shifted by `shadow xshift` and `shadow yshift`.

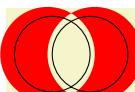
Note that since scaling and shifting is done using canvas transformations, shadows are not taken into account when the picture’s bounding box is computed.



```
\usetikzlibrary {shadows}  
\tikz [even odd rule]  
 \draw [general shadow={fill=red}] (0,0) circle (.5) (0.5,0) circle (.5);
```

`/tikz/shadow scale=<factor>` (no default, initially 1)

Shadows are scaled by `<factor>`.



```
\usetikzlibrary {shadows}  
\tikz [even odd rule]  
 \draw [general shadow={fill=red,shadow scale=1.25}]  
 (0,0) circle (.5) (0.5,0) circle (.5);
```

`/tikz/shadow xshift=<dimension>` (no default, initially 0pt)

Shadows are shifted horizontally by `<dimension>`.



```
\usetikzlibrary {shadows}  
\tikz [even odd rule]  
 \draw [general shadow={fill=red,shadow xshift=-5pt}]  
 (0,0) circle (.5) (0.5,0) circle (.5);
```

`/tikz/shadow yshift=<dimension>` (no default, initially 0pt)
 Shadows are shifted vertically by `<dimension>`.

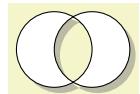
71.3 Shadows for Arbitrary Paths and Shapes

71.3.1 Drop Shadows

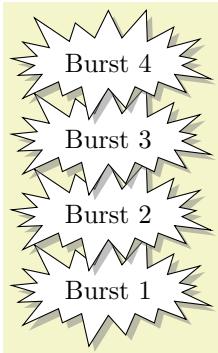
`/tikz/drop shadow=<shadow options>` (default empty)

This option adds a drop shadow to a `\path` or a `node`. It uses the `general shadow` and passes the `<shadow options>` to it, plus, before them, the following extra options:

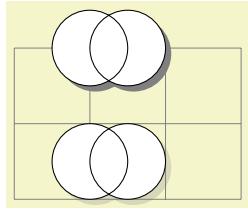
```
shadow scale=1, shadow xshift=.5ex, shadow yshift=-.5ex,
opacity=.5, fill=black!50, every shadow
```



```
\usetikzlibrary {shadows}
\tikz [even odd rule]
\filldraw [drop shadow,fill=white] (0,0) circle (.5) (0.5,0) circle (.5);
```



```
\usetikzlibrary {shadows,shapes.symbols}
\begin{tikzpicture}
\foreach \i in {1,...,4}
\node[starburst,drop shadow,fill=white,draw] at (0,\i) {Burst \i};
\end{tikzpicture}
```

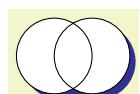


```
\usetikzlibrary {shadows}
\begin{tikzpicture}
\draw [help lines] (0,0) grid (3,2);
\filldraw [drop shadow={opacity=1},fill=white]
(1,2) circle (.5) (1.5,2) circle (.5);

\filldraw [drop shadow={opacity=0.25},fill=white]
(1,.5) circle (.5) (1.5,.5) circle (.5);
\end{tikzpicture}
```

`/tikz/every shadow` (style, initially empty)

This style is executed in addition to any `<shadow options>` for each shadow. Use this style to reconfigure the way shadows are drawn.



```
\usetikzlibrary {shadows}
\begin{tikzpicture}[every shadow/.style={opacity=.8,fill=blue!50!black}]
\filldraw [drop shadow,fill=white] (0,0) circle (.5) (0.5,0) circle (.5);
\end{tikzpicture}
```

71.3.2 Copy Shadows

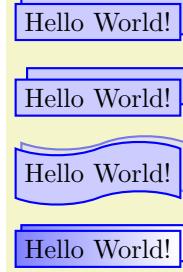
A *copy shadow* is not really a shadow. Rather, it looks like another copy of the path drawn behind the path and a little bit offset. This creates the visual impression of having multiple copies of the path/object present.

`/tikz/copy shadow=<shadow options>` (default empty)

This shadow installs the following default options:

```
shadow scale=1, shadow xshift=.5ex, shadow yshift=-.5ex, every shadow
```

Furthermore, the options `fill=<fill color>` and `draw=<draw color>` are also set, where the `<fill color>` and `<draw color>` are the fill and draw colors used for the main path.



```
\usetikzlibrary {shadows,shapes.symbols}
\begin{tikzpicture}
\node [copy shadow,fill=blue!20,draw=blue,thick] {Hello World!};

\node at (0,-1) [copy shadow={shadow xshift=1ex,shadow yshift=1ex},
                fill=blue!20,draw=blue,thick]
{Hello World!};

\node at (0,-2) [copy shadow={opacity=.5},tape,
                fill=blue!20,draw=blue,thick]
{Hello World!};

% We have to repeat the left color since shadings are not
% automatically applied to shadows
\node at (0,-3) [copy shadow={left color=blue!50},
                left color=blue!50,draw=blue,thick]
{Hello World!};
\end{tikzpicture}
```

`/tikz/double copy shadow=<shadow options>`

(default empty)

This shadow works like a `copy shadow`, only the shadow is added twice, the second time with the double `xshift` and `yshift`.



```
\usetikzlibrary {shadows,shapes.symbols}
\begin{tikzpicture}
\node [double copy shadow,fill=blue!20,draw=blue,thick] {Hello World!};

\node at (0,-1) [double copy shadow={shadow xshift=1ex,shadow yshift=1ex},
                fill=blue!20,draw=blue,thick]
{Hello World!};

\node at (0,-2) [double copy shadow={opacity=.5},tape,
                fill=blue!20,draw=blue,thick]
{Hello World!};

\node at (0,-3) [double copy shadow={left color=blue!50},
                left color=blue!50,draw=blue,thick]
{Hello World!};
\end{tikzpicture}
```

71.4 Shadows for Special Paths and Nodes

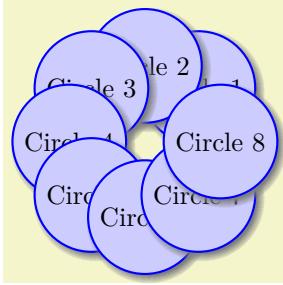
The shadows in this section should normally be added only to paths that have a special shape. They will look strange with other shapes.

`/tikz/circular drop shadow=<shadow options>`

(no default)

This shadow works like a drop shadow, only it adds a circular fading to the shadow. This means that the shadow will fade out at the border. The following options are preset for this shadow:

```
shadow scale=1.1, shadow xshift=.3ex, shadow yshift=-.3ex,
fill=black, path fading={circle with fuzzy edge 15 percent},
every shadow,
```



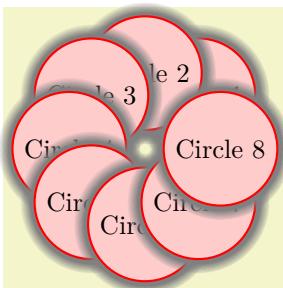
```
\usetikzlibrary {shadows}
\begin{tikzpicture}
\foreach \i in {1,...,8}
\node[circle,circular drop shadow,draw=blue,fill=blue!20,thick]
at (\i*45:1) {Circle \i};
\end{tikzpicture}
```

`/tikz/circular glow=<shadow options>`

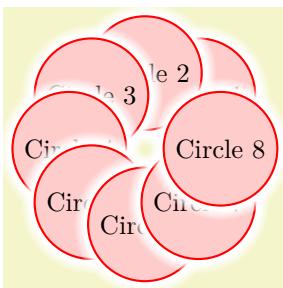
(no default)

This shadow works much like the `circular shadow`, only it is not shifted. This creates a visual effect of a “glow” behind the circle. The following options are preset for this shadow:

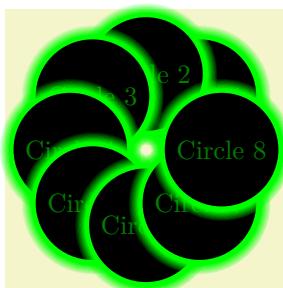
```
shadow scale=1.25, shadow xshift=0pt, shadow yshift=0pt,
fill=black, path fading={circle with fuzzy edge 15 percent},
every shadow,
```



```
\usetikzlibrary {shadows}
\begin{tikzpicture}
\foreach \i in {1,...,8}
\node[circle,circular glow,fill=red!20,draw=red,thick]
at (\i*45:1) {Circle \i};
\end{tikzpicture}
```

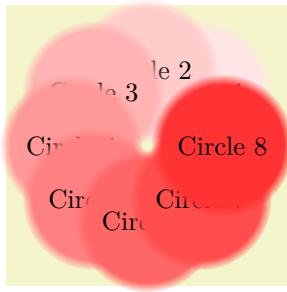


```
\usetikzlibrary {shadows}
\begin{tikzpicture}
\foreach \i in {1,...,8}
\node[circle,circular glow={fill=white},fill=red!20,draw=red,thick]
at (\i*45:1) {Circle \i};
\end{tikzpicture}
```



```
\usetikzlibrary {shadows}
\begin{tikzpicture}
\foreach \i in {1,...,8}
\node[circle,circular glow={fill=green},fill=black,text=green!50!black]
at (\i*45:1) {Circle \i};
\end{tikzpicture}
```

An especially interesting effect can be achieved by only using the glow and not filling the path:



```
\usetikzlibrary {shadows}
\begin{tikzpicture}
  \foreach \i in {1,...,8}
    \node[circle,circular glow={fill=red!\i0}]
      at (\i*45:1) {Circle \i};
\end{tikzpicture}
```

72 Shape Library

72.1 Overview

In addition to the standard shapes `rectangle`, `circle` and `coordinate`, there exist a number of additional shapes defined in different shape libraries. Most of these shapes have been contributed by Mark Wibrow. In the present section, these shapes are described. Note that the library `shapes` is provided for compatibility only. Please include sublibraries like `shapes.geometric` or `shapes.misc` directly.

The appearance of shapes is influenced by numerous parameters like `minimum height` or `inner xsep`. These general parameters are documented in Section 17.2.3

In all of the examples presented in this section, the following `shape example` style is used:

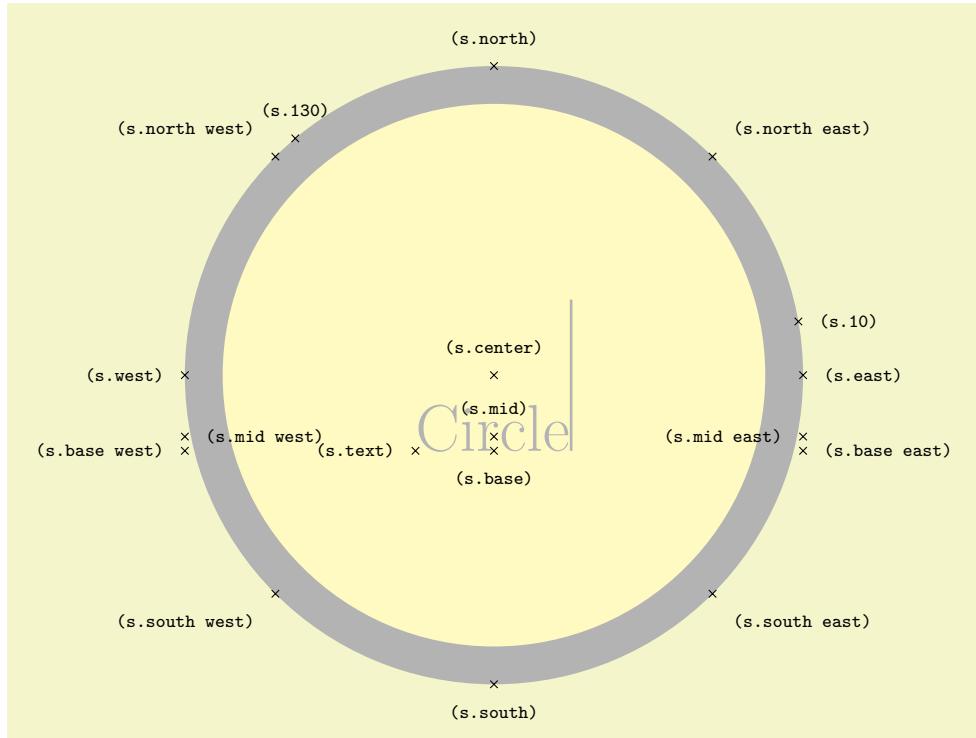
```
\tikzset{
  shape example/.style= {color      = black!30,
                        draw,
                        fill       = yellow!30,
                        line width = .5cm,
                        inner xsep = 2.5cm,
                        inner ysep = 0.5cm}
}
```

72.2 Predefined Shapes

The three shapes `rectangle`, `circle`, and `coordinate` are always defined and no library needs to be loaded for them. While the `coordinate` shape defines only the `center` anchor, the other two shapes define a standard set of anchors.

Shape `circle`

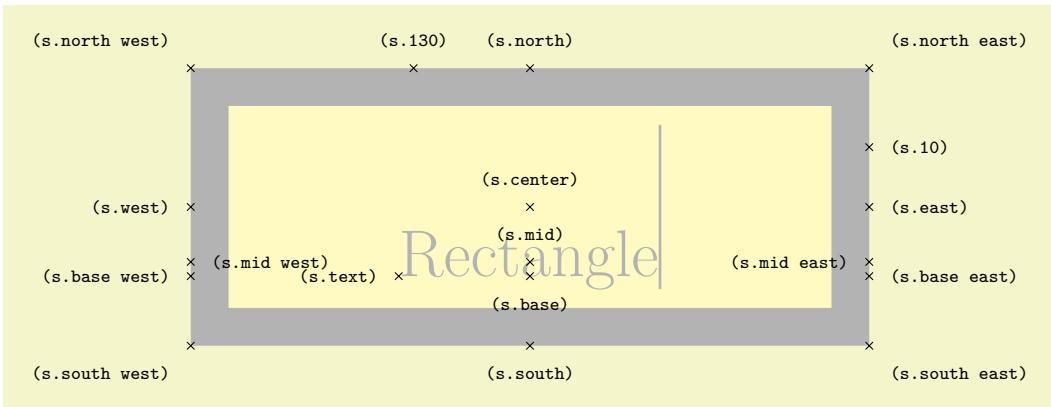
This shape draws a tightly fitting circle around the text. The following figure shows the anchors this shape defines; the anchors 10 and 130 are example of border anchors.



```
\usetikzlibrary {shapes.geometric}
\Huge
\begin{tikzpicture}
  \node[name=s,shape=circle,shape example] {Circle\vrule width 1pt height 2cm};
  \foreach \anchor/\placement in
  {north west/above left, north/above, north east/above right,
   west/left, center/above, east/right,
   mid west/right, mid/above, mid east/left,
   base west/left, base/below, base east/right,
   south west/below left, south/below, south east/below right,
   text/left, 10/right, 130/above}
    \draw[shift=(s.\anchor)] plot[mark=x] coordinates{(0,0)};
    node[\placement] {\scriptsize\textrm{\texttt{(s.\anchor)}}};
\end{tikzpicture}
```

Shape `rectangle`

This shape, which is the standard, is a rectangle around the text. The inner and outer separations (see Section 17.2.3) influence the white space around the text. The following figure shows the anchors this shape defines; the anchors 10 and 130 are example of border anchors.



```
\usetikzlibrary {shapes.geometric}
\Huge
\begin{tikzpicture}
  \node[name=s,shape=rectangle,shape example] {Rectangle\vrule width 1pt height 2cm};
  \foreach \anchor/\placement in
  {north west/above left, north/above, north east/above right,
   west/left, center/above, east/right,
   mid west/right, mid/above, mid east/left,
   base west/left, base/below, base east/right,
   south west/below left, south/below, south east/below right,
   text/left, 10/right, 130/above}
    \draw[shift=(s.\anchor)] plot[mark=x] coordinates{(0,0)};
    node[\placement] {\scriptsize\textrm{\texttt{(s.\anchor)}}};
\end{tikzpicture}
```

72.3 Geometric Shapes

TikZ Library `shapes.geometric`

```
\usepgflibrary{shapes.geometric} % LATEX and plain TEX and pure pgf
\usepgflibrary[shapes.geometric] % ConTEXt and pure pgf
\usetikzlibrary{shapes.geometric} % LATEX and plain TEX when using TikZ
\usetikzlibrary[shapes.geometric] % ConTEXt when using TikZ
```

This library defines different shapes that correspond to basic geometric objects like ellipses or polygons.

Shape `diamond`

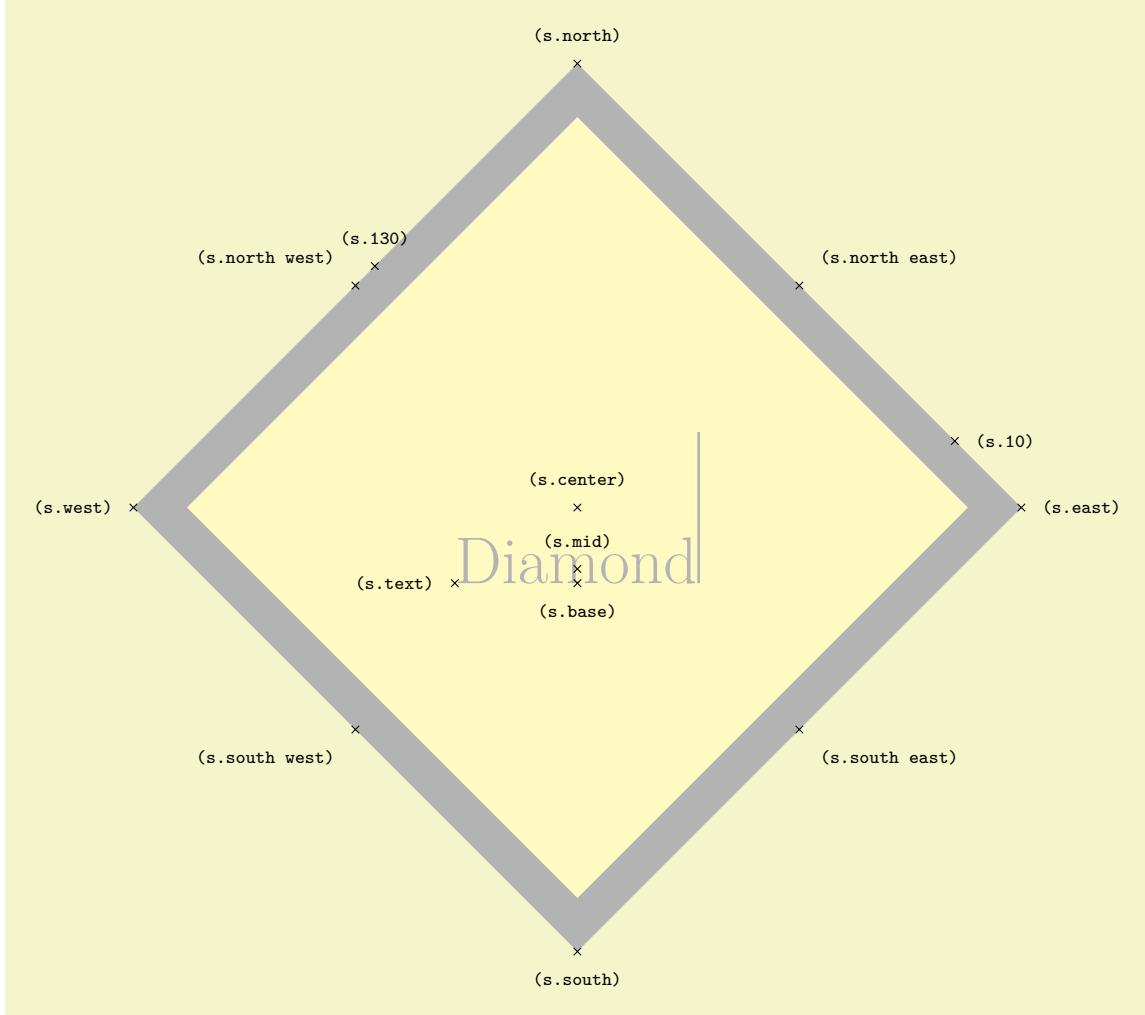
This shape is a diamond tightly fitting the text box. The ratio between width and height is 1 by default, but can be changed by setting the shape aspect ratio using the following PGF key (to use this key in TikZ simply remove the `/pgf/` path).

`/pgf/aspect=(value)`

(no default, initially 1.0)

The aspect is a recommendation for the quotient of the width and the height of a shape. This key calls the macro `\pgfsetshapeaspect`.

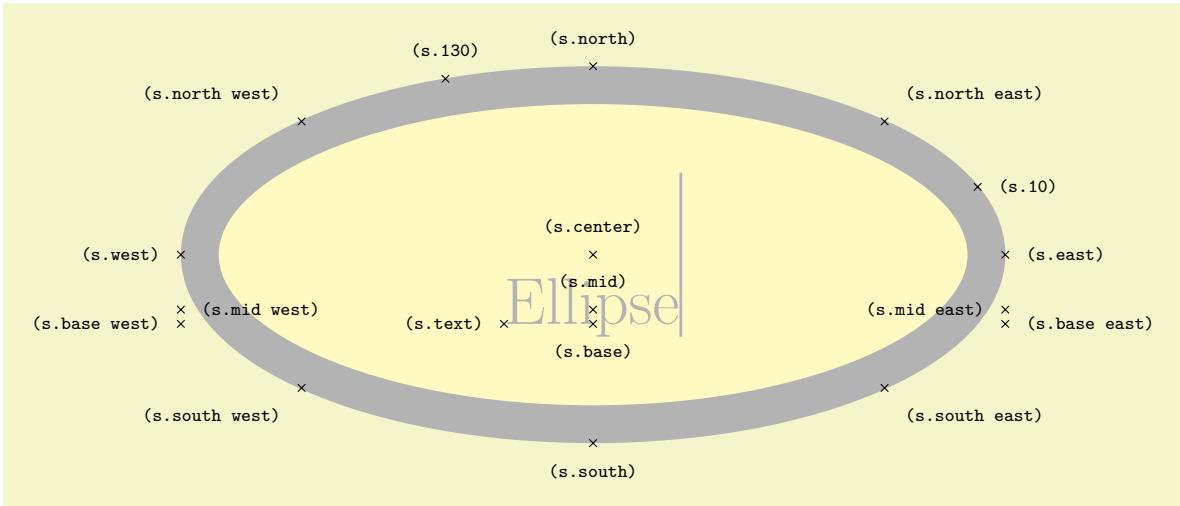
The following figure shows the anchors this shape defines; the anchors 10 and 130 are example of border anchors.



```
\usetikzlibrary {shapes.geometric}
\Huge
\begin{tikzpicture}
\node[name=s,shape=diamond,shape example] {Diamond \vrule width 1pt height 2cm};
\foreach \anchor/\placement in
{north west/above left, north/above, north east/above right,
west/left, center/above, east/right,
mid/above,
base/below,
south west/below left, south/below, south east/below right,
text/left, 10/right, 130/above}
\draw[shift=(s.\anchor)] plot[mark=x] coordinates{(0,0)};
\node[\placement] {\scriptsize\textrm{\texttt{(s.\anchor)}}};
\end{tikzpicture}
```

Shape `ellipse`

This shape is an ellipse tightly fitting the text box, if no inner separation is given. The following figure shows the anchors this shape defines; the anchors 10 and 130 are example of border anchors.

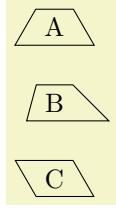


```
\usetikzlibrary {shapes.geometric}
\Huge
\begin{tikzpicture}
\node [name=s,shape=ellipse,shape example] {Ellipse\vrule width 1pt height 2cm};
\foreach \anchor/\placement in
{north west/above left, north/above, north east/above right,
west/left, center/above, east/right,
mid west/right, mid/above, mid east/left,
base west/left, base/below, base east/right,
south west/below left, south/below, south east/below right,
text/left, 10/right, 130/above}
\draw[shift=(s.\anchor)] plot[mark=x] coordinates{(0,0)};
\node[\placement] {\scriptsize\textrm{\texttt{(s.\anchor)}}};
\end{tikzpicture}
```

Shape trapezium

This shape is a trapezium, that is, a quadrilateral with a single pair of parallel lines (this can sometimes be known as a trapezoid). The trapezium shape supports the rotation of the shape border, as described in Section 17.2.3.

The lower internal angles at the lower corners of the trapezium can be specified independently, and the resulting extensions are in addition to the natural dimensions of the node contents (which includes any `inner sep`.



```
\usetikzlibrary {shapes.geometric}
\begin{tikzpicture}[every node/.style={trapezium, draw}]
\node at (0,2) {A};
\node[trapezium left angle=75, trapezium right angle=45]
at (0,1) {B};
\node[trapezium left angle=120, trapezium right angle=60]
at (0,0) {C};
\end{tikzpicture}
```

The PGF keys to set the lower internal angles of the trapezium are shown below. To use these keys in TikZ, simply remove the `/pgf/` path.

`/pgf/trapezium left angle=<angle>` (no default, initially 60)

Sets the lower internal angle of the left side.

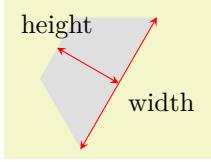
`/pgf/trapezium right angle=<angle>` (no default, initially 60)

Sets the lower internal angle of the right side.

`/pgf/trapezium angle=<angle>` (style, no default)

This key stores no value itself, but sets the value of the previous two keys to `<angle>`.

Regardless of the rotation of the shape border, the width and height of the trapezium are as follows:

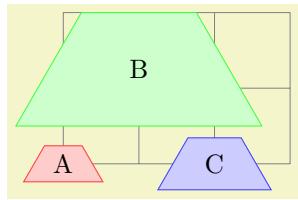


```
\usetikzlibrary {shapes.geometric}
\begin{tikzpicture} [>=stealth, every node/.style={text=black},
    shape border uses incircle, shape border rotate=60]
\node [trapezium, fill=gray!25, minimum width=2cm] (t) {};
\draw [red, <->] (t.bottom left corner) -- (t.bottom right corner)
    node [midway, below right] {width};
\draw [red, <->] (t.top side) -- (t.bottom side)
    node [at start, above] {height};
\end{tikzpicture}
```

`/pgf/trapezium stretches=(boolean)`

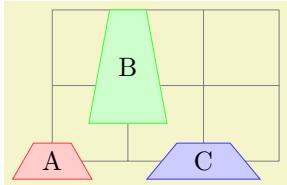
(default `true`)

This key controls whether PGF allows the width and the height of the trapezium to be enlarged independently, when considering any minimum size specification. This is initially `false`, ensuring that the shape “looks the same but bigger” when enlarged.



```
\usetikzlibrary {shapes.geometric}
\tikzset{my node/.style={trapezium, fill=#1!20, draw=#1!75, text=black}}
\begin{tikzpicture}
\draw [help lines] grid (3,2);
\node [my node=red] {A};
\node [my node=green, minimum height=1.5cm] at (1, 1.25) {B};
\node [my node=blue, minimum width=1.5cm] at (2, 0) {C};
\end{tikzpicture}
```

By setting `<boolean>` to `true`, the trapezium can be stretched horizontally or vertically.

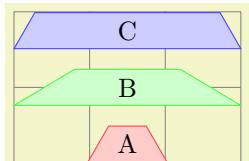


```
\usetikzlibrary {shapes.geometric}
\tikzset{my node/.style={trapezium, fill=#1!20, draw=#1!75, text=black}}
\tikzset{trapezium stretches=true}
\begin{tikzpicture}
\draw [help lines] grid (3,2);
\node [my node=red] {A};
\node [my node=green, minimum height=1.5cm] at (1, 1.25) {B};
\node [my node=blue, minimum width=1.5cm] at (2, 0) {C};
\end{tikzpicture}
```

`/pgf/trapezium stretches body=(boolean)`

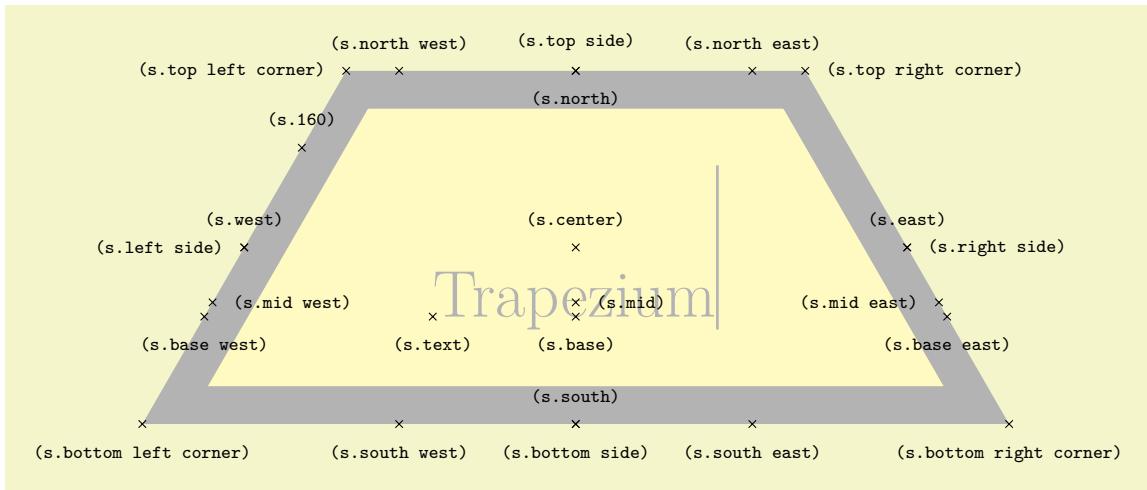
(default `true`)

This is similar to the `trapezium stretches` key except that when `<boolean>` is `true`, PGF enlarges only the body of the trapezium when applying minimum width.



```
\usetikzlibrary {shapes.geometric}
\tikzset{my node/.style={trapezium, fill=#1!20, draw=#1!75, text=black}}
\begin{tikzpicture}
\draw [help lines] grid (3,2);
\node [my node=red] at (1.5,.25) {A};
\node [my node=green, minimum width=3cm, trapezium stretches]
    at (1.5,1) {B};
\node [my node=blue, minimum width=3cm, trapezium stretches body]
    at (1.5,1.75) {C};
\end{tikzpicture}
```

The anchors for the trapezium are shown below. The anchor 160 is an example of a border anchor.



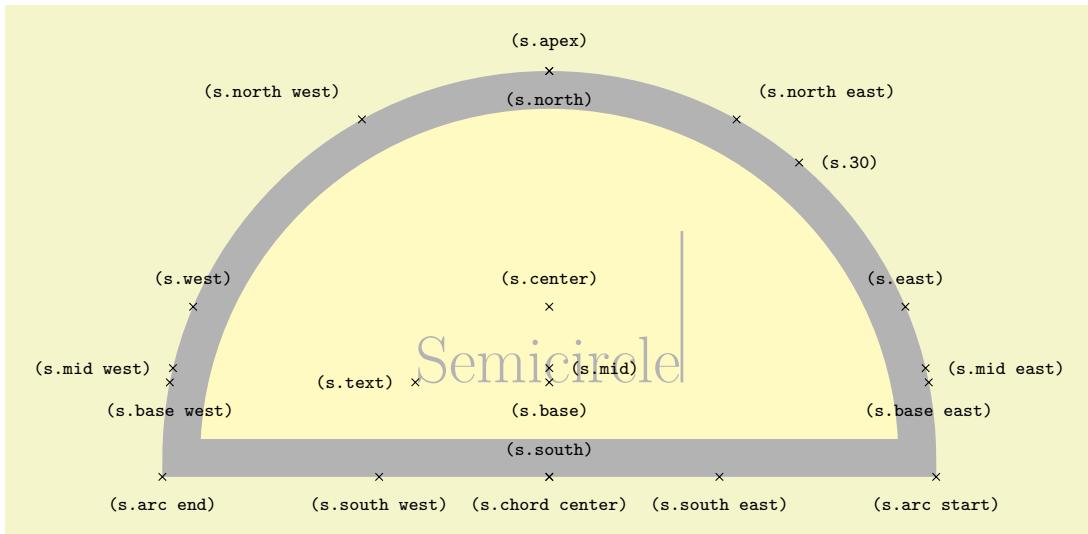
```

\usetikzlibrary {shapes.geometric}
\Huge
\begin{tikzpicture}
\node[name=s, shape=trapezium, shape example, inner sep=1cm]
  {Trapezium\vrule width 1pt height 2cm};
\foreach \anchor/\placement in
  {bottom left corner/below, top right corner/right,
   top left corner/left, bottom right corner/below,
   bottom side/below, left side/left,
   right side/right, top side/above,
   center/above, text/below, mid/right, base/below,
   mid west/right, base west/below, mid east/left, base east/below,
   west/above, east/above, north/below, south/above,
   north west/above, north east/above,
   south west/below, south east/below, 160/above}
\draw[shift=(s.\anchor)] plot[mark=x] coordinates{(0,0)}
  node[\placement] {\scriptsize\textrm{\texttt{(s.\anchor)}}};
\end{tikzpicture}

```

Shape semicircle

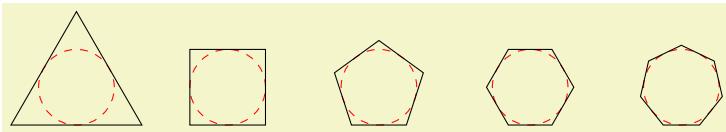
This shape is a semicircle, which tightly fits the node contents. This shape supports the rotation of the shape border, as described in Section 17.2.3. The anchors for the `semicircle` shape are shown below. Anchor 30 is an example of a border anchor.



```
\usetikzlibrary {shapes.geometric}
\Huge
\begin{tikzpicture}
  \node [name=s,shape=semicircle,shape border rotate=0,shape example, inner sep=1cm]
    {Semicircle\vrule width 1pt height 2cm};
  \foreach \anchor/\placement in
    {apex/above, arc start/below, arc end/below, chord center/below,
     center/above, base/below, mid/right, text/left,
     base west/below, base east/below, mid west/left, mid east/right,
     north/below, south/above, east/above, west/above,
     north west/above left, north east/above right,
     south west/below, south east/below, 30/right}
    {\draw[shift=(s.\anchor)] plot[mark=x] coordinates{(0,0)}}
    node[\placement] {\scriptsize\textrm{\texttt{(s.\anchor)}}};
\end{tikzpicture}
```

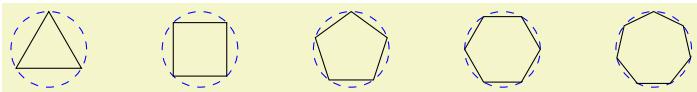
Shape `regular polygon`

This shape is a regular polygon, which, by default, is drawn so that a side (rather than a corner) is always at the bottom. This shape supports the rotation as described in Section 17.2.3, but the border of the polygon is *always* constructed using the incircle, whose radius is calculated to tightly fit the node contents (including any `inner sep`).



```
\usetikzlibrary {shapes.geometric}
\begin{tikzpicture}
  \foreach \a in {3,...,7}{
    \draw[red, dashed] (\a*2,0) circle(0.5cm);
    \node[regular polygon, regular polygon sides=\a, draw,
          inner sep=0.3535cm] at (\a*2,0) {};
  }
\end{tikzpicture}
```

If the node is enlarged to any specified minimum size, this is interpreted as the diameter of the circumcircle, that is, the circle that passes through all the corners of the polygon border.

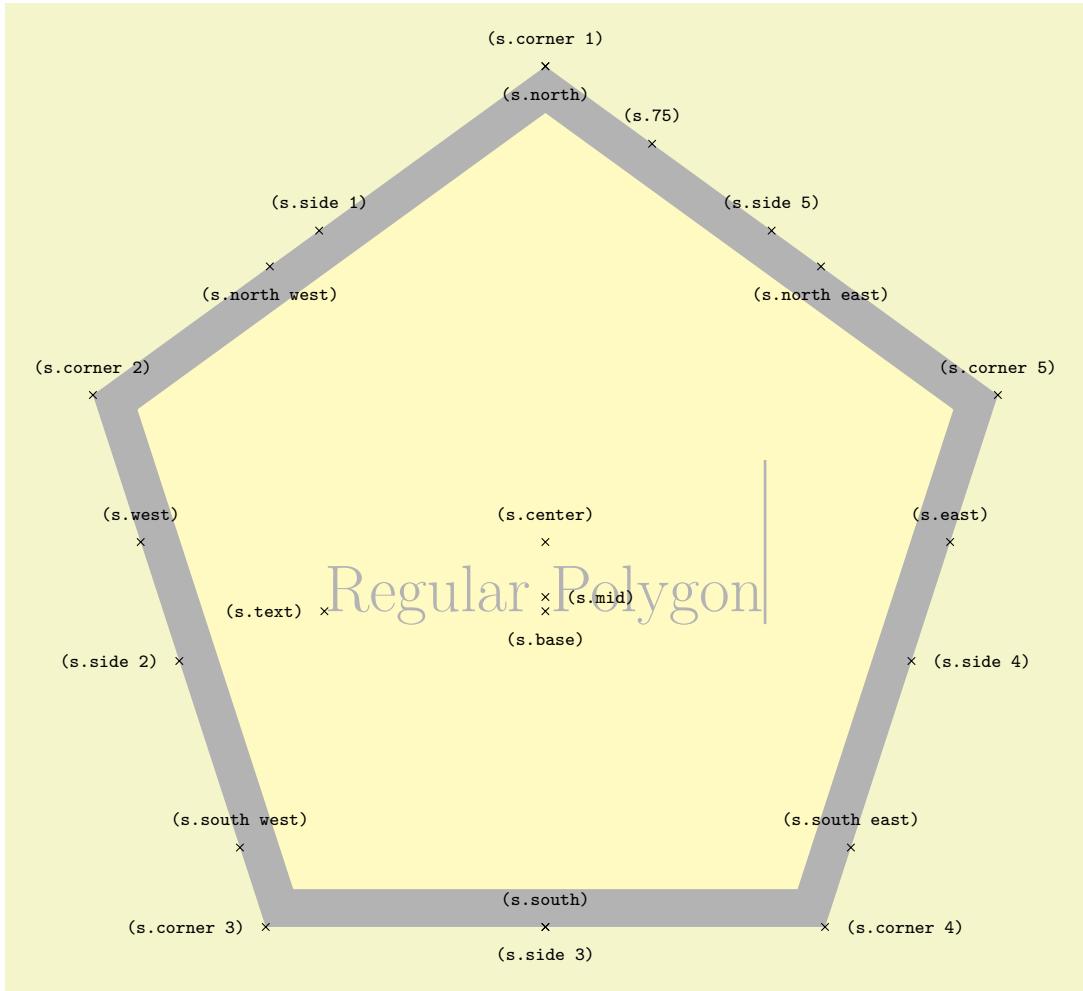


```
\usetikzlibrary {shapes.geometric}
\begin{tikzpicture}
  \foreach \a in {3,...,7}{
    \draw[blue, dashed] (\a*2,0) circle(0.5cm);
    \node[regular polygon, regular polygon sides=\a, minimum size=1cm, draw] at (\a*2,0) {};
  }
\end{tikzpicture}
```

There is a PGF key to set the number of sides for the regular polygon. To use this key in TikZ, simply remove the `/pgf/` path.

`/pgf/regular polygon sides=(integer)` (no default, initially 5)

The anchors for a regular polygon shape are shown below. The anchor 75 is an example of a border anchor.

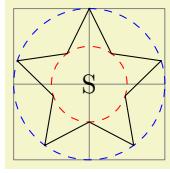


```
\usetikzlibrary {shapes.geometric}
\Huge
\begin{tikzpicture}
\node[name=s, shape=regular polygon, shape example, inner sep=.5cm]
{Regular Polygon \vrule width 1pt height 2cm};
\foreach \anchor/\placement in
{corner 1/above, corner 2/above, corner 3/left, corner 4/right, corner 5/above,
 side 1/above, side 2/left, side 3/below, side 4/right, side 5/above,
 center/above, text/left, mid/right, base/below, 75/above,
 west/above, east/above, north/below, south/above,
 north east/below, south east/above, north west/below, south west/above}
\draw[shift=(s.\anchor)] plot [mark=x] coordinates{(0,0)}
node[\placement] {\scriptsize\textrm{\texttt{(s.\anchor)}}};
\end{tikzpicture}
```

Shape `star`

This shape is a star, which by default (minus any transformations) is drawn with the first point pointing upwards. This shape supports the rotation as described in Section 17.2.3, but the border of the star is always constructed using the incircle.

A star should be thought of as having a set of “inner points” and “outer points”. The inner points of the border are based on the radius of the circle which tightly fits the node contents, and the outer points are based on the circumcircle, the circle that passes through every outer point. Any specified minimum size, width or height, is interpreted as the diameter of the circumcircle.



```
\usetikzlibrary {shapes.geometric}
\begin{tikzpicture}
  \draw [help lines] (0,0) grid (2,2);
  \draw [blue, dashed] (1,1) circle(1cm);
  \draw [red, dashed] (1,1) circle(.5cm);
  \node [star, star point height=.5cm, minimum size=2cm, draw]
    at (1,1) {S};
\end{tikzpicture}
```

The PGF keys to set the number of star points, and the height of the star points, are shown below. To use these keys in TikZ, simply remove the `/pgf/` path.

`/pgf/star points=<integer>` (no default, initially 5)

Sets the number of points for the star.

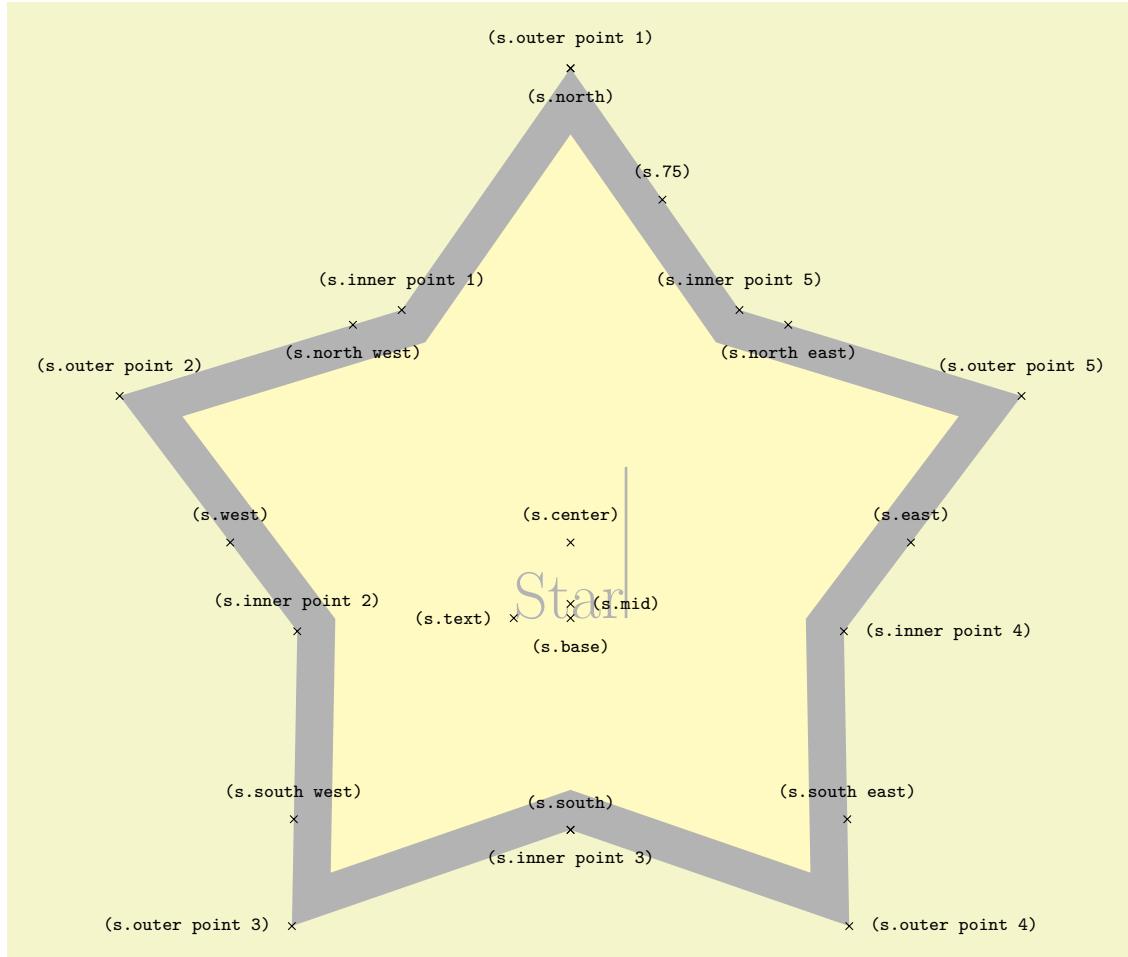
`/pgf/star point height=<distance>` (no default, initially .5cm)

Sets the height of the star points. This is the distance between the inner point and outer point radii. If the star is enlarged to some specified minimum size, the inner radius is increased to maintain the point height.

`/pgf/star point ratio=<number>` (no default, initially 1.5)

Sets the ratio between the inner point and outer point radii. If the star is enlarged to some specified minimum size, the inner radius is increased to maintain the ratio.

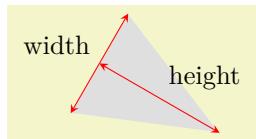
The inner and outer points form the principal anchors for the star, as shown below (anchor 75 is an example of a border anchor).



```
\usetikzlibrary {shapes.geometric}
\Huge
\begin{tikzpicture}
\node [name=s, shape=star, star points=5, star point ratio=1.65, shape example, inner sep=1.5cm]
  {Star\vrule width 1pt height 2cm};
\foreach \anchor/\placement in
  {inner point 1/above, inner point 2/above, inner point 3/below, inner point 4/right,
   inner point 5/above, outer point 1/above, outer point 2/above, outer point 3/left,
   outer point 4/right, outer point 5/above,
   center/above, text/left, mid/right, base/below, 75/above,
   west/above, east/above, north/below, south/above,
   north east/below, south east/above, north west/below, south west/above}
  \draw[shift=(s.\anchor)] plot [mark=x] coordinates{(0,0)};
  node[\placement] {\scriptsize\textrm{\texttt{(s.\anchor)}}};
\end{tikzpicture}
```

Shape `isosceles triangle`

This shape is an isosceles triangle, which supports the rotation of the shape border, as described in Section 17.2.3. The angle of rotation determines the direction in which the apex of the triangle points (provided no other transformations are applied). However, regardless of the rotation of the shape border, the width and height are always considered as follows:



```
\usetikzlibrary {shapes.geometric}
\begin{tikzpicture} [>=stealth, every node/.style={text=black},
  shape border uses incircle, shape border rotate=-30]
\node [isosceles triangle, fill=gray!25, minimum width=1.5cm] (t) {};
\draw [red, <->] (t.left corner) -- (t.right corner)
  node [midway, above left] {width};
\draw [red, <->] (t.apex) -- (t.lower side)
  node [midway, above right] {height};
\end{tikzpicture}
```

There are PGF keys to customize this shape. To use these keys in TikZ, simply remove the `/pgf/` path.

`/pgf/isosceles triangle apex angle=<angle>`

(no default, initially 30)

Sets the angle of the apex of the isosceles triangle.

`/pgf/isosceles triangle stretches=<boolean>`

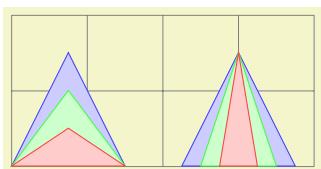
(default `true`)

By default `<boolean>` is `false`. This means, that when applying any minimum width or minimum height requirements, increasing the height will increase the width (and vice versa), in order to keep the apex angle the same.



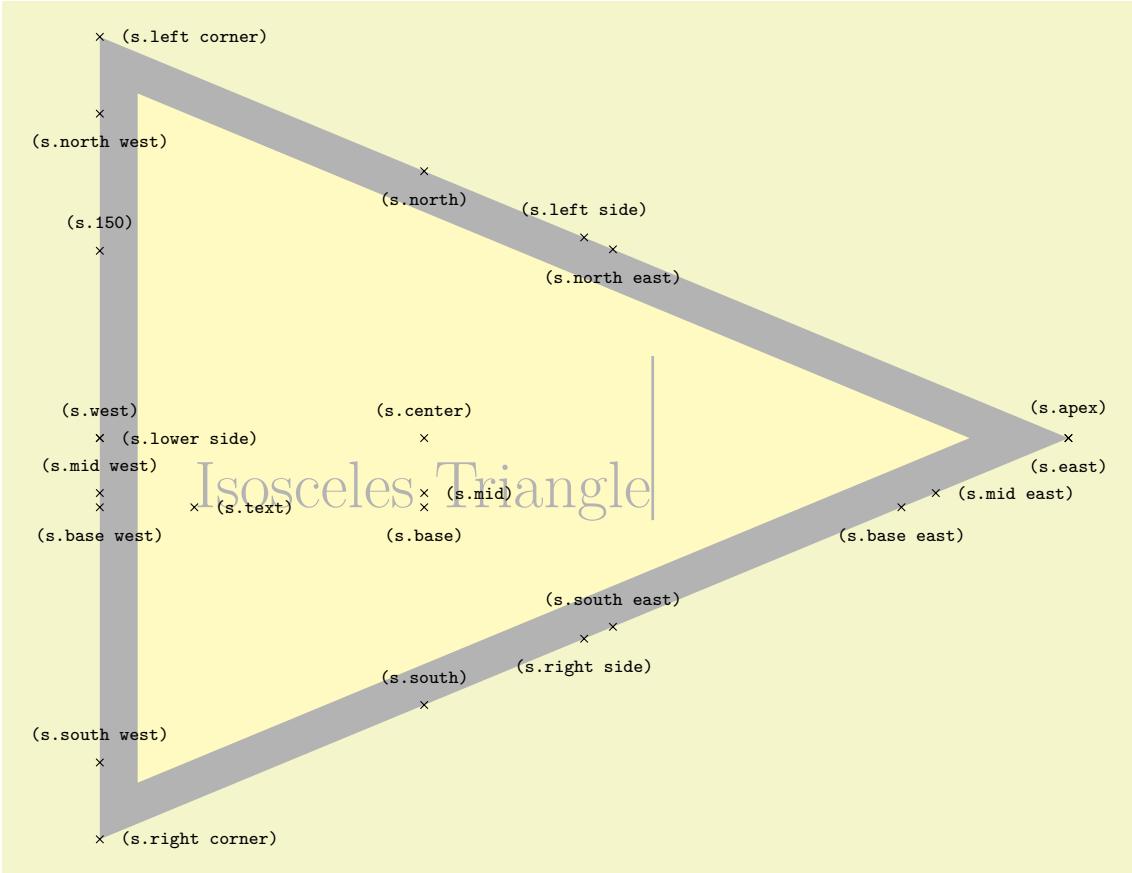
```
\usetikzlibrary {shapes.geometric}
\begin{tikzpicture} [paint/.style={draw=#1!75, fill=#1!20}]
\tikzset{every node/.style={isosceles triangle, draw, inner sep=0pt,
  anchor=left corner, shape border rotate=90}}
\draw [help lines] grid(4,2);
\foreach \a/\c in {1.5/blue, 1/green, 0.5/red}{
  \node[paint=\c, minimum height=\a cm] at (0,0) {};
  \node[paint=\c, minimum width=\a cm] at (2,0) {};
}
\end{tikzpicture}
```

However, by setting `<boolean>` to `true`, minimum width and height can be applied independently.



```
\usetikzlibrary {shapes.geometric}
\begin{tikzpicture} [paint/.style={draw=#1!75, fill=#1!20}]
\tikzset{every node/.style={isosceles triangle, draw, inner sep=0pt,
  anchor=south, shape border rotate=90, isosceles triangle stretches}}
\draw [help lines] grid(4,2);
\foreach \a/\c in {1.5/blue, 1/green, 0.5/red}{
  \node[paint=\c, minimum height=\a cm, minimum width=1.5cm] at (0.75,0) {};
  \node[paint=\c, minimum width=\a cm, minimum height=1.5cm] at (3,0) {};
}
\end{tikzpicture}
```

The anchors for the `isosceles triangle` are shown below (anchor 150 is an example of a border anchor). Note that, somewhat confusingly, the anchor names such as `left side` and `right corner` are named as if the triangle is rotated to 90 degrees. Note also that the `center` anchor does not necessarily correspond to any kind of geometric center.



```
\usetikzlibrary {shapes.geometric}
\Huge
\begin{tikzpicture}
\node[name=s, shape=isosceles triangle, shape example, inner xsep=1cm]
  {Isosceles Triangle};
\foreach \anchor/\placement in
  {apex/above, left corner/right, right corner/right,
   left side/above, right side/below, lower side/right,
   center/above, text/right, 150/above,
   mid/right, mid west/above, mid east/right,
   base/below, base west/below, base east/below,
   west/above, east/below, north/below, south/above,
   north west/below, north east/below,
   south west/above, south east/above}
  \draw[shift=(s.\anchor)] plot [mark=x] coordinates{(0,0)}
    node[\placement] {\scriptsize\tt{(s.\anchor)}};
\end{tikzpicture}
```

Shape `kite`

This shape is a kite, which supports the rotation of the shape border, as described in Section 17.2.3. There are PGF keys to specify the upper and lower vertex angles of the kite. To use these keys in TikZ, simply remove the `/pgf/` path.

`/pgf/kite upper vertex angle=<angle>` (no default, initially 120)

Sets the upper internal angle of the kite.

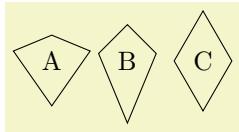
`/pgf/kite lower vertex angle=<angle>` (no default, initially 60)

Sets the lower internal angle of the kite.

/pgf/kite vertex angles={angle specification})

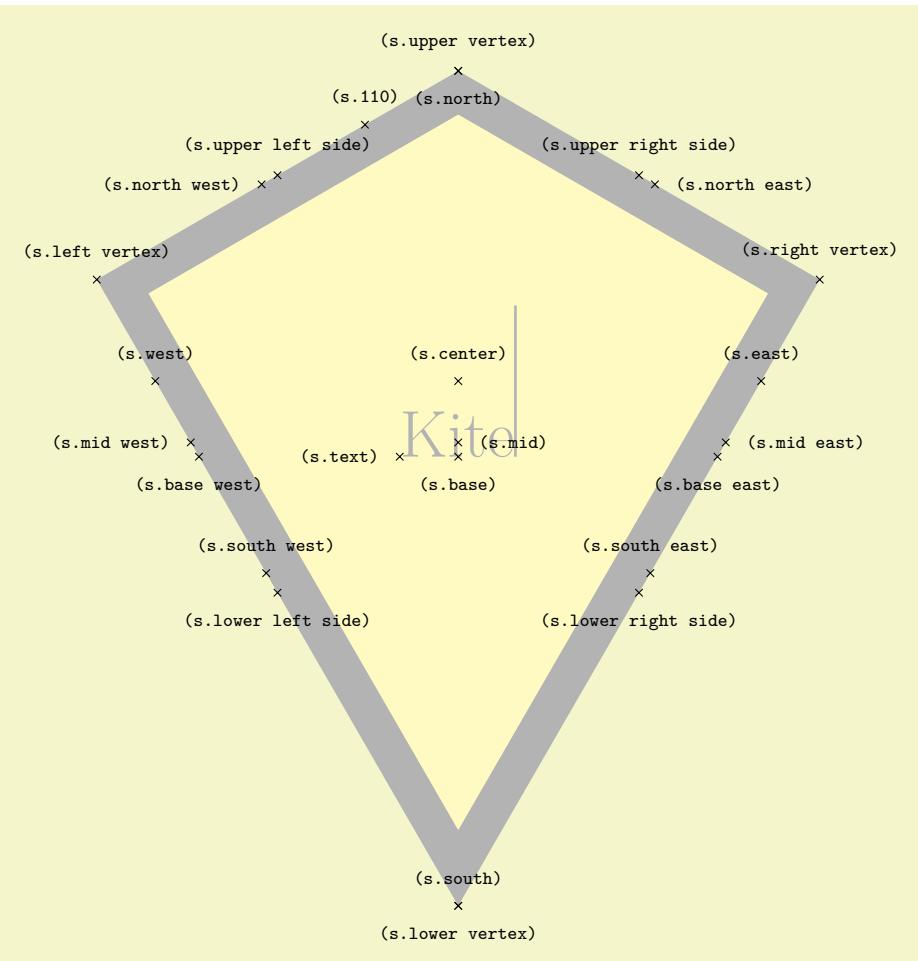
(no default)

This key sets the keys for both the upper and lower vertex angles (it stores no value itself). *<angle specification>* can be pair of angles in the form *<upper angle>* and *<lower angle>*, or a single angle. In this latter case, both the upper and lower vertex angles will be the same.



```
\usetikzlibrary {shapes.geometric}
\begin{tikzpicture}[every node/.style=kite, draw]
  \node[kite upper vertex angle=135, kite lower vertex angle=70] at (0,0) {A};
  \node[kite vertex angles=90 and 45] at (1,0) {B};
  \node[kite vertex angles=60] at (2,0) {C};
\end{tikzpicture}
```

The anchors for the kite are shown below. Anchor 110 is an example of a border anchor.

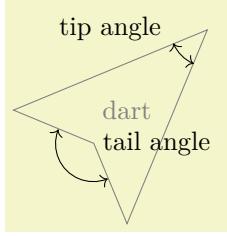


```
\usetikzlibrary {shapes.geometric}
\Huge
\begin{tikzpicture}
  \node[name=s, shape=kite, shape example, inner sep=1.5cm]
    {Kite\vrule width 1pt height 2cm};
  \foreach \anchor/\placement in
    {upper vertex/above, left vertex/above, lower vertex/below,
     right vertex/above, upper left side/above, upper right side/above,
     lower left side/below, lower right side/below,
     center/above, text/left, mid/right, base/below,
     mid west/left, base west/below, mid east/right, base east/below,
     west/above, east/above, north/below, south/above,
     north west/left, north east/right,
     south west/above, south east/above, 110/above}
    \draw[shift=(s.\anchor)] plot [mark=x] coordinates{(0,0)}
      node[\placement] {\scriptsize\tt(s.\anchor)};
\end{tikzpicture}
```

Shape `dart`

This shape is a dart (which can also be known as an arrowhead or concave kite). This shape supports the rotation of the shape border, as described in Section 17.2.3. The angle of the border rotation determines the direction in which the dart points (unless other transformations have been applied).

There are PGF keys to set the angle for the ‘tip’ of the dart and the angle between the ‘tails’ of the dart. To use these keys in TikZ, simply remove the `/pgf/` path.



```
\usetikzlibrary {shapes.geometric}
\begin{tikzpicture}
  \node[dart, draw, gray, shape border uses incircle, shape border rotate=45]
    (d) {dart};
  \draw [->] (d.tip)+(202.5:.5cm) arc(202.5:247.5:.5cm);
  \node [left=.5cm] at (d.tip) {tip angle};
  \draw [->] (d.tail center)+(157.5:.5cm) arc(157.5:292.5:.5cm);
  \node [right] at (d.tail center) {tail angle};
\end{tikzpicture}
```

`/pgf/dart tip angle=<angle>`

(no default, initially 45)

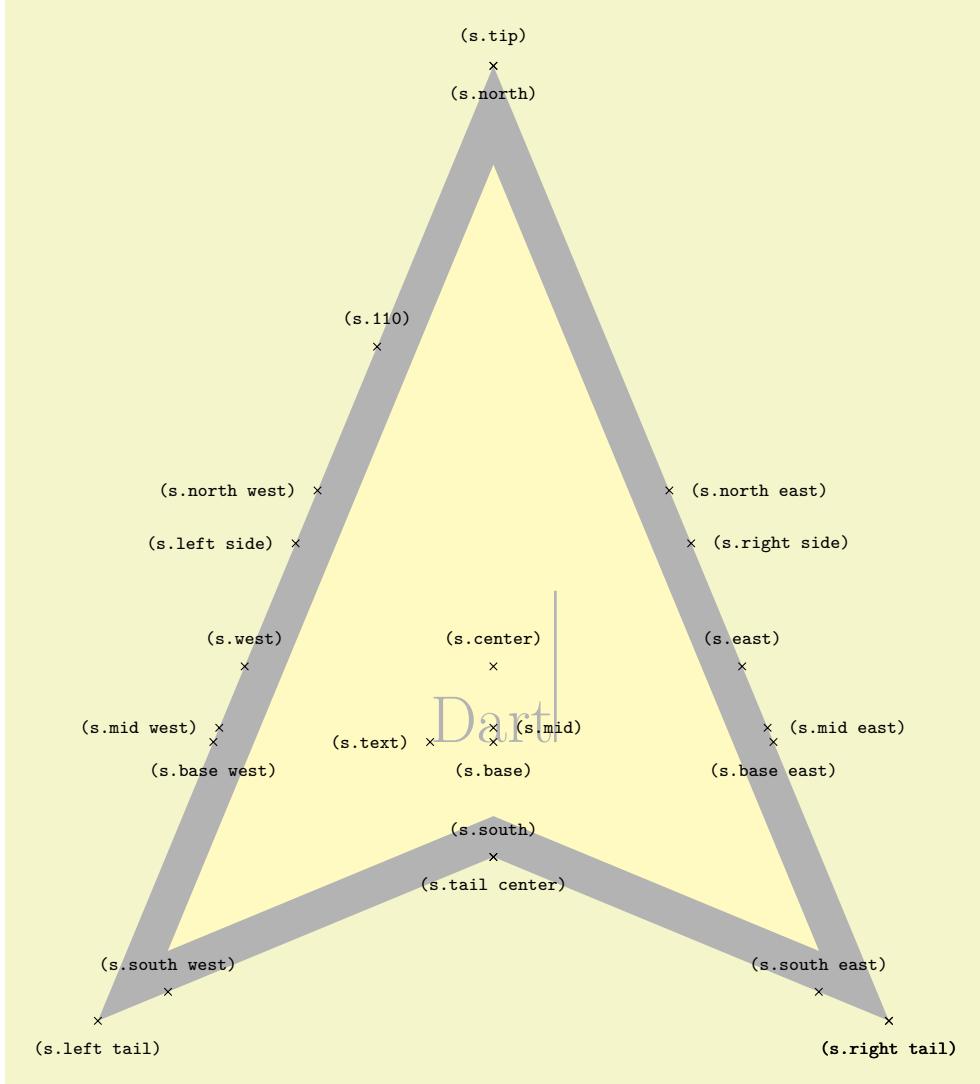
Sets the angle at the tip of the dart.

`/pgf/dart tail angle=<angle>`

(no default, initially 135)

Sets the angle between the tails of the dart.

The anchors for the `dart` shape are shown below (note that the shape is rotated 90 degrees anti-clockwise). Anchor 110 is an example of a border anchor.



```
\usetikzlibrary {shapes.geometric}
\Huge
\begin{tikzpicture}
\node[name=s, shape=dart, shape border rotate=90, shape example, inner sep=1.25cm]
{Dart\vrule width 1pt height 2cm};
\foreach \anchor/\placement in
{tip/above, tail center/below, right tail/below,
 left tail/below, right tail/below, left side/left, right side/right,
 center/above, text/left, mid/right, base/below,
 mid west/left, base west/below, mid east/right, base east/below,
 west/above, east/above, north/below, south/above,
 north west/left, north east/right, south west/above, south east/above,
 110/above}
\draw[shift=(s.\anchor)] plot [mark=x] coordinates{(0,0)}
node[\placement]{\scriptsize\textrm{\texttt{(s.\anchor)}}};
\end{tikzpicture}
```

Shape `circular sector`

This shape is a circular sector (which can also be known as a wedge). This shape supports the rotation of the shape border, as described in Section 17.2.3. The angle of the border rotation determines the direction in which the ‘apex’ of the sector points (unless other transformations have been applied).



```
\usetikzlibrary {shapes.geometric}
\begin{tikzpicture}[
  every node/.style={circular sector, shape border uses incircle, draw},
]
  \node at (0,0) {A};
  \node [shape border rotate=30] at (1.5,0) {A};
\end{tikzpicture}
```

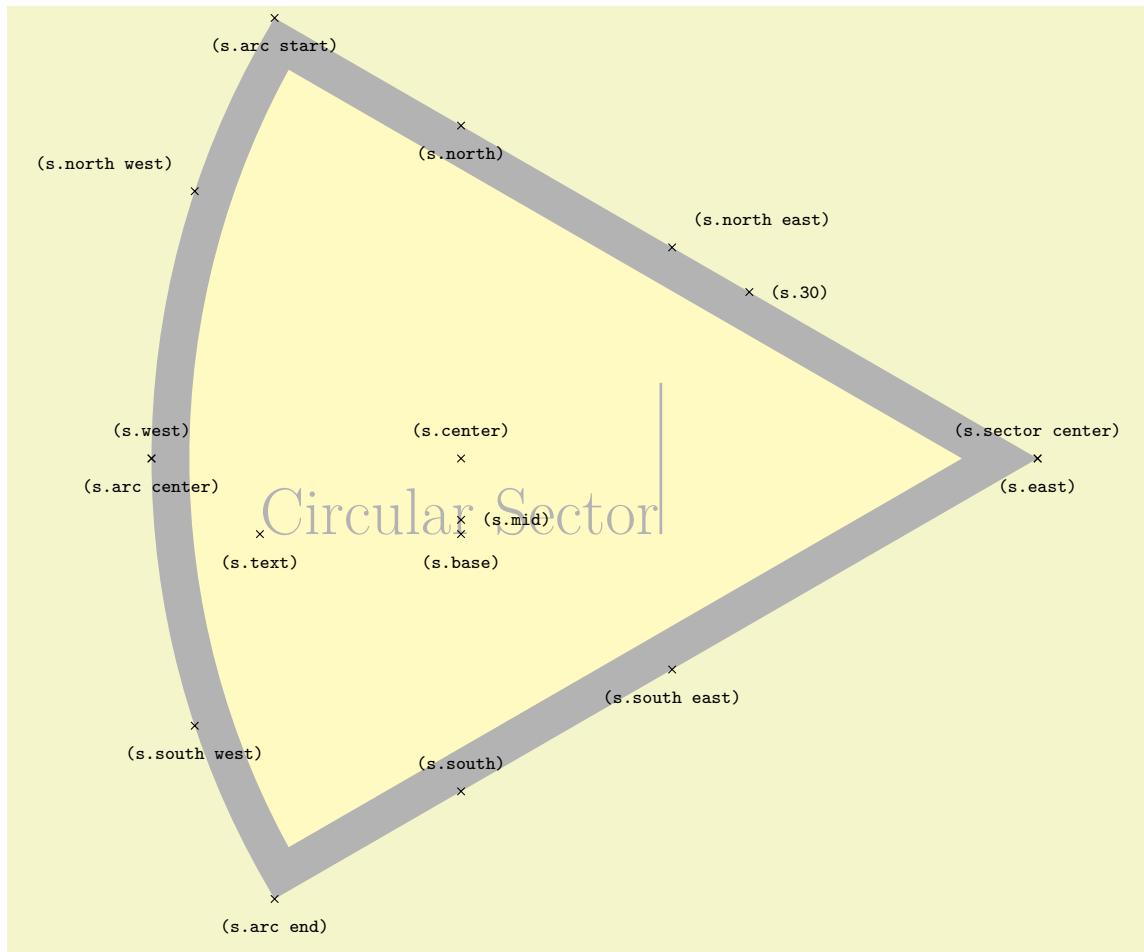
There is a PGF key to set the central angle of the sector, which is expected to be less than 180 degrees. To use this key in TikZ, simply remove the `/pgf/` path.

`/pgf/circular sector angle=<angle>`

(no default, initially 60)

Sets the central angle of the sector.

The anchors for the circular sector shape are shown below. Anchor 30 is an example of a border anchor.



```
\usetikzlibrary {shapes.geometric}
\Huge
\begin{tikzpicture}
  \node[name=s,shape=circular sector, style=shape example, inner sep=1cm]
    {Circular Sector}\rule width 1pt height 2cm;
  \foreach \anchor/\placement in
  {sector center/above, arc start/below, arc end/below, arc center/below,
   center/above, base/below, mid/right, text/below,
   north/below, south/above, east/below, west/above,
   north west/above left, north east/above right,
   south west/below, south east/below, 30/right}
    \draw[shift=(s.\anchor)] plot[mark=x] coordinates{(0,0)};
    \node[\placement] {\scriptsize\texttt{\{s.\anchor\}}};
\end{tikzpicture}
```

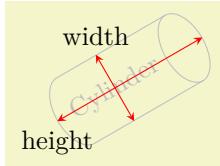
Shape `cylinder`

This shape is a 2-dimensional representation of a cylinder, which supports the rotation of the shape border as described in Section 17.2.3.



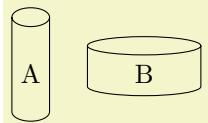
```
\usetikzlibrary {shapes.geometric}
\begin{tikzpicture}
\node [cylinder, draw, shape aspect=.5] {ABC};
\end{tikzpicture}
```

Regardless the rotation of the shape border, the height is always the distance between the curved ends, and the width is always the distance between the straight sides.



```
\usetikzlibrary {shapes.geometric}
\begin{tikzpicture} [>=stealth]
\node [cylinder, gray!50, rotate=30, draw,
minimum height=2cm, minimum width=1cm] (c) {Cylinder};
\draw[red, <->] (c.top) -- (c.bottom)
node [at end, below, black] {height};
\draw[red, <->] (c.north) -- (c.south)
node [at start, above, black] {width};
\end{tikzpicture}
```

Enlarging the shape to some minimum height will stretch only the body of the cylinder. By contrast, enlarging the shape to some minimum width will stretch the curved ends.



```
\usetikzlibrary {shapes.geometric}
\begin{tikzpicture} [shape aspect=.5]
\tikzset{every node/.style={cylinder, shape border rotate=90, draw}}
\node [minimum height=1.5cm] {A};
\node [minimum width=1.5cm] at (1.5,0) {B};
\end{tikzpicture}
```

There are various keys to customize this shape (to use PGF keys in TikZ, simply remove the `/pgf/` path).

`/pgf/aspect=(value)`

(no default, initially 1.0)

The aspect is a recommendation for the quotient of the radii of the cylinder end. This may be ignored if the shape is enlarged to some minimum width.

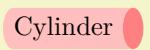


```
\usetikzlibrary {shapes.geometric}
\begin{tikzpicture} []
\tikzset{every node/.style={cylinder, shape border rotate=90, draw}}
\node [aspect=1.0] {A};
\node [aspect=0.5] at (1,0) {B};
\node [aspect=0.25] at (2,0) {C};
\end{tikzpicture}
```

`/pgf/cylinder uses custom fill=(boolean)`

(default `true`)

This enables the use of a custom fill for the body and the end of the cylinder. The background path for the shape should not be filled (e.g., in TikZ, the `fill` option for the node must be implicitly or explicitly set to `none`). Internally, this key sets the TeX-if `\ifpgfcylinderusescustomfill` appropriately.



```
\usetikzlibrary {shapes.geometric}
\begin{tikzpicture} [aspect=0.5]
\node [cylinder, cylinder uses custom fill, cylinder end fill=red!50,
cylinder body fill=red!25] {Cylinder};
\end{tikzpicture}
```

`/pgf/cylinder end fill=(color)`

(no default, initially `white`)

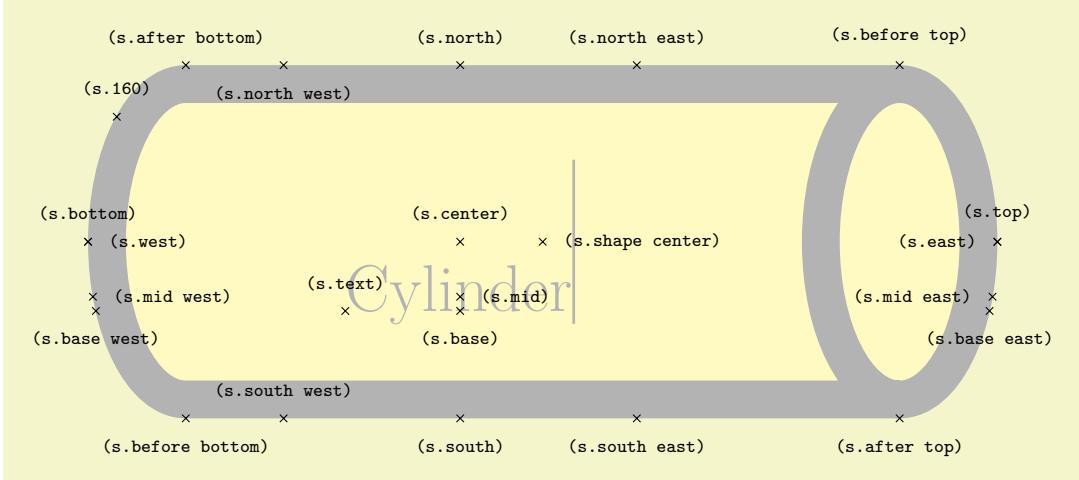
Sets the color for the end of the cylinder.

`/pgf/cylinder body fill=(color)`

(no default, initially `white`)

Sets the color for the body of the cylinder.

The anchors of this shape are shown below (anchor 160 is an example of a border anchor). Note that the cylinder shape does not distinguish between `outer xsep` and `outer ysep`. Only the larger of the two values is used for the shape. Note also the difference between the `center` and `shape center` anchors: `center` is the center of the cylinder body and also the center of rotation. The `shape center` is the center of the shape which includes the 2-dimensional representation of the cylinder top.



```
\usetikzlibrary {shapes.geometric}
\Huge
\begin{tikzpicture}
\node[name=s, shape=cylinder, shape example, aspect=.5, inner xsep=3cm,
      inner ysep=1cm] {Cylinder};
\foreach \anchor/\placement in
  {before top/above, top/above, after top/below,
   before bottom/below, bottom/above, after bottom/above,
   mid/right, mid west/right, mid east/left,
   base/below, base west/below, base east/below,
   center/above, text/above, shape center/right,
   west/right, east/left, north/above, south/below,
   north west/below, north east/above,
   south west/above, south east/below, 160/above}
\draw[shift=(s.\anchor)] plot [mark=x] coordinates{(0,0)}
  node[\placement] {\scriptsize\textrm{\texttt{(s.\anchor)}}};
\end{tikzpicture}
```

72.4 Symbol Shapes

TikZ Library `shapes.symbols`

```
\usepgflibrary{shapes.symbols} % LATEX and plain TEX and pure pgf
\usepgflibrary[shapes.symbols] % ConTEXt and pure pgf
\usetikzlibrary{shapes.symbols} % LATEX and plain TEX when using TikZ
\usetikzlibrary[shapes.symbols] % ConTEXt when using TikZ
```

This library defines shapes that can be used for drawing symbols like a forbidden sign or a cloud.

Shape `correct forbidden sign`

This shape places the node inside a circle with a diagonal from the upper left to the lower right added. The circle is part of the background, the diagonal line part of the foreground path; thus, the diagonal line is on top of the text.



```
\usetikzlibrary {shapes.symbols}
\begin{tikzpicture}
\node [correct forbidden sign, line width=1ex, draw=red, fill=white] {Smoking};
\end{tikzpicture}
```

The shape inherits all anchors from the `circle` shape.

Shape `forbidden sign`

This shape is like `correct forbidden sign`, only the line goes from the lower left to the upper right. The strange naming of these shapes is for historical reasons.



```
\usetikzlibrary {shapes.symbols}
\begin{tikzpicture}
  \node [forbidden sign,line width=1ex,draw=red,fill=white] {Smoking};
\end{tikzpicture}
```

The shape inherits all anchors from the `circle` shape.

Shape `magnifying glass`

This shape places the node inside a circle with a handle attached to the node. The angle of the handle and its length can be adjusted using two keys:

`/pgf/magnifying glass handle angle fill=(degree)`

(default -45)

The angle of the handle.

`/pgf/magnifying glass handle angle aspect=(factor)`

(default 1.5)

The length of the handle as a multiple of the circle radius.



```
\usetikzlibrary {shapes.symbols}
\begin{tikzpicture}
  \node [magnifying glass,line width=1ex,draw] {huge};
\end{tikzpicture}
```

The shape inherits all anchors from the `circle` shape.

Shape `cloud`

This shape is a cloud, drawn to tightly fit the node contents (strictly speaking, using an ellipse which tightly fits the node contents – including any `inner sep`).



```
\usetikzlibrary {shapes.symbols}
\begin{tikzpicture}
  \node [cloud, draw, fill=gray!20, aspect=2] {ABC};
  \node [cloud, draw, fill=gray!20] at (1.5,0) {D};
\end{tikzpicture}
```

A cloud should be thought of as having a number of “puffs”, which are the individual arcs drawn around the border. There are PGF keys to specify how the cloud is drawn (to use these keys in TikZ, simply remove the `/pgf/` path).

`/pgf/cloud puffs=(integer)`

(no default, initially 10)

Sets the number of puffs for the cloud.

`/pgf/cloud puff arc=(angle)`

(no default, initially 135)

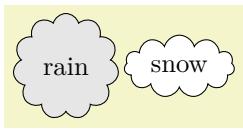
Sets the length of the puff arc (in degrees). A shorter arc can produce better looking joins between puffs for larger line widths.

Like the diamond shape, the cloud shape also uses the `aspect` key to determine the ratio of the width and the height of the cloud. However, there may be circumstances where it may be undesirable to continually specify the `aspect` for the cloud. Therefore, the following key is implemented:

`/pgf/cloud ignores aspect=(boolean)`

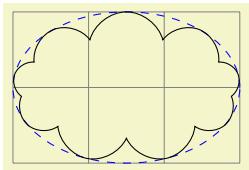
(default `true`)

Instruct PGF to ignore the `aspect` key. Internally, the T_EX-if `\ifpgfcloudignoresaspect` is set appropriately. The initial value is `false`.



```
\usetikzlibrary {shapes.symbols}
\begin{tikzpicture}[aspect=1, every node/.style={cloud, cloud puffs=11, draw}]
  \node [fill=gray!20]                                [rain];
  \node [cloud ignores aspect, fill=white] at (1.5,0) [snow];
\end{tikzpicture}
```

Any minimum size requirements are applied to the “circum-ellipse”, which is the ellipse which passes through all the midpoints of the puff arcs. These requirements are considered *after* any aspect specification is applied.

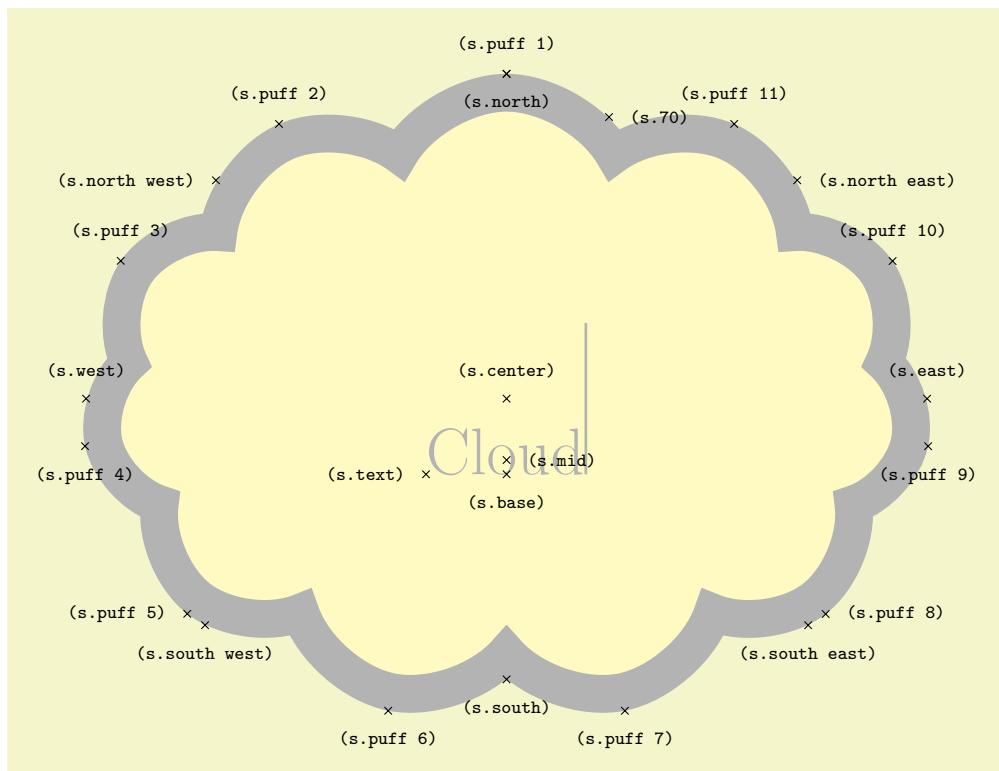


```

\usetikzlibrary {shapes.symbols}
\begin{tikzpicture}
  \draw [help lines] grid (3,2);
  \draw [blue, dashed] (1.5, 1) ellipse (1.5cm and 1cm);
  \node [cloud, cloud puffs=9, draw, minimum width=3cm, minimum height=2cm]
    at (1.5, 1) {};
\end{tikzpicture}

```

The anchors for the cloud shape are shown below for a cloud with eleven puffs. Anchor 70 is an example of a border anchor.



```

\usetikzlibrary {shapes.symbols}
\Huge
\begin{tikzpicture}
\node[name=s, shape=cloud, style=shape example, cloud puffs=11, aspect=1.5,
      cloud puff arc=120,inner ysep=1cm] {Cloud\vrule width 1pt height 2cm};
\foreach \anchor/\placement in
{puff 1/above, puff 2/above, puff 3/above, puff 4/below,
 puff 5/left, puff 6/below, puff 7/below, puff 8/right,
 puff 9/below, puff 10/above, puff 11/above, 70/right,
 center/above, base/below, mid/right, text/left,
 north/below, south/below, east/above, west/above,
 north west/left, north east/right,
 south west/below, south east/below}
\draw[shift=(s.\anchor)] plot[mark=x] coordinates{(0,0)}
    node[\placement] {\scriptsize\textrm{\tt{(s.\anchor)}}};
\end{tikzpicture}

```

Shape `starburst`

This shape is a randomly generated elliptical star, which supports the rotation of the shape border as described in Section 17.2.3.



```
\usetikzlibrary {shapes.symbols}
\begin{tikzpicture}
  \node[starburst, fill=yellow, draw=red, line width=2pt] {\bf BANG!};
\end{tikzpicture}
```

Like the `star` shape, the starburst should be thought of as having a set of inner points and outer points. The inner points lie on the ellipse which tightly fits the node contents (including any `inner sep`).

Using a specified ‘starburst point height’ value, the outer points are generated randomly between this value and one quarter of this value. For a given starburst shape, the angle between each point is fixed, and is determined by the number of points specified for the starburst.

It is important to note that, whilst the maximum possible point height is used to calculate minimum width or height requirements, the outer points are randomly generated, so there is (unfortunately) no guarantee that any such requirements will be fully met.



```
\usetikzlibrary {shapes.symbols}
\begin{tikzpicture}
  \draw[help lines] grid(3,2);
  \node[starburst, draw, minimum width=3cm, minimum height=2cm]
    at (1.5, 1) {\bf BOOM!};
\end{tikzpicture}
```

There are PGF keys to control the drawing of the starburst shape. To use these keys in TikZ, simply remove the `/pgf/` path.

`/pgf/starburst points=<integer>` (no default, initially 17)

Sets the number of outer points for the starburst.

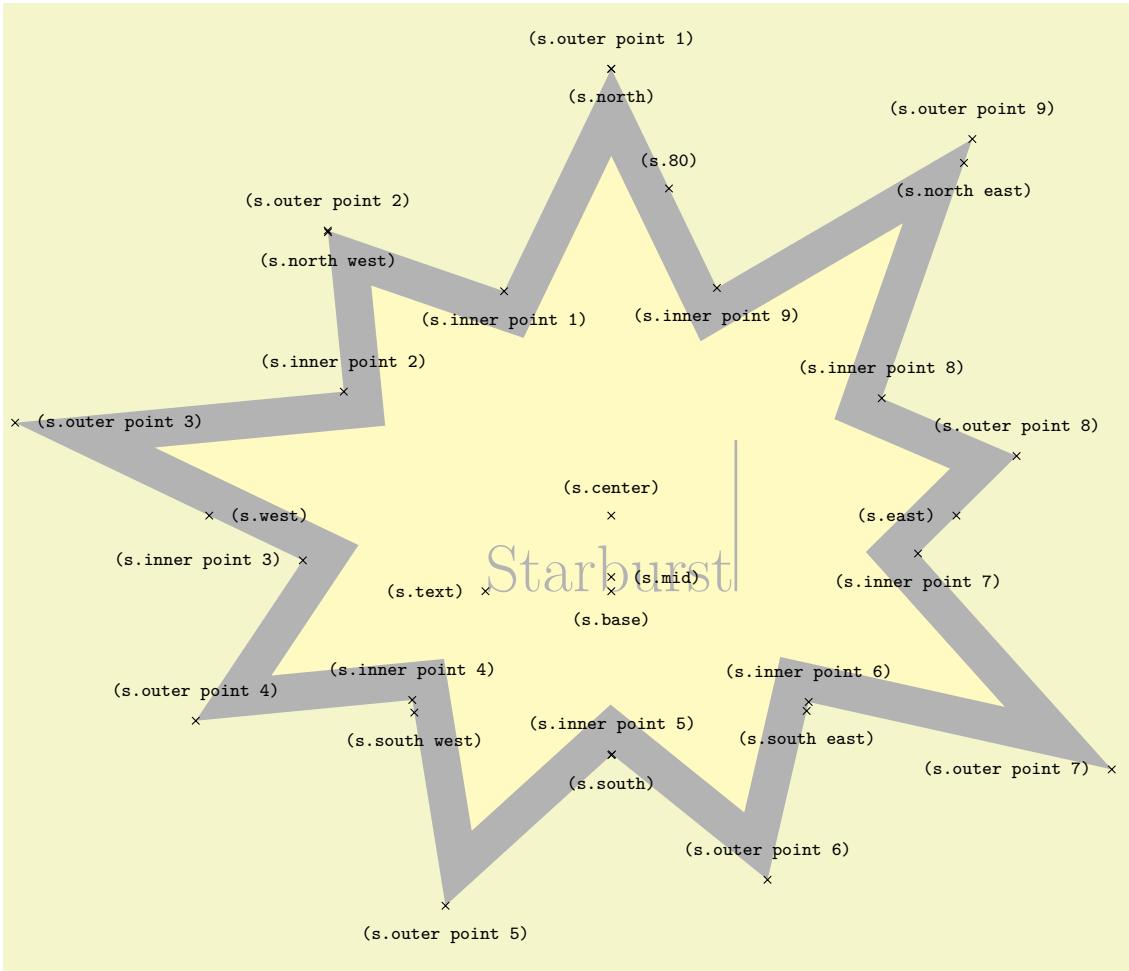
`/pgf/starburst point height=<length>` (no default, initially .5cm)

Sets the *maximum* distance between the inner point radius and the outer point radius.

`/pgf/random starburst=<integer>` (no default, initially 100)

Sets the seed for the random number generator for creating the starburst. The maximum value for `<integer>` is 16383. If `<integer>=0`, the random number generator will not be used, and the maximum point height will be used for all outer points. If `<integer>` is omitted, a seed will be randomly chosen.

The basic anchors for a nine point `starburst` shape are shown below. Anchor 80 is an example of a border anchor.



```
\usetikzlibrary {shapes.symbols}
\Huge
\begin{tikzpicture}
\node[name=s, shape=starburst, starburst points=9, starburst point height=3.5cm,
      style=shape example,inner sep=1cm]
{Starburst\textcolor{violet}{r}ule width 1pt height 2cm};
\foreach \anchor/\placement in
{outer point 1/above, outer point 2/above, outer point 3/right,
 outer point 4/above, outer point 5/below, outer point 6/above,
 outer point 7/left, outer point 8/above, outer point 9/above,
 inner point 1/below, inner point 2/above, inner point 3/left,
 inner point 4/above, inner point 5/above, inner point 6/above,
 inner point 7/below, inner point 8/above, inner point 9/below,
 center/above, text/left, mid/right, base/below, 80/above,
 north/below, south/below, east/left, west/right,
 north east/below, south west/below, south east/below, north west/below}
\draw[shift=(s.\anchor)] plot[mark=x] coordinates{(0,0)}
node[\placement] {\textcolor{blue}{\scriptsize\textrm{\texttt{(s.\anchor)}}}};
\end{tikzpicture}
```

Shape `signal`

This shape is a “signal” or sign shape, that is, a rectangle, with optionally pointed sides. A signal can point “to” somewhere, with outward points in that direction. It can also be “from” somewhere, with inward points from that direction. The resulting points extend the node contents (which include the `inner sep`).

To East

From East

```
\usetikzlibrary {shapes.symbols}
\begin{tikzpicture}
[every node/.style={signal, draw, text=white, signal to=north}]
\node[fill=green!65!black, signal to=east] at (0,1) {To East};
\node[fill=red!65!black, signal from=east] at (0,0) {From East};
\end{tikzpicture}
```

There are PGF keys for drawing the signal shape (to use these keys in TikZ, simply remove the `/pgf/` path):

`/pgf/signal pointer angle=<angle>`

(no default, initially 90)

Sets the angle for the pointed sides of the shape. This angle is maintained when enforcing any minimum size requirements, so any adjustment to the width will affect the height, and vice versa.

`/pgf/signal from=<direction> and <opposite direction>`

(no default, initially `nowhere`)

Sets which sides take an inward pointer (i.e., that points towards the center of the shape). The possible values for `<direction>` and `<opposite direction>` are the compass point directions `north`, `south`, `east` and `west` (or `above`, `below`, `right` and `left`). An additional keyword `nowhere` can be used to reset the sides so they have no pointers. When used with `signal from` key, this only resets inward pointers; used with the `signal to` key, it only resets outward pointers.

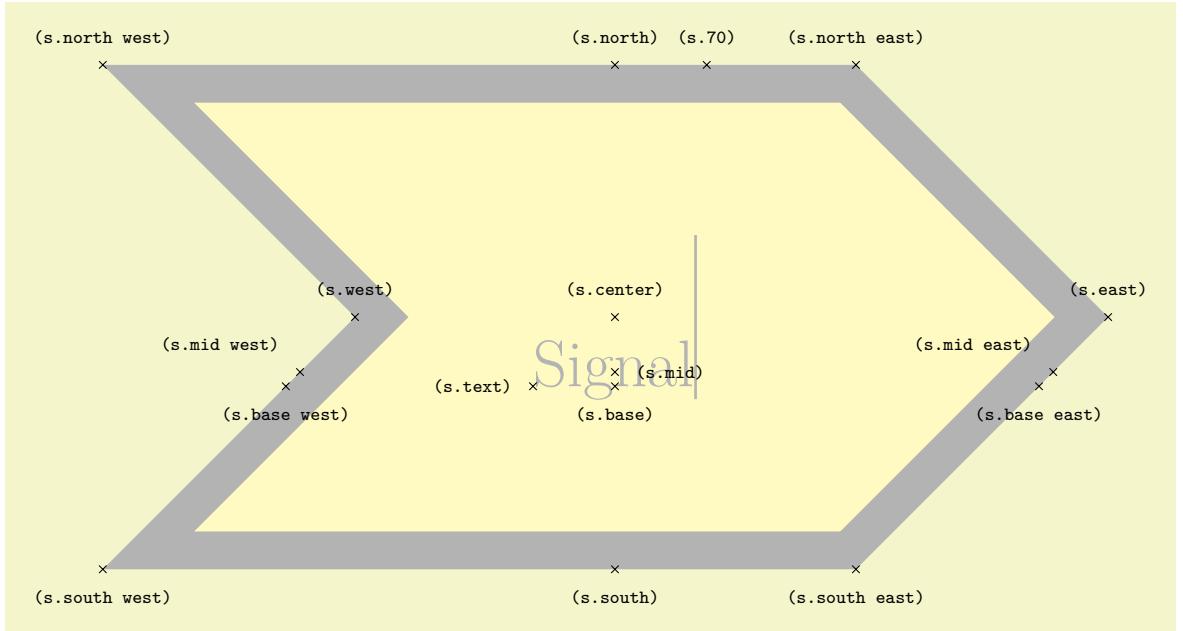
`/pgf/signal to=<direction> and <opposite direction>`

(no default, initially `east`)

Sets which sides take an outward pointer (i.e., that points away from the shape).

Note that PGF will ignore any instruction to use directions that are not opposites (so using the value `east` and `north`, will result in only `north` being assigned a pointer). This is also the case if non-opposite values are used in the `signal to` and `signal from` keys at the same time. So, for example, it is not possible for a signal to have an outward point to the left, and also have an inward point from below.

The anchors for the signal shape are shown below. Anchor 70 is an example of a border anchor.



```
\usetikzlibrary {shapes.symbols}
\Huge
\begin{tikzpicture}
\node[name=s, shape=signal, signal from=west, shape example, inner sep=2cm]
{Signal \vrule width1pt height2cm};
\foreach \anchor/\placement in
{text/left, center/above, 70/above,
base/below, base east/below, base west/below,
mid/right, mid east/above left, mid west/above left,
north/above, south/below,
east/above, west/above,
north west/above, north east/above,
south west/below, south east/below}
\draw[shift=(s.\anchor)] plot[mark=x] coordinates{(0,0)}
node[\placement] {\scriptsize\tt(s.\anchor)};
\end{tikzpicture}
```

Shape tape

This shape is a rectangle with optional, “bendy” top and bottom sides, which tightly fits the node contents (including the `inner sep`).

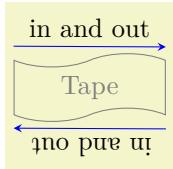


```
\usetikzlibrary {shapes.symbols}
\begin{tikzpicture}
\node[tape, draw] {ABCD};
\node[tape, draw, tape bend top=none] at (1.5, 0) {EFGH};
\end{tikzpicture}
```

There are PGF keys to specify which sides bend and how high the bends are (to use these keys in TikZ, simply remove the `/pgf/` path):

`/pgf/tape bend top=(bend style)` (no default, initially `in` and `out`)

Specifies how the top side bends. The `(bend style)` is either `in` and `out`, `out` and `in` or `none` (i.e., a straight line). The bending sides are drawn in a clockwise direction, and using the bend style `in` and `out` will mean the side will first bend inwards and then bend outwards. The opposite holds true for `out` and `in`.



```
\usetikzlibrary {shapes.symbols}
\begin{tikzpicture}[-stealth]
\node[tape, draw, gray, minimum width=2cm] (t){Tape};
\draw [blue] ([yshift=5pt] t.north west) -- ([yshift=5pt] t.north east)
node [midway, above, black] {in and out};
\draw [blue] ([yshift=-5pt] t.south east) -- ([yshift=-5pt] t.south west)
node [sloped, allow upside down, midway, above, black] {in and out};
\end{tikzpicture}
```

This might take a bit of getting used to, but just remember that when you want the bendy sides to be parallel, the sides take the same bend style. It is possible for the top and bottom sides to take opposite bend styles, but the author of this shape cannot think of a single use for such a combination.



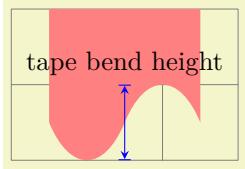
```
\usetikzlibrary {shapes.symbols}
\begin{tikzpicture}[every node/.style={tape, draw}]
\node [tape bend top=out and in, tape bend bottom=out and in] {Parallel};
\node at (2,0) [tape bend bottom=out and in] {Why?};
\end{tikzpicture}
```

`/pgf/tape bend bottom=(bend style)` (no default, initially `in` and `out`)

Specifies how the bottom side bends.

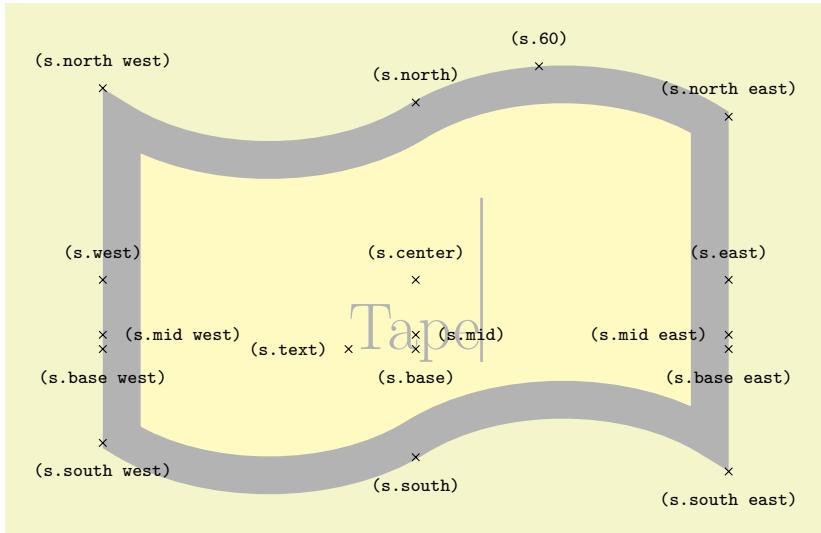
`/pgf/tape bend height=(length)` (no default, initially 5pt)

Sets the total height for a side with a bend.



```
\usetikzlibrary {shapes.symbols}
\begin{tikzpicture}[>=stealth]
\draw [help lines] grid(3,2);
\node [tape, fill, minimum size=2cm, red!50, tape bend top=none,
tape bend height=1cm] at (1.5,1.5) (t) {};
\draw [|->|, blue] (1.5,0) -- (1.5,1)
node [at end, above, black] {tape bend height};
\end{tikzpicture}
```

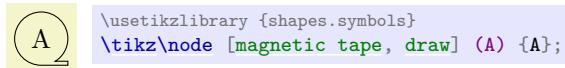
The anchors for the tape shape are shown below. Anchor 60 is an example of a border anchor. Note that border anchors will snap to the center of convex curves (i.e. when bending in).



```
\usetikzlibrary {shapes.symbols}
\Huge
\begin{tikzpicture}
\node [name=s, shape=tape, tape bend height=1cm, shape example, inner xsep=3cm]
{Tape\vrule width1pt height2cm};
\foreach \anchor/\placement in
{text/left, center/above, 60/above,
base/below, base east/below, base west/below,
mid/right, mid east/left, mid west/right,
north/above, south/below, east/above, west/above,
north west/above, north east/above,
south west/below, south east/below}
\draw[shift=(s.\anchor)] plot[mark=x] coordinates{(0,0)}
node[\placement] {\scriptsize\tt(s.\anchor)};
\end{tikzpicture}
```

Shape `magnetic tape`

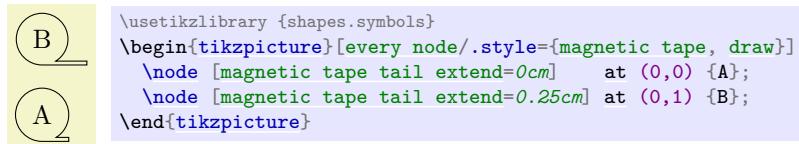
This shape represents a ‘magnetic tape’ or any sequential data store that is sometimes used in flowcharts. It is essentially a circle with a little tail:



The following keys can be used to customise the `magnetic tape` shape:

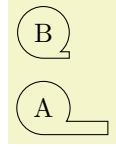
`/pgf/magnetic tape tail extend=(distance)` (no default, initially 0cm)

This key sets how far the tail extends beyond the radius of the tape. Negative values will be ignored.



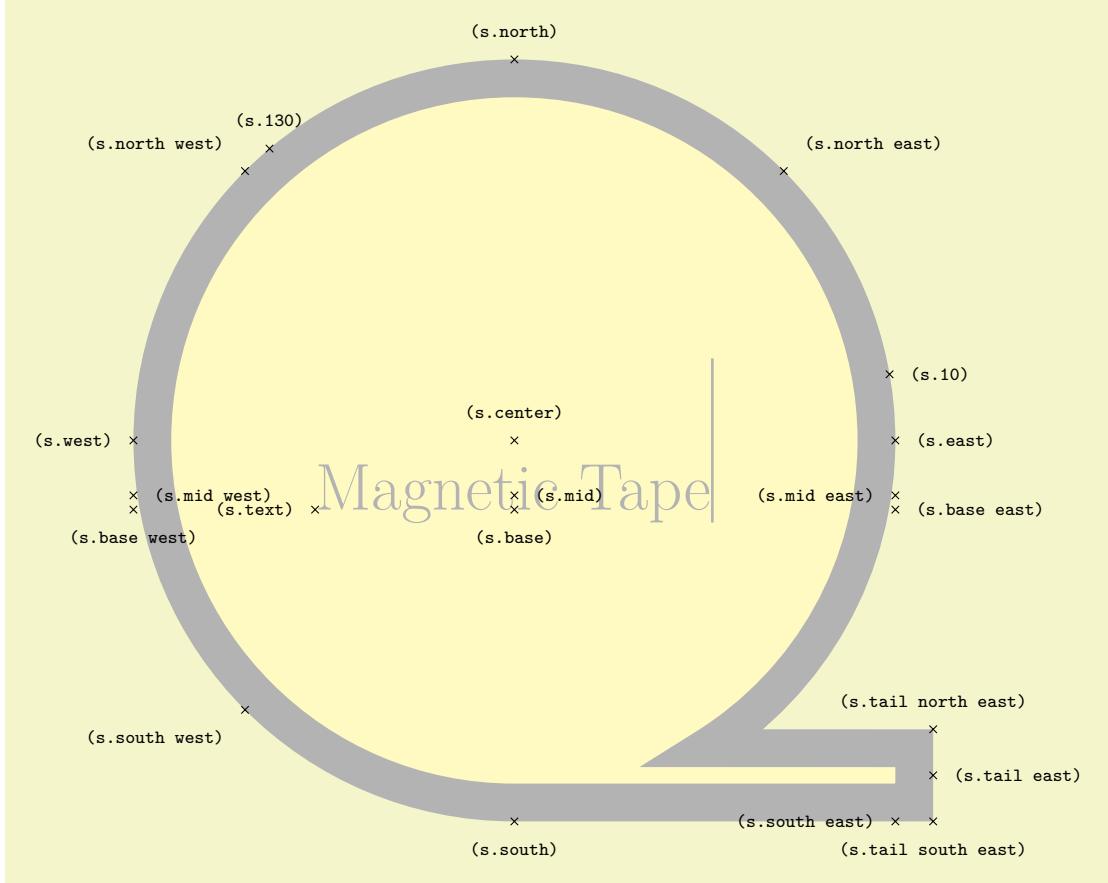
`/pgf/magnetic tape tail=(proportion)` (no default, initially 0.15)

This key sets the thickness of the ‘tail’ to be $\langle proportion \rangle$ times the radius of the shape. The $\langle proportion \rangle$ should be between 0.0 and 1.0.



```
\usetikzlibrary {shapes.symbols}
\begin{tikzpicture}[every node/.style={magnetic tape, draw}]
  \node [magnetic tape tail=0.5, magnetic tape tail extend=0.5cm] {A};
  \node [magnetic tape tail=0.25] at (0,1) {B};
\end{tikzpicture}
```

The following figure shows the anchors this shape defines; the anchors 10 and 130 are example of border anchors.



```
\usetikzlibrary {shapes.symbols}
\Huge
\begin{tikzpicture}
\node[name=s,shape=magnetic tape,shape example,inner sep=0.75cm,
magnetic tape tail extend=0.5cm]
{Magnetic Tape\vrule width 1pt height 2cm};
\foreach \anchor/\placement in
{north west/above left, north/above, north east/above right,
west/left, center/above, east/right,
mid west/right, mid/right, mid east/left,
base west/below, base/below, base east/right,
south west/below left, south/below, south east/left,
text/left, 10/right, 130/above,
tail east/right, tail south east/below, tail north east/above}
\draw[shift=(s.\anchor)] plot[mark=x] coordinates{(0,0)}
node[\placement]{\scriptsize\sffamily\texttt{(s.\anchor)}};
\end{tikzpicture}
```

72.5 Arrow Shapes

TikZ Library `shapes.arrows`

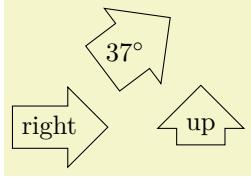
```
\usepgflibrary{shapes.arrows} % LATEX and plain TEX and pure pgf
\usepgflibrary[shapes.arrows] % ConTeXt and pure pgf
```

```
\usetikzlibrary{shapes.arrows} % LATEX and plain TEX when using TikZ
\usetikzlibrary[shapes.arrows] % ConTEXt when using TikZ
```

This library defines arrow shapes. Note that an arrow shape is something quite different from a (normal) arrow tip: It is a shape that just “happens” to “look like” an arrow. In particular, you cannot use these shapes as arrow tips.

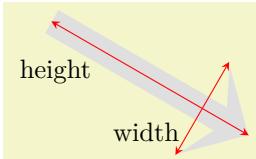
Shape `single arrow`

This shape is an arrow, which tightly fits the node contents (including any `inner sep`). This shape supports the rotation of the shape border, as described in Section 17.2.3. The angle of rotation determines in which direction the arrow points (provided no other rotational transformations are applied).



```
\usetikzlibrary {shapes.arrows}
\begin{tikzpicture}[every node/.style={single arrow, draw},
    rotate border/.style={shape border uses incircle, shape border rotate=#1}]
\node [right];
\node at (2,0) [shape border rotate=90]{up};
\node at (1,1) [rotate border=37, inner sep=0pt]{$37^\circ$};
\end{tikzpicture}
```

Regardless of the rotation of the arrow border, the width is measured between the back ends of the arrow head, and the height is measured from the arrow tip to the end of the arrow tail.



```
\usetikzlibrary {shapes.arrows}
\begin{tikzpicture}[>=stealth,
    rotate border/.style={shape border uses incircle, shape border rotate=#1}]
\node[rotate border=-30, fill=gray!25, minimum height=3cm, single arrow,
    single arrow head extend=.5cm, single arrow head indent=.25cm] (arrow) {};
\draw[red, <->] (arrow.before tip) -- (arrow.after tip)
node [near end, left, black] {width};
\draw[red, <->] (arrow.tip) -- (arrow.tail)
node [near end, below left, black] {height};
\end{tikzpicture}
```

There are PGF keys that can be used to customize this shape (to use these keys in TikZ, simply remove the `/pgf/` path).

`/pgf/single arrow tip angle=<angle>`

(no default, initially 90)

Sets the angle for the arrow tip. Enlarging the arrow to some minimum width may increase the height of the shape to maintain this angle.

`/pgf/single arrow head extend=<length>`

(no default, initially .5cm)

This sets the distance between the tail of the arrow and the outer end of the arrow head. This may change if the shape is enlarged to some minimum width.

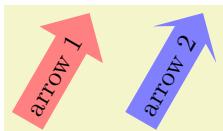


```
\usetikzlibrary {shapes.arrows}
\begin{tikzpicture}
\node[single arrow, draw, single arrow head extend=.5cm, gray!50, rotate=60]
(a) {Arrow};
\draw[red, |->|] (a.before tip) -- (a.before head)
node [midway, below, sloped, black] {head extend};
\end{tikzpicture}
```

`/pgf/single arrow head indent=<length>`

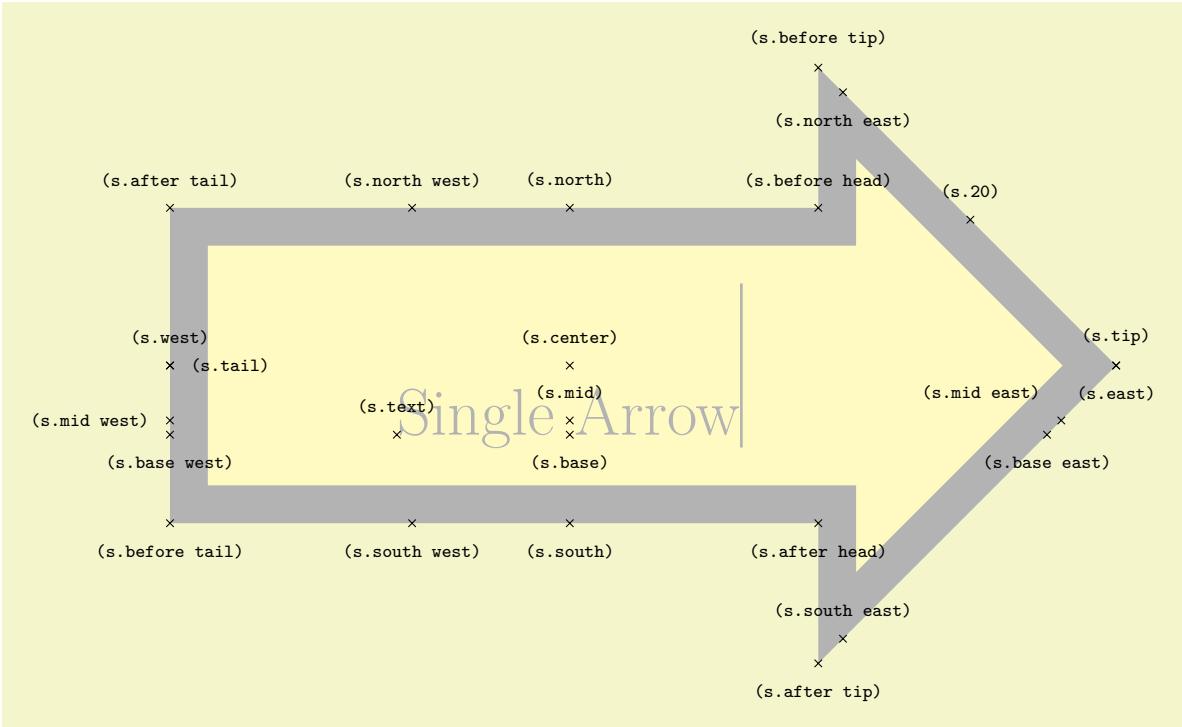
(no default, initially 0cm)

This moves the point where the arrow head joins the shaft of the arrow *towards* the arrow tip, by `<length>`.



```
\usetikzlibrary {shapes.arrows}
\begin{tikzpicture}[every node/.style={single arrow, draw=none, rotate=60}]
\node [fill=red!50] {arrow 1};
\node [fill=blue!50, single arrow head indent=1ex] at (1.5,0) {arrow 2};
\end{tikzpicture}
```

The anchors for this shape are shown below (anchor 20 is an example of a border anchor).



```
\usetikzlibrary {shapes.arrows}
\Huge
\begin{tikzpicture}
\node [name=s,shape=single arrow, shape example, single arrow head extend=1.5cm]
{Single Arrow\rlap{\rule{1pt}{height2cm}}};
\foreach \anchor/\placement in
{text/above, center/above, 20/above,
mid west/left, mid/above, mid east/above left,
base west/below, base/below, base east/below,
tip/above, before tip/above, after tip/below, before head/above,
after head/below, after tail/above, before tail/below, tail/right,
north/above, south/below, east/below, west/above,
north west/above, north east/below, south west/below, south east/above}
\draw[shift=(s.\anchor)] plot[mark=x] coordinates{(0,0)}
node[\placement] {\scriptsize\textrm{\texttt{(s.\anchor)}}};
\end{tikzpicture}
```

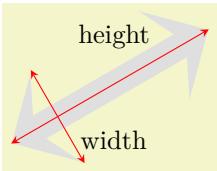
Shape double arrow

This shape is a double arrow, which tightly fits the node contents (including any `inner sep`), and supports the rotation of the shape border, as described in Section 17.2.3.



```
\usetikzlibrary {shapes.arrows}
\begin{tikzpicture}[every node/.style={double arrow, draw}]
\node [double arrow, draw] {Left or Right};
\end{tikzpicture}
```

The double arrow behaves exactly like the single arrow, so you need to remember that the width is *always* the distance between the back ends of the arrow heads, and the height is *always* the tip-to-tip distance.



```

\usetikzlibrary {shapes.arrows}
\begin{tikzpicture} [->=stealth,
    rotate border/.style={shape border uses incircle, shape border rotate=#1}]
\node [rotate border=210, fill=gray!25, minimum height=3cm, double arrow,
    double arrow head extend=.5cm, double arrow head indent=.25cm] (arrow) {};
\draw[red, <->] (arrow.before tip 1) -- (arrow.after tip 1)
    node [near start, right, black] {width};
\draw[red, <->] (arrow.tip 1) -- (arrow.tip 2)
    node [near end, above left, black] {height};
\end{tikzpicture}

```

The PGF keys that can be used to customize the double arrow behave similarly to the keys for the single arrow (to use these keys in TikZ, simply remove the `/pgf/` path).

`/pgf/double arrow tip angle=<angle>`

(no default, initially 90)

Sets the angle for the arrow tip. Enlarging the arrow to some minimum width may increase the height of the shape to maintain this angle.

/pgf/double arrow head extend=<length>

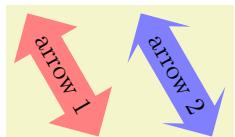
(no default, initially .5cm)

This sets the distance between the shaft of the arrow and the outer end of the arrow heads. This may change if the shape is enlarged to some minimum width.

/pgf/double arrow head indent= $\langle length \rangle$

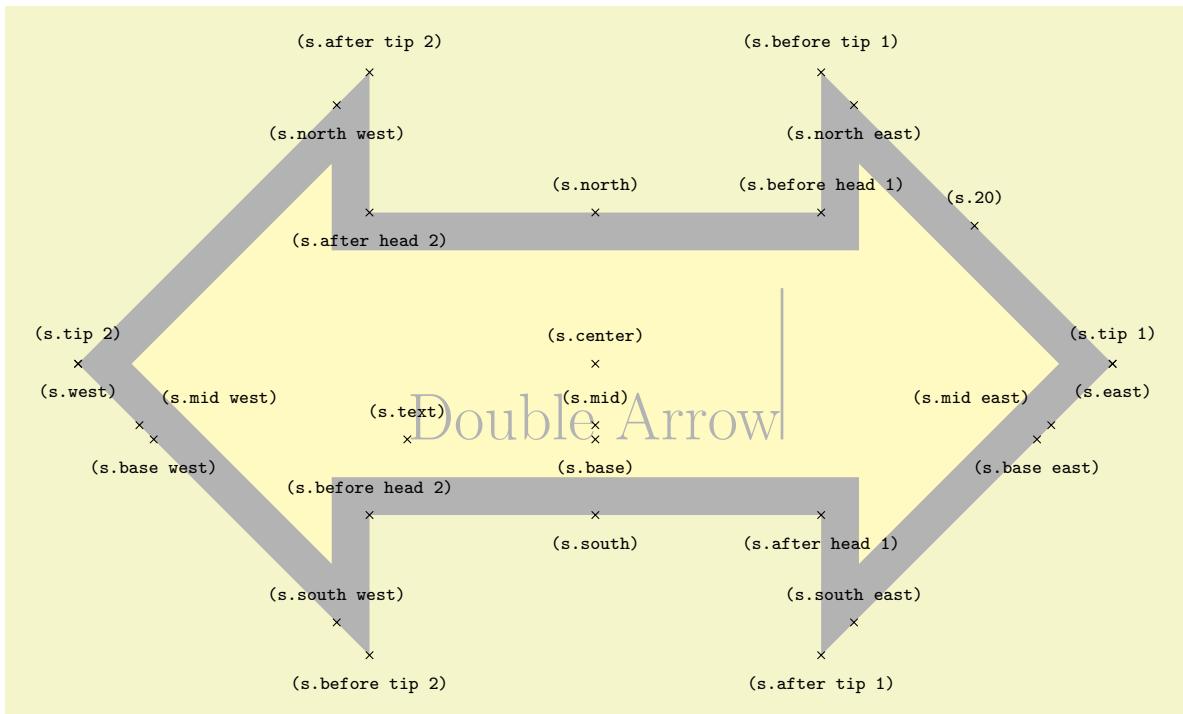
(no default, initially 0cm)

This moves the point where the arrow heads join the shaft of the arrow *towards* the arrow tips, by $\langle length \rangle$.



```
\usetikzlibrary {shapes.arrows}
\begin{tikzpicture}[every node/.style={double arrow, draw=none, rotate=-60}]
  \node [fill=red!50]                                {arrow 1};
  \node [fill=blue!50, double arrow head indent=1ex] at (1.5,0) {arrow 2};
\end{tikzpicture}
```

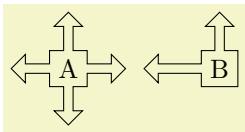
The anchors for this shape are shown below (anchor 20 is an example of a border anchor).



```
\usetikzlibrary {shapes.arrows}
\Huge
\begin{tikzpicture}
\node [name=s,shape=double arrow, double arrow head extend=1.5cm, shape example, inner xsep=2cm]
  {Double Arrow\vrule width1pt height2cm};
\foreach \anchor/\placement in
  {text/above, center/above, 20/above,
   mid west/above right, mid/above, mid east/above left,
   base west/below, base/below, base east/below,
   before head 1/above, before tip 1/above, tip 1/above, after tip 1/below, after head 1/below,
   before head 2/above, before tip 2/below, tip 2/above, after tip 2/above, after head 2/below,
   north/above, south/below, east/below, west/below,
   north west/below, north east/below, south west/above, south east/above}
  \draw[shift=(s.\anchor)] plot[mark=x] coordinates{(0,0)};
  node[\placement] {\scriptsize\textrm{\tt\{s.\anchor\}}};
\end{tikzpicture}
```

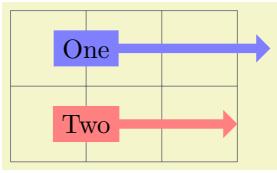
Shape arrow box

This shape is a rectangle with optional arrows which extend from the four sides.



```
\usetikzlibrary {shapes.arrows}
\begin{tikzpicture}
\node[arrow box, draw] {A};
\node[arrow box, draw, arrow box arrows={north:.5cm, west:0.75cm}]
  at (2,0) {B};
\end{tikzpicture}
```

Any minimum size requirements are applied to the main rectangle *only*. This does not pose too many problems if you wish to accommodate the length of the arrows, as it is possible to specify the length of each arrow independently, from either the border of the rectangle (the default) or the center of the rectangle.



```
\usetikzlibrary {shapes.arrows}
\begin{tikzpicture}
\tikzset{box/.style={arrow box, fill=#1}}
\draw [help lines] grid(3,2);
\node[box=blue!50, arrow box arrows={east:2cm}] at (1,1.5){One};
\node[box=red!50, arrow box arrows={east:2cm from center}] at (1,0.5){Two};
\end{tikzpicture}
```

There are various PGF keys for drawing this shape (to use these keys in TikZ, simply remove the `/pgf/` path).

`/pgf/arrow box tip angle=<angle>` (no default, initially 90)

Sets the angle at the arrow tip for all four arrows.

`/pgf/arrow box head extend=<length>` (no default, initially .125cm)

Sets the distance the arrow head extends away from the shaft of the arrow. This applies to all arrows.

`/pgf/arrow box head indent=<length>` (no default, initially 0cm)

Moves the point where the arrow head joins the shaft of the arrow *towards* the arrow tip. This applies to all arrows.

`/pgf/arrow box shaft width=<length>` (no default, initially .125cm)

Sets the width of the shaft of all arrows.

`/pgf/arrow box north arrow=<distance>` (no default, initially .5cm)

Sets the distance the north arrow extends from the node. By default this is from the border of the shape, but by using the additional keyword `from center`, the distance will be measured from the center of the shape. If `<distance>` is `0pt` or a negative distance, the arrow will not be drawn.

`/pgf/arrow box south arrow=<distance>` (no default, initially .5cm)

Sets the distance the south arrow extends from the node.

`/pgf/arrow box east arrow=<distance>` (no default, initially .5cm)

Sets the distance the east arrow extends from the node.

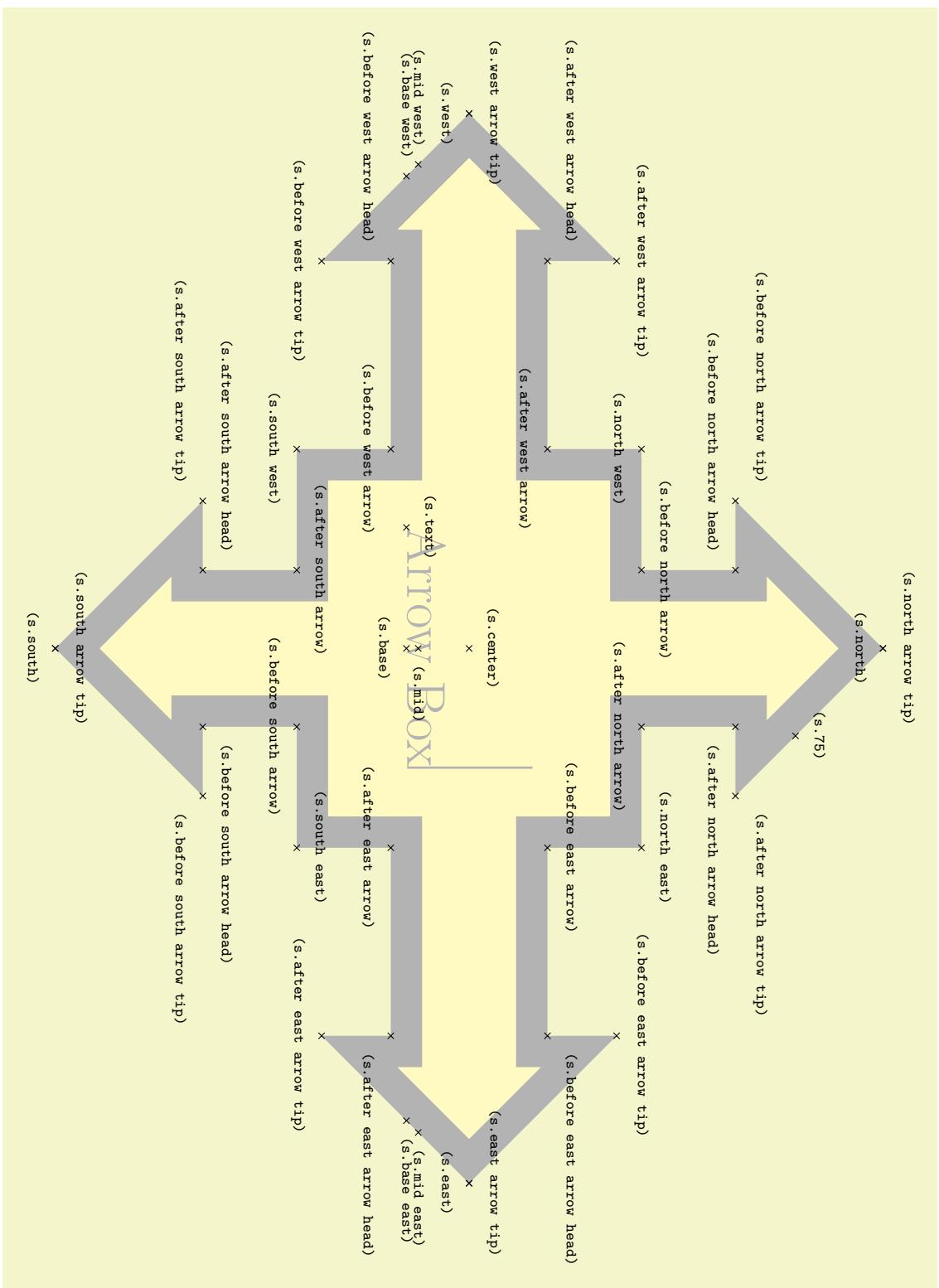
`/pgf/arrow box west arrow=<distance>` (no default, initially .5cm)

Sets the distance the west arrow extends from the node.

`/pgf/arrow box arrows={<list>}` (no default)

Sets the distance that all arrows extend from the node. The specification in `<list>` consists of the four compass points `north`, `south`, `east` or `west`, separated by commas (so the list must be contained within braces). The distances can be specified after each side separated by a colon (e.g., `north:1cm`, or `west:5cm from center`). If an item specifies no distance, the most recently specified distance will be used (at the start of the list this is `0cm`, so the first item in the list should specify a distance). Any sides not specified will not be drawn with an arrow.

The anchors for this shape are shown below (unfortunately, due to its size, this example must be rotated). Anchor 75 is an example of a border anchor. If a side is drawn without an arrow, the anchors for that arrow should be considered unavailable. They are (unavoidably) defined, but default to the center of the appropriate side.



```
\usetikzlibrary {shapes.arrows}
\Huge
\begin{tikzpicture}
\node[shape=arrow box, shape example, inner xsep=1cm, inner ysep=1.5cm, arrow box shaft width=2cm,
    arrow box arrows={north:3.5cm from border, south, east:5cm from border, west},
    arrow box head extend=0.75cm, rotate=-90](s) {Arrow Box\vrule width1pt height2cm};
\foreach \anchor/\placement in
{center/above, text/above, mid/right, base/below, 75/above,
 mid east/right, mid west/left, base east/right, base west/left,
 north/below, south/below, east/below, west/below,
 north east/above, south east/above, south west/below, north west/below,
 north arrow tip/above,south arrow tip/above, east arrow tip/above, west arrow tip/above,
 before north arrow/above, before north arrow head/below left, before north arrow tip/above left,
 after north arrow tip/above right, after north arrow head/below right, after north arrow/below,
 before south arrow/below, before south arrow head/above right, before south arrow tip/below right,
 after south arrow tip/below left, after south arrow head/above left, after south arrow/above,
 before east arrow/above, before east arrow head/above right, before east arrow tip/above,
 after east arrow tip/below, after east arrow head/below right, after east arrow/below,
 before west arrow/below, before west arrow head/below left, before west arrow tip/below,
 after west arrow tip/above, after west arrow head/above left, after west arrow/below}
\draw[shift=(s.\anchor)] plot[mark=x] coordinates{(0,0)}
    node[\placement, rotate=-90] {\scriptsize\textrm{\texttt{(s.\anchor)}}};
\end{tikzpicture}
```

72.6 Shapes with Multiple Text Parts

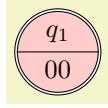
TikZ Library `shapes.multipart`

```
\usepgflibrary{shapes.multipart} % LATEX and plain TEX and pure pgf
\usepgflibrary[shapes.multipart] % ConTeXt and pure pgf
\usetikzlibrary{shapes.multipart} % LATEX and plain TEX when using TikZ
\usetikzlibrary[shapes.multipart] % ConTeXt when using TikZ
```

This library defines general-purpose shapes that are composed of multiple (text) parts.

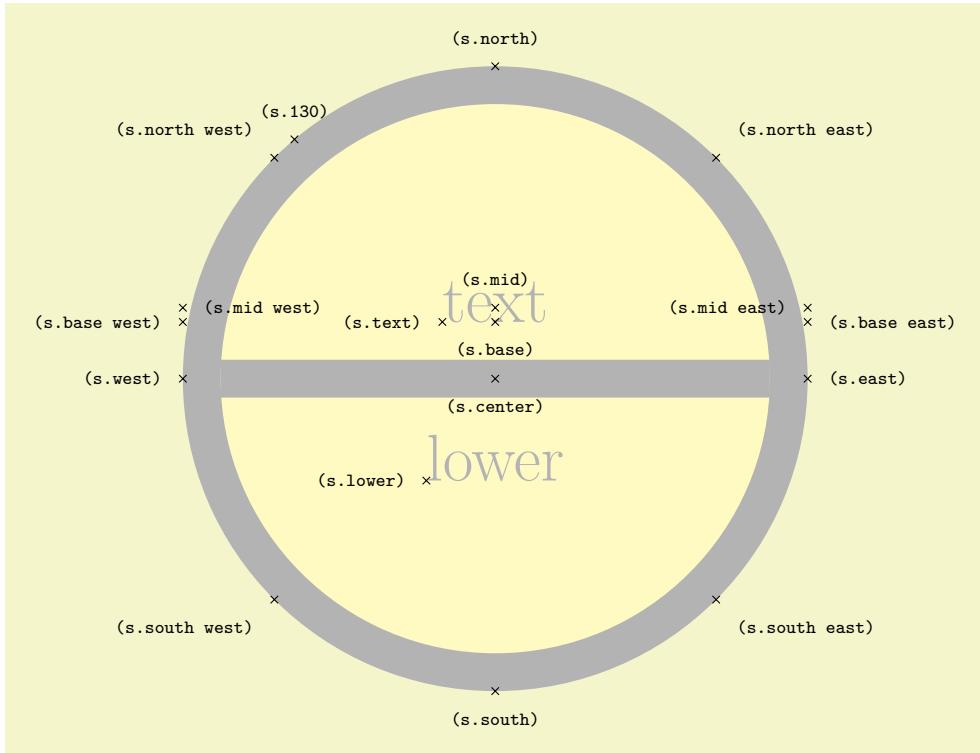
Shape `circle split`

This shape is a multi-part shape consisting of a circle with a line in the middle. The upper part is the main part (the `text` part), the lower part is the `lower` part.



```
\usetikzlibrary {shapes.multipart}
\begin{tikzpicture}
\node [circle split,draw,double,fill=red!20]
{
    $q_1$ 
    \nodepart{lower}
    $00$ 
};
\end{tikzpicture}
```

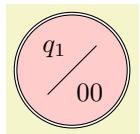
The shape inherits all anchors from the `circle` shape and defines the `lower` anchor in addition. See also the following figure:



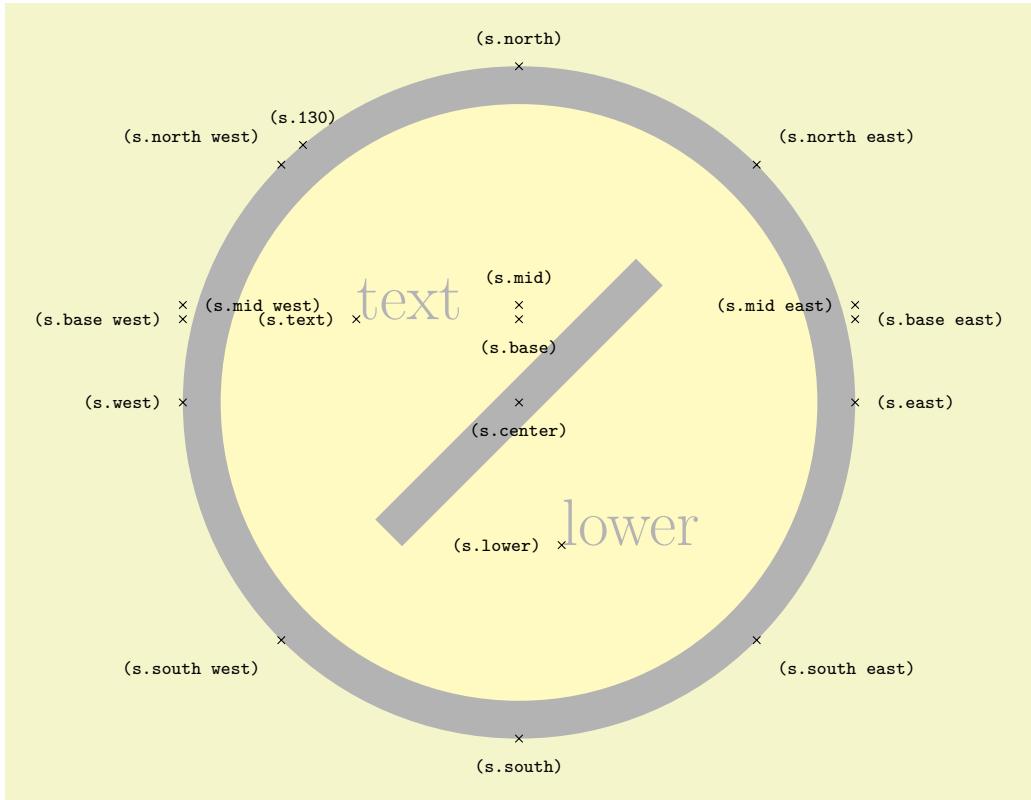
```
\usetikzlibrary {shapes.multipart}
\Huge
\begin{tikzpicture}
\node[name=s,shape=circle split,shape example] {text\nodepart{lower}lower};
\foreach \anchor/\placement in
{north west/above left, north/above, north east/above right,
west/left, center/below, east/right,
mid west/right, mid/above, mid east/left,
base west/left, base/below, base east/right,
south west/below left, south/below, south east/below right,
text/left, lower/left, 130/above}
\draw[shift=(s.\anchor)] plot[mark=x] coordinates{(0,0)};
\node[\placement] {\scriptsize\textrm{\texttt{(s.\anchor)}}};
\end{tikzpicture}
```

Shape `circle solidus`

This shape (due to Manuel Lacruz) is similar to the split circle, but the two text parts are arranged diagonally.



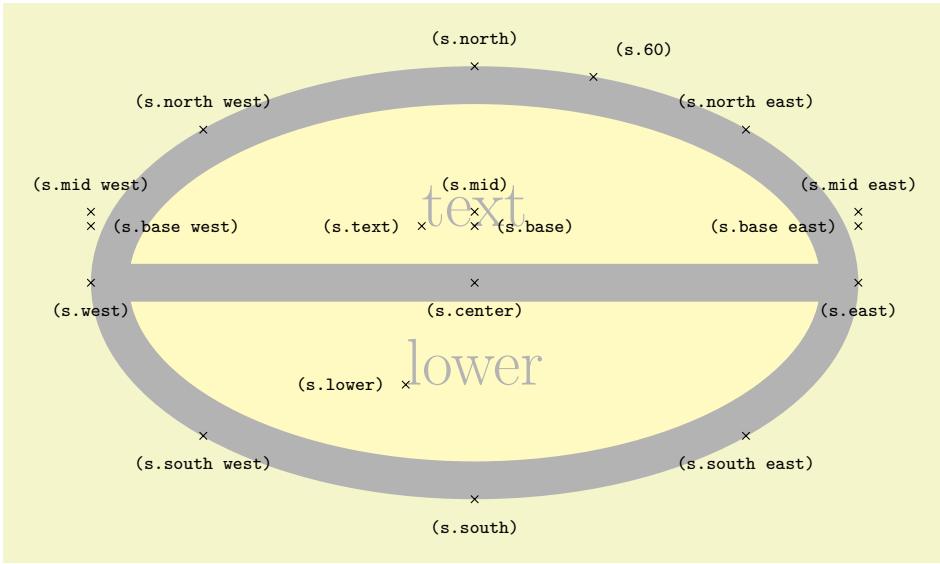
```
\usetikzlibrary {shapes.multipart}
\begin{tikzpicture}
\node [circle solidus,draw,double,fill=red!20]
{
$q_1$ 
\nodepart{lower}
$00$ 
};
\end{tikzpicture}
```



```
\usetikzlibrary {shapes.multipart}
\Huge
\begin{tikzpicture}
\node [name=s,shape=circle solidus,shape example,inner xsep=1cm] {text\nodepart{lower}lower};
\foreach \anchor/\placement in
{north west/above left, north/above, north east/above right,
west/left, center/below, east/right,
mid west/right, mid/above, mid east/left,
base west/left, base/below, base east/right,
south west/below left, south/below, south east/below right,
text/left, lower/left, 130/above}
\draw[shift=(s.\anchor)] plot[mark=x] coordinates{(0,0)}
node[\placement]{\scriptsize\sffamily\texttt{(s.\anchor)}};
\end{tikzpicture}
```

Shape `ellipse split`

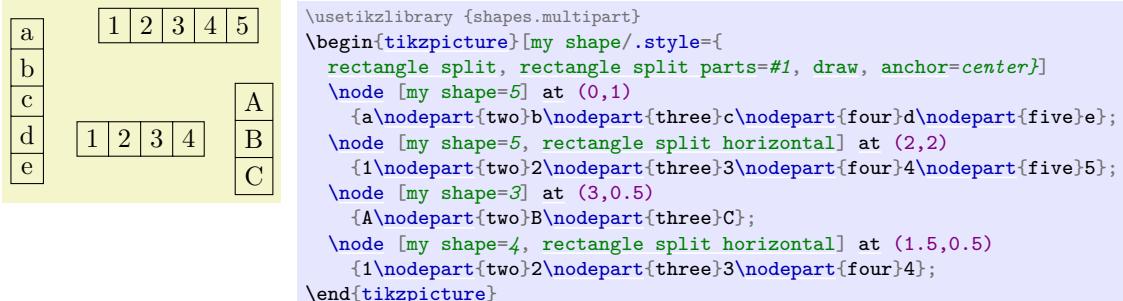
This shape is a multi-part shape consisting of an ellipse with a line in the middle. The upper part is the main part (the `text` part), the lower part is the `lower` part. The anchors for this shape are shown below. Anchor 60 is a border anchor.



```
\usetikzlibrary {shapes.multipart}
\Huge
\begin{tikzpicture}
\node [name=s,shape=ellipse split,shape example] {text\nodepart{lower}lower};
\foreach \anchor/\placement in
{center/below, text/left, lower/left, 60/above right,
 mid/above, mid east/above, mid west/above,
 base/right, base east/left, base west/right,
 north/above, south/below, east/below, west/below,
 north east/above, south east/below, south west/below, north west/above}
\draw[shift=(s.\anchor)] plot[mark=x] coordinates{(0,0)}
node[\placement] {\scriptsize\textrm{\texttt{(s.\anchor)}}};
\end{tikzpicture}
```

Shape `rectangle split`

This shape is a rectangle which can be split either horizontally or vertically into several parts.



The shape can be split into a maximum of twenty parts. However, to avoid allocating a lot of unnecessary boxes, PGF only allocates four boxes by default. To use the `rectangle split` shape with more than four boxes, the extra boxes must be allocated manually in advance (perhaps using `\newbox` or `\let`). The boxes take the form `\pgfnodepart<number>box`, where `<number>` is from the cardinal numbers `one`, `two`, `three`, ... and so on. `\pgfnodepartonebox` is special in that it is synonymous with `\pgfnodeparttextbox`. For compatibility with earlier versions of this shape, the boxes `\pgfnodeparttwobox`, `\pgfnodepartthreebox` and `\pgfnodepartfourbox`, can be referred to using the ordinal numbers: `\pgfnodepartsecondbox`, `\pgfnodepartthirdbox` and `\pgfnodepartfourthbox`. In order to facilitate the allocation of these extra boxes, the following key is provided:

`/pgf/rectangle split allocate boxes=<number>` (no default)

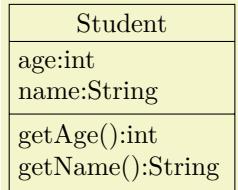
This key checks if `<number>` boxes have been allocated, and if not, it allocates the required boxes using `\newbox` (some “magic” is performed to get around the fact that `\newbox` is declared `\outer` in plain T_EX).

When split vertically, the rectangle split will meet any `minimum width` requirements, but any `minimum height` will be ignored. Conversely when split horizontally, `minimum height` requirements will be met, but any `minimum width` will be ignored. In addition, `inner sep` is applied to every part that is used, so it cannot be specified independently for a particular part.

There are several PGF keys to specify how the shape is drawn. To use these keys in TikZ, simply remove the `/pgf/` path:

`/pgf/rectangle split parts=<number>` (no default, initially 4)

Split the rectangle into `<number>` parts, which should be in the range 1 to 20. If more than four parts are needed, the boxes should be allocated in advance as described above.



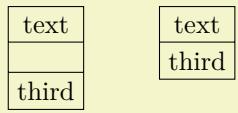
```
\usetikzlibrary {shapes.multipart}
\begin{tikzpicture}[every text node part/.style={align=center}]
\node[rectangle split, rectangle split parts=3, draw, text width=2.75cm]
  {Student
   \nodepart[two]
   age:int \\
   name:String
   \nodepart[three]
   getAge():int \\
   getName():String};
\end{tikzpicture}
```

`/pgf/rectangle split horizontal=<boolean>` (default `true`)

This key determines whether the rectangle is split horizontally or vertically

`/pgf/rectangle split ignore empty parts=<boolean>` (default `true`)

When `<boolean>` is true, PGF will ignore any part that is empty *except the text part*. This effectively overrides the `rectangle split parts` key in that, if 3 parts (for example) are specified, but one is empty, only two will be shown.



```
\usetikzlibrary {shapes.multipart}
\begin{tikzpicture}[every node/.style={draw, anchor=text, rectangle split,
  rectangle split parts=3}]
\node [text \nodepart[second] \nodepart[third] third];
\node [rectangle split ignore empty parts] at (2,0)
  {text \nodepart[second] \nodepart[third] third};
\end{tikzpicture}
```

`/pgf/rectangle split empty part width=<length>` (no default, initially `1ex`)

Sets the default width for a node part box if it is empty and empty parts are not ignored.

`/pgf/rectangle split empty part height=<length>` (no default, initially `1ex`)

Sets the default height for a node part box if it is empty and empty parts are not ignored.

`/pgf/rectangle split empty part depth=<length>` (no default, initially `0ex`)

Sets the default depth for a node part box if it is empty and empty parts are not ignored.

`/pgf/rectangle split part align={<list>}` (no default, initially `center`)

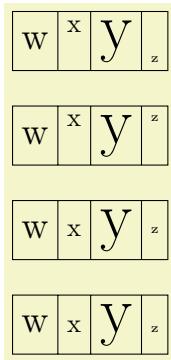
Sets the alignment of the boxes inside the node parts. Each item in `<list>` should be separated by commas (so if there is more than one item in `<list>`, it must be surrounded by braces).

When the rectangle is split vertically, the entries in `<list>` must be one of `left`, `right`, or `center`. If `<list>` has less entries than node parts then the remaining boxes are aligned according to the last entry in the list. Note that this only aligns the boxes in each part and *does not* affect the alignment of the contents of the boxes.

one	one	one
2	2	2
three	three	three
4	4	4

```
\usetikzlibrary {shapes.multipart}
\def\x{one \nodepart{two} 2 \nodepart{three} three \nodepart{four} 4}
\begin{tikzpicture}[
  every node/.style={rectangle split, rectangle split parts=4,
    draw}
]
\node[rectangle split part align={center, left, right}] at (0,0) {\x};
\node[rectangle split part align={center, left}] at (1.25,0) {\x};
\node[rectangle split part align={center}] at (2.5,0) {\x};
\end{tikzpicture}
```

When the rectangle is split horizontally, the entries in *<list>* must be one of **top**, **bottom**, **center** or **base**. Note that using the value **base** will only make sense if all the node part boxes are being aligned in this way. This is because the **base** value aligns the boxes in relation to each other, whereas the other values align the boxes in relation to the part of the shape they occupy.



```
\usetikzlibrary {shapes.multipart}
\def\x{Large w\nodepart{two}x\nodepart{three}\Huge y\nodepart{four}\tiny z}
\begin{tikzpicture}[
  every node/.style={rectangle split, rectangle split parts=4,
    draw, rectangle split horizontal}
]
\node[rectangle split part align={center, top, bottom}] at (0,0) {\x};
\node[rectangle split part align={center, top}] at (0,-1.25) {\x};
\node[rectangle split part align={center}] at (0,-2.5) {\x};
\node[rectangle split part align=base] at (0,-3.75) {\x};
\end{tikzpicture}
```

/pgf/rectangle split draw splits=(boolean) (default **true**)

Sets whether the line or lines between node parts will be drawn. Internally, this sets the TeX-if **\ifpgfrectanglesplitdrawsplits** appropriately.

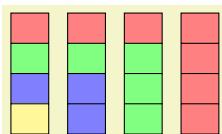
/pgf/rectangle split use custom fill=(boolean) (default **true**)

This enables the use of a custom fill for each of the node parts (including the area covered by the **inner sep**). The background path for the shape should not be filled (e.g., in TikZ, the **fill** option for the node must be implicitly or explicitly set to **none**). Internally, this key sets the TeX-if **\ifpgfrectanglesplitusecustomfill** appropriately.

/pgf/rectangle split part fill={<list>}

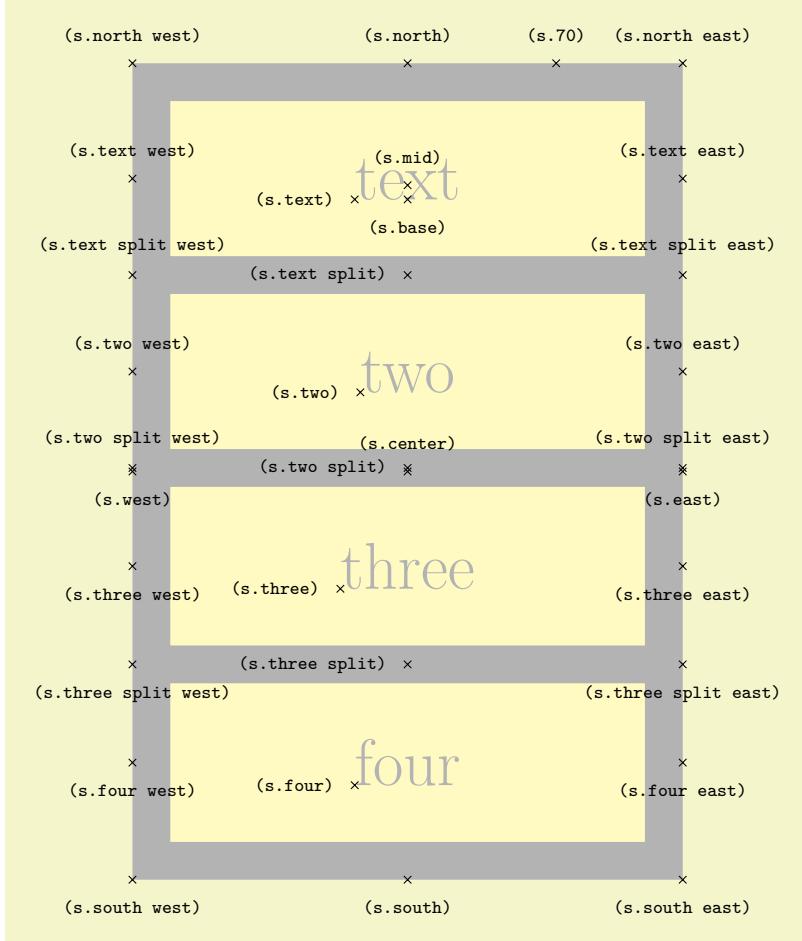
(no default, initially **white**)

Sets the custom fill color for each node part shape. The items in *<list>* should be separated by commas (so if there is more than one item in *<list>*, it must be surrounded by braces). If *<list>* has less entries than node parts, then the remaining node parts use the color from the last entry in the list. This key will automatically set **/pgf/rectangle split use custom fill**.



```
\usetikzlibrary {shapes.multipart}
\begin{tikzpicture}
\tikzset{every node/.style={rectangle split, draw, minimum width=.5cm}}
\node[rectangle split part fill={red!50, green!50, blue!50, yellow!50}] {};
\node[rectangle split part fill={red!50, green!50, blue!50}] at (0.75,0) {};
\node[rectangle split part fill={red!50, green!50}] at (1.5,0) {};
\node[rectangle split part fill={red!50}] at (2.25,0) {};
\end{tikzpicture}
```

The anchors for the **rectangle split** shape split vertically into four, are shown below (anchor 70 is an example of a border angle). When a node part is missing, the anchors prefixed with the name of that node part should be considered unavailable. They are (unavoidably) defined, but default to other anchor positions.



```
\usetikzlibrary {shapes.multipart}
\Huge
\begin{tikzpicture}
\node[name=s,shape=rectangle split, rectangle split parts=4, shape example,
inner ysep=0.75cm]
{\nodepart{text}text\nodepart{two}two
 \nodepart{three}three\nodepart{four}four};
\foreach \anchor/\placement in
{text/left, text east/above, text west/above,
two/left, two east/above, two west/above,
three/left, three east/below, three west/below,
four/left, four east/below, four west/below,
text split/left, text split east/above, text split west/above,
two split/left, two split east/above, two split west/above,
three split/left, three split east/below, three split west/below,
north/above, south/below, east/below, west/below,
north west/above, north east/above, south west/below, south east/below,
center/above, 70/above, mid/above, base/below}
\draw[shift=(s.\anchor)] plot[mark=x] coordinates{(0,0)}
node[\placement] {\scriptsize\textrm{\texttt{(s.\anchor)}}};
\end{tikzpicture}
```

72.7 Callout Shapes

TikZ Library `shapes.callouts`

```
\usepgflibrary{shapes.callouts} % LATEX and plain TEX and pure pgf
\usepgflibrary[shapes.callouts] % ConTeXt and pure pgf
\usetikzlibrary{shapes.callouts} % LATEX and plain TEX when using TikZ
\usetikzlibrary[shapes.callouts] % ConTeXt when using TikZ
```

Producing basic callouts can be done quite easily in PGF and TikZ by creating a node and then subsequently drawing a path from the border of the node to the required point. This library provides more

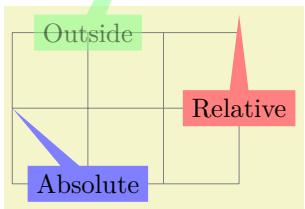
fancy, ‘balloon’-style callouts.

Callouts consist of a main shape and a pointer (which is part of the shape) which points to something in (or outside) the picture. The position on the border of the main shape to which the pointer is connected is determined automatically. However, the pointer is ignored when calculating the minimum size of the shape, and also when positioning anchors.



```
\usetikzlibrary {shapes.callouts}
\begin{tikzpicture} [remember picture]
  \node [ellipse callout, draw] (hallo) {Hallo!};
\end{tikzpicture}
```

There are two kinds of pointer: the “relative” pointer and the “absolute” pointer. The relative pointer calculates the angle of a specified coordinate relative to the center of the main shape, locates the point on the border to which this angle corresponds, and then adds the coordinate to this point. This seemingly over-complex approach means than you do not have to guess the size of the main shape: the relative pointer will always be outside. The absolute pointer, on the other hand, is much simpler: it points to the specified coordinate absolutely (and can even point to named coordinates in different pictures).



```
\usetikzlibrary {shapes.callouts}
\begin{tikzpicture} [remember picture, note/.style={rectangle callout, fill=#1}]
  \draw [help lines] grid(3,2);
  \node [note=red!50, callout relative pointer={(0,1)}] at (3,1) {Relative};
  \node [note=blue!50, callout absolute pointer={(0,1)}] at (1,0) {Absolute};
  \node [note=green!50, opacity=.5, overlay,
        callout absolute pointer={(hallo.south)}] at (1,2) {Outside};
\end{tikzpicture}
```

The following keys are common to all callouts. Please remember that the `callout relative pointer`, and `callout absolute pointer` keys take a different format for their value depending on whether they are being used in PGF or TikZ.

`/pgf/callout relative pointer=<coordinate>` (no default, initially `\pgfpointpolar{315}{.5cm}`)

Sets the vector of the callout pointer ‘relative’ to the callout shape.

`/pgf/callout absolute pointer=<coordinate>` (no default)

Sets the vector of the callout pointer absolutely within the picture.

`/tikz/callout relative pointer=<coordinate>` (no default, initially `(315:.5cm)`)

The TikZ version of the `callout relative pointer` key. Here, `<coordinate>` can be specified using the TikZ format for coordinates.

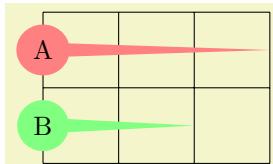
`/tikz/callout absolute pointer=<coordinate>` (no default)

The TikZ version of the `callout absolute pointer` key. Here, `<coordinate>` can be specified using the TikZ format for coordinates.

It is also possible to shorten the pointer by some distance, using the following key:

`/pgf/callout pointer shorten=<distance>` (no default, initially `0cm`)

Moves the callout pointer towards the center of the callout’s main shape by `<distance>`.



```
\usetikzlibrary {shapes.callouts}
\begin{tikzpicture}
  \tikzset{callout/.style={ellipse callout, callout pointer arc=30,
    callout absolute pointer={#1}}}
  \draw (0,0) grid (3,2);
  \node[callout={(3,.15)}, fill=red!50] at (0,.15) {A};
  \node[callout={(3,.5)}, fill=green!50, callout pointer shorten=1cm]
    at (0,.5) {B};
\end{tikzpicture}
```

Shape `rectangle callout`

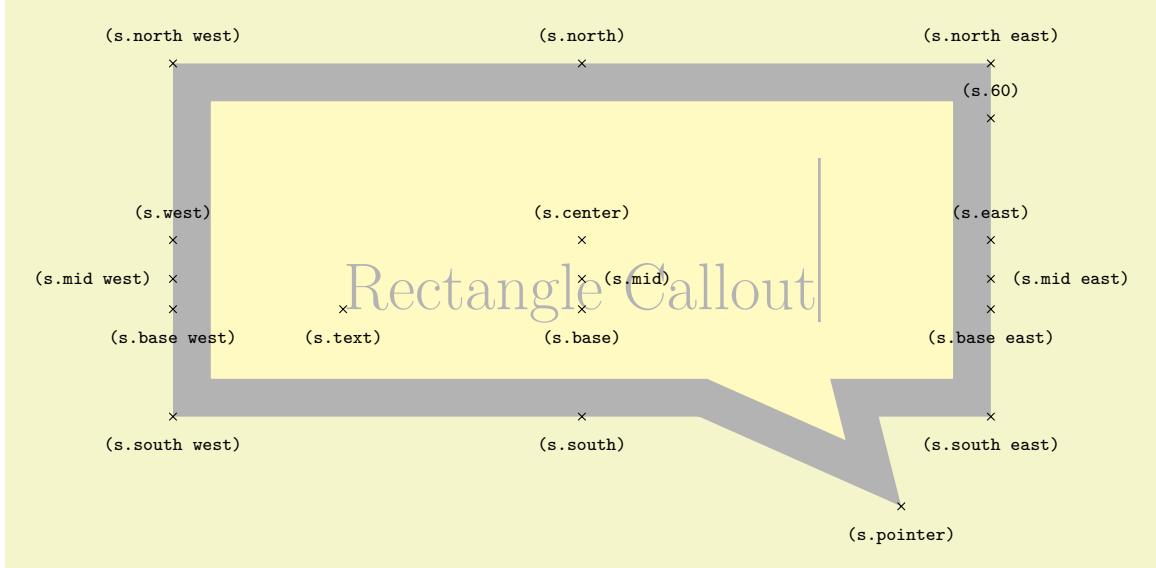
This shape is a callout whose main shape is a rectangle, which tightly fits the node contents (including any `inner sep`). It supports the keys described above and also the following key:

`/pgf/callout pointer width=<length>`

(no default, initially `.25cm`)

Sets the width of the pointer at the border of the rectangle.

The anchors for this shape are shown below (anchor 60 is an example of a border anchor). The pointer direction is ignored when placing anchors. Additionally, when using an absolute pointer, the `pointer` anchor should not be used to position the shape as it is calculated whilst the shape is being drawn.



```
\usetikzlibrary {shapes.callouts}
\Huge
\begin{tikzpicture}
\node[name=s,shape=rectangle callout, callout relative pointer={(1.25cm,-1cm)},
      callout pointer width=2cm, shape example, inner xsep=2cm, inner ysep=1cm]
      {Rectangle Callout\text{vrule} width 1pt height 2cm};
\foreach \anchor/\placement in
{center/above, text/below, 60/above,
 mid/right, mid west/left, mid east/right,
 base/below, base west/below, base east/below,
 north/above, south/below, east/above, west/above,
 north west/above, north east/above,
 south west/below, south east/below,
 pointer/below}
\draw[shift=(s.\anchor)] plot[mark=x] coordinates{(0,0)}
  node[\placement] {\scriptsize\textrm{\texttt{(s.\anchor)}}};
\end{tikzpicture}
```

Shape `ellipse callout`

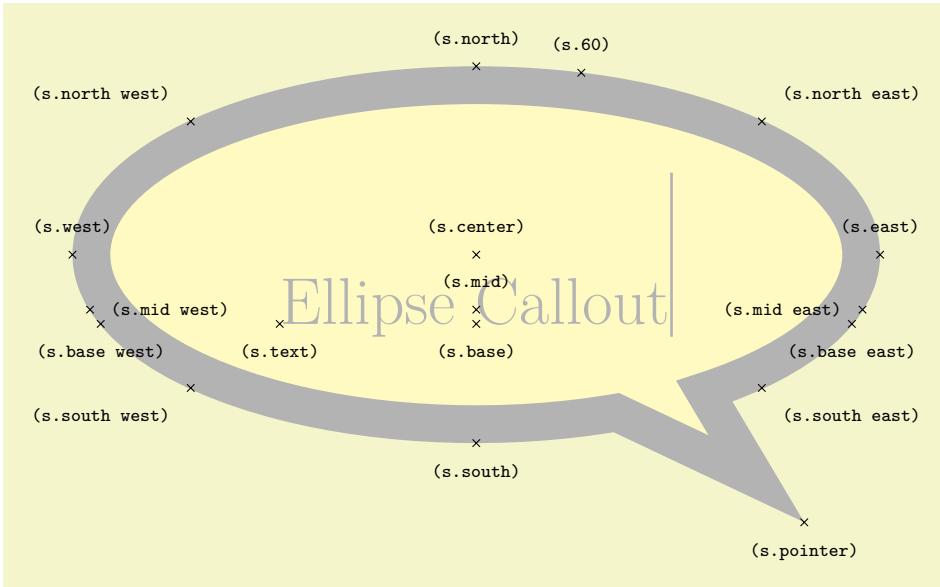
This shape is a callout whose main shape is an ellipse, which tightly fits the node contents (including any `inner sep`). It uses the `absolute callout pointer`, `relative callout pointer` and `callout pointer shorten` keys, and also the following key:

`/pgf/callout pointer arc=<angle>`

(no default, initially `15`)

Sets the width of the pointer at the border of the ellipse according to an arc of length `<angle>`.

The anchors for this shape are shown below (anchor 60 is an example of a border anchor). The pointer direction is ignored when placing anchors and the `pointer` anchor can only be used to position the shape when the relative anchor is specified.



```
\usetikzlibrary {shapes.callouts}
\Huge
\begin{tikzpicture}
\node [name=s,shape=ellipse callout, callout relative pointer={(1.25cm,-1cm)},
callout pointer width=2cm, shape example, inner xsep=1cm, inner ysep=.5cm]
{Ellipse Callout\text{\vrule width 1pt height 2cm}};
\foreach \anchor/\placement in
{center/above, text/below, 60/above,
mid/above, mid west/right, mid east/left,
base/below, base west/below, base east/below,
north/above, south/below, east/above, west/above,
north west/above left, north east/above right,
south west/below left, south east/below right,
pointer/below}
\draw [shift=(s.\anchor)] plot [mark=x] coordinates{(0,0)};
\node[\placement] {\scriptsize\textrm{\texttt{(s.\anchor)}}};
\end{tikzpicture}
```

Shape `cloud callout`

This shape is a callout whose main shape is a cloud which fits the node contents. The pointer is segmented, consisting of a series of shrinking ellipses. This callout requires the `shapes.callouts` library (for the cloud shape). If this library is not loaded an error will result.



```
\usetikzlibrary {shapes.callouts}
\begin{tikzpicture}
\node [cloud callout, cloud puffs=15, aspect=2.5, cloud puff arc=120,
shading=ball, text=white] {\bf Imagine...};
\end{tikzpicture}
```

The `cloud callout` supports the `absolute callout pointer`, `relative callout pointer` and `callout pointer shorten` keys, as described above. The main shape can be modified using the same keys as the `cloud` shape. The following keys are also supported:

`/pgf/callout pointer start size=<value>` (no default, initially .2 of callout)

Sets the size of the first segment in the pointer (i.e., the segment nearest the main cloud shape). There are three possible forms for `<value>`:

- A single dimension (e.g., 5pt), in which case the first ellipse will have equal diameters of 5pt.
- Two dimensions (e.g., 10pt and 2.5pt), which sets the *x* and *y* diameters of the first ellipse.
- A decimal fraction (e.g., .2 of callout), in which case the *x* and *y* diameters of the first ellipse will be set as fractions of the width and height of the main shape. The keyword of `callout` cannot be omitted.

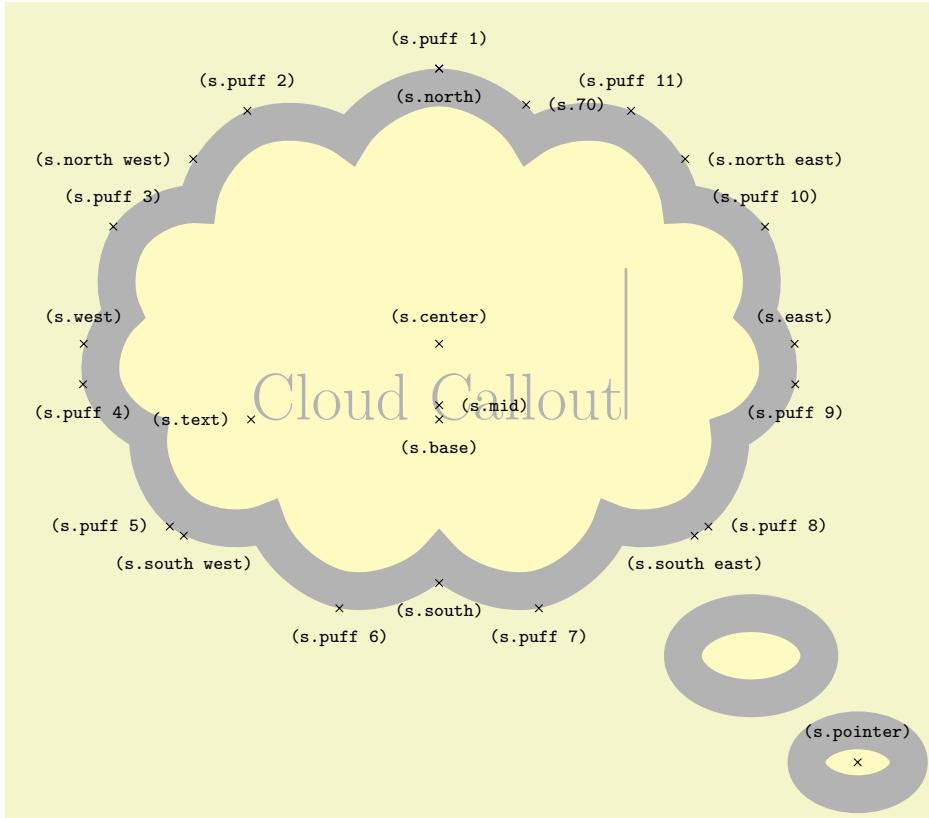
`/pgf/callout pointer end size=<value>` (no default, initially .1 of callout)

Sets the size of the last ellipse in the pointer.

`/pgf/callout pointer segments=<number>` (no default, initially 2)

Sets the number of segments in the pointer. Note that PGF will happily overlap segments if too many are specified.

The anchors for this shape are shown below (anchor 70 is an example of a border anchor). The pointer direction is ignored when placing anchors and the pointer anchor can only be used to position the shape when the relative anchor is specified. Note that the center of the last segment is drawn at the `pointer` anchor.



```
\usetikzlibrary {shapes.callouts}
\Huge
\begin{tikzpicture}
\node[name=s, shape=cloud callout, style=shape example, cloud puffs=11, aspect=1.5,
    cloud puff arc=120, inner xsep=.5cm, callout pointer start size=.25 of callout,
    callout pointer end size=.15 of callout, callout relative pointer={f(315:4cm)},
    callout pointer segments=2] {Cloud Callout\hrule width 1pt height 2cm};
\foreach \anchor/\placement in
{puff 1/above, puff 2/above, puff 3/above, puff 4/below,
puff 5/left, puff 6/below, puff 7/below, puff 8/right,
puff 9/below, puff 10/above, puff 11/above, 70/right,
center/above, base/below, mid/right, text/left,
north/below, south/below, east/above, west/above,
north west/left, north east/right,
south west/below, south east/below,pointer/above}
\draw[shift=(s.\anchor)] plot[mark=x] coordinates{(0,0)}
node[\placement] {\scriptsize\tt(s.\anchor)};
\end{tikzpicture}
```

72.8 Miscellaneous Shapes

TikZ Library `shapes.mis`

```
\usepgflibrary{shapes.mis} % LATEX and plain TEX and pure pgf
\usepgflibrary[shapes.mis] % ConTeXt and pure pgf
\usetikzlibrary{shapes.mis} % LATEX and plain TEX when using TikZ
\usetikzlibrary[shapes.mis] % ConTeXt when using TikZ
```

This library defines general-purpose shapes that do not fit into the previous categories.

Shape `cross out`

This shape “crosses out” the node. Its foreground path are simply two diagonal lines between the corners of the node’s bounding box. Here is an example:



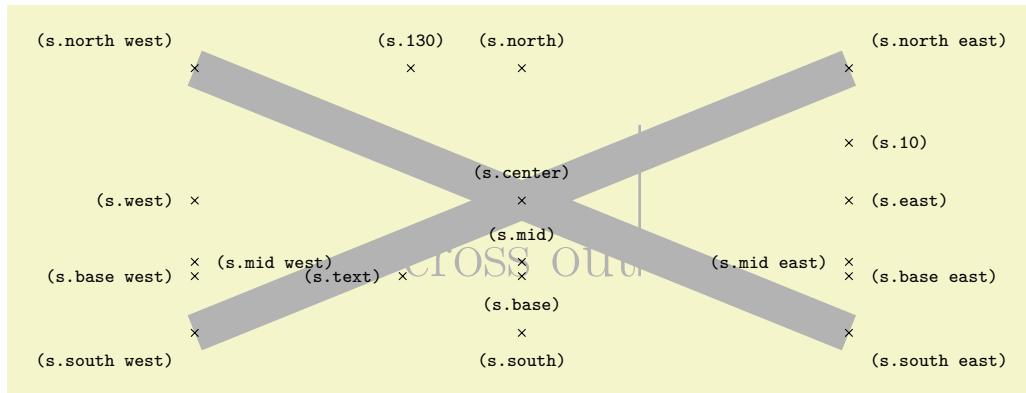
```
\usetikzlibrary {shapes.mis}
\begin{tikzpicture}
  \draw [help lines] (0,0) grid (3,2);
  \node [cross out,draw=red] at (1.5,1) {cross out};
\end{tikzpicture}
```

A useful application is inside text as in the following example:

Cross ~~me~~ out!

```
\usetikzlibrary {shapes.mis}
Cross \tikz[baseline] \node [cross out,draw,anchor=text] {me}; out!
```

This shape inherits all anchors from the `rectangle` shape, see also the following figure:



```
\usetikzlibrary {shapes.mis}
\Huge
\begin{tikzpicture}
  \node[name=s,shape=cross out,shape example] {cross out\text{\vrule width 1pt height 2cm}};
  \foreach \anchor/\placement in
    {north west/above left, north/above, north east/above right,
     west/left, center/above, east/right,
     mid west/right, mid/above, mid east/left,
     base west/left, base/below, base east/right,
     south west/below left, south/below, south east/below right,
     text/left, 10/right, 130/above}
    \draw[shift=(s.\anchor)] plot[mark=x] coordinates{(0,0)};
    \node[\placement] {\scriptsize\texttt{(s.\anchor)}};
\end{tikzpicture}
```

Shape `strike out`

This shape is identical to the `cross out` shape, only its foreground path consists of a single line from the lower left to the upper right.

Strike ~~me~~ out!

```
\usetikzlibrary {shapes.mis}
Strike \tikz[baseline] \node [strike out,draw,anchor=text] {me}; out!
```

See the `cross out` shape for the anchors.

Shape rounded rectangle

This shape is a rectangle which can have optionally rounded sides.

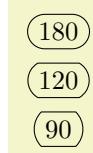


```
\usetikzlibrary {shapes.mis}
\begin{tikzpicture}
  \node[rounded rectangle, draw, fill=red!20]{Hallo};
\end{tikzpicture}
```

There are keys to specify how the sides are rounded (to use these keys in TikZ, simply remove the `/pgf/` path).

`/pgf/rounded rectangle arc length=<angle>` (no default, initially 180)

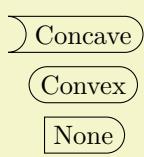
Sets the length of the arcs for the rounded ends. Recommended values for `<angle>` are between 90 and 180.



```
\usetikzlibrary {shapes.mis}
\begin{tikzpicture}
  \matrix [row sep=5pt, every node/.style={draw, rounded rectangle}]{%
    \node[rounded rectangle arc length=180] {180}; \\
    \node[rounded rectangle arc length=120] {120}; \\
    \node[rounded rectangle arc length=90] {90}; \\
  };
\end{tikzpicture}
```

`/pgf/rounded rectangle west arc=<arc type>` (no default, initially convex)

Sets the style of the rounding for the left side. The permitted values for `<arc type>` are `concave`, `convex`, or `none`.



```
\usetikzlibrary {shapes.mis}
\begin{tikzpicture}
  \matrix [row sep=5pt, every node/.style={draw, rounded rectangle}]{%
    \node[rounded rectangle west arc=concave] {Concave}; \\
    \node[rounded rectangle west arc=convex] {Convex}; \\
    \node[rounded rectangle left arc=none] {None}; \\
  };
\end{tikzpicture}
```

`/pgf/rounded rectangle left arc=<arc type>` (style, no default)

Alternative key for specifying the west arc.

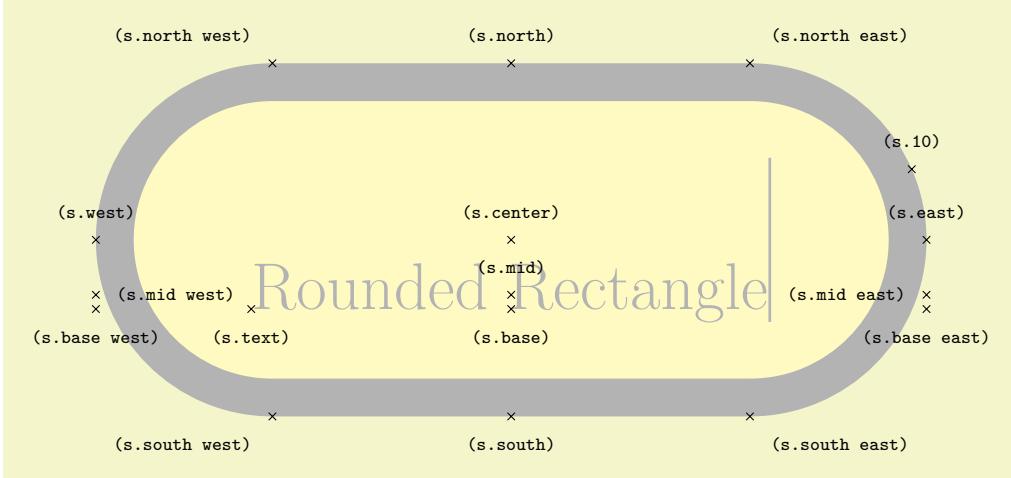
`/pgf/rounded rectangle east arc=<arc type>` (no default, initially convex)

Sets the style of the rounding for the east side.

`/pgf/rounded rectangle right arc=<arc type>` (style, no default)

Alternative key for specifying the east arc.

The anchors for this shape are shown below (anchor 10 is an example of a border angle). Note that if only one side is rounded, the center anchor will not be the precise center of the shape.



```
\usetikzlibrary {shapes.mis}
\Huge
\begin{tikzpicture}
\node [name=s,shape=rounded rectangle, shape example, inner xsep=1.5cm, inner ysep=1cm]
  {Rounded Rectangle\vrule width 1pt height 2cm};
\foreach \anchor/\placement in
  {center/above, text/below,      10/above,
   mid/above,    mid west/right, mid east/left,
   base/below,   base west/below, base east/below,
   north/above,  south/below,   east/above, west/above,
   north west/above left,  north east/above right,
   south west/below left,  south east/below right}
  \draw[shift=(s.\anchor)] plot[mark=x] coordinates{(0,0)};
  node[\placement] {\scriptsize\textrm{\texttt{(s.\anchor)}}};
\end{tikzpicture}
```

Shape `chamfered rectangle`

This shape is a rectangle with optionally chamfered corners.



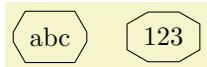
```
\usetikzlibrary {shapes.mis}
\begin{tikzpicture}
\node[chamfered rectangle, white, fill=red, double=red, draw, very thick]
  {\bf STOP!};
\end{tikzpicture}
```

There are PGF keys to specify how this shape is drawn (to use these keys in TikZ simply remove the `/pgf/` path).

`/pgf/chamfered rectangle angle=<angle>`

(no default, initially 45)

Sets the angle *from the vertical* for the chamfer.



```
\usetikzlibrary {shapes.mis}
\begin{tikzpicture}
\tikzset{every node/.style={chamfered rectangle, draw}}
\node[chamfered rectangle angle=30] {abc};
\node[chamfered rectangle angle=60] at (1.5,0) {123};
\end{tikzpicture}
```

`/pgf/chamfered rectangle xsep=<length>`

(no default, initially .666ex)

Sets the distance that the chamfer extends horizontally beyond the node contents (which includes the `inner sep`). If `<length>` is large, such that the top and bottom chamfered edges would cross, then `<length>` is ignored and the chamfered edges are drawn so that they meet in the middle.



```
\usetikzlibrary {shapes.mis}
\begin{tikzpicture}
\tikzset{every node/.style={chamfered rectangle, draw}}
\node[chamfered rectangle xsep=2pt] {def};
\node[chamfered rectangle xsep=2cm] at (1.5,0) {456};
\end{tikzpicture}
```

`/pgf/chamfered rectangle ysep=<length>`

(no default, initially `.666ex`)

Sets the distance that the chamfer extends vertically beyond the node contents. If `<length>` is large, such that the left and right chamfered edges would cross, then `<length>` is ignored and the chamfered edges are drawn so that they meet in the middle.

`/pgf/chamfered rectangle sep=<length>`

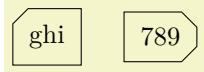
(no default, initially `.666ex`)

Sets both the `xsep` and `ysep` simultaneously.

`/pgf/chamfered rectangle corners=<list>`

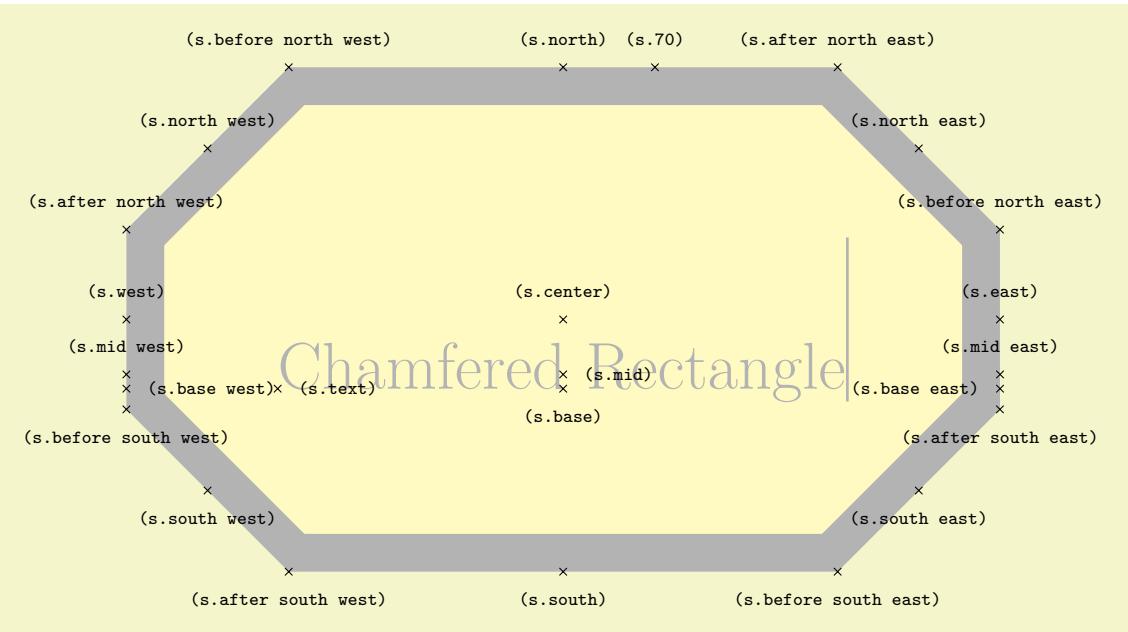
(no default, initially `chamfer all`)

Specifies which corners are chamfered. The corners are identified by their “compass point” directions (i.e. `north east`, `north west`, `south west`, and `south east`), and must be separated by commas (so if there is more than one corner in the list, it must be surrounded by braces). Any corners not mentioned in `<list>` are automatically not chamfered. Two additional values `chamfer all` and `chamfer none`, are also permitted.



```
\usetikzlibrary {shapes.mis}
\begin{tikzpicture}
  \tikzset{every node/.style={chamfered rectangle, draw}}
  \node[chamfered rectangle corners=north west] {ghi};
  \node[chamfered rectangle corners={north east, south east}] at (1.5,0) {789};
\end{tikzpicture}
```

The anchors for this shape are shown below (anchor 60 is an example of a border angle).



```
\usetikzlibrary {shapes.mis}
\Huge
\begin{tikzpicture}
  \node[name=s,shape=chamfered rectangle, chamfered rectangle sep=1cm,
    shape example, inner ysep=1cm, inner xsep=.75cm]
  {Chamfered Rectangle\hrule width1pt height2cm};
  \foreach \anchor/\placement in
  {text/right, center/above, 70/above,
  base/below, base east/left, base west/right,
  mid/right, mid east/above, mid west/above,
  north/above, south/below, east/above, west/above,
  before north east/above, north east/above, after north east/above,
  before north west/above, north west/above, after north west/above,
  before south west/below, south west/below, after south west/below,
  before south east/below, south east/below, after south east/below}
  \draw[shift=(s.\anchor)] plot [mark=x] coordinates{(0,0)};
  node[\placement] {\scriptsize\texttt{(s.\anchor)}};
\end{tikzpicture}
```

73 Spy Library: Magnifying Parts of Pictures

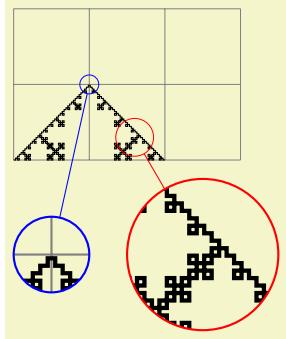
TikZ Library `spy`

```
\usetikzlibrary{spy} % LATEX and plain TEX  
\usetikzlibrary[spy] % ConTEXt
```

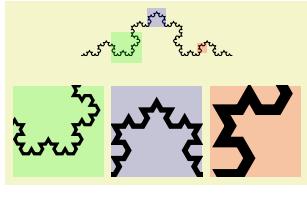
The package defines options for creating pictures in which some part of the picture is repeated in another area in a magnified way (as if you were looking through a spyglass, hence the name).

73.1 Magnifying a Part of a Picture

The idea behind the `spy` library is to make it easy to create high-density pictures in which some important parts are repeated somewhere, but magnified as if you were looking through a spyglass:



```
\usetikzlibrary {decorations.fractals,spy}  
\begin{tikzpicture}[spy using outlines={circle, magnification=4, size=2cm, connect spies}]  
    \draw [help lines] (0,0) grid (3,2);  
  
    \draw [decoration=Koch curve type 1]  
        decorate { decorate[ decorate[ decorate[ (0,0) -- (2,0) ] ] ] };  
  
    \spy [red] on (1.6,0.3)  
        in node [left] at (3.5,-1.25);  
  
    \spy [blue, size=1cm] on (1,1)  
        in node [right] at (0,-1.25);  
\end{tikzpicture}
```



```
\usetikzlibrary {decorations.fractals,spy}  
\begin{tikzpicture}[spy using overlays={size=12mm}]  
    \draw [decoration=Koch snowflake]  
        decorate { decorate[ decorate[ decorate[ (0,0) -- (2,0) ] ] ] };  
  
    \spy [green,magnification=3] on (0.6,0.1) in node at (-0.3,-1);  
    \spy [blue,magnification=5] on (1,0.5) in node at (1,-1);  
    \spy [red,magnification=10] on (1.6,0.1) in node at (2.3,-1);  
\end{tikzpicture}
```

Note that this magnification uses what is called a *canvas transformation* in this manual: Everything is magnified, including line width and text.

In order for “spying” to work, the picture obviously has to be drawn several times: Once at its normal size and then again for each “magnifying glass”. Several keys and commands work in concert to make this possible:

- You need to make TikZ aware of the fact that a picture (or just a scope) is to be magnified. This is done by adding the special key `spy scope` to a `{scope}` or `{tikzpicture}` (which is also just a scope). Some special keys like `spy using outlines` implicitly set the `spy scope`.
- Inside this scope you may then use the command `\spy`, which is only available inside such scopes (so there is no danger of you inadvertently using this command outside such a scope). This command has a special syntax and will (at some point) create two nodes: One node that shows the magnified picture (called the *spy-in node*) and another node showing which part of the original picture is magnified (called the *spy-on node*). The spy-in node is, indeed, a normal node, so it can have any shape or border that you like and you can apply all of TikZ’s advanced features to it. The only difference compared to a normal node is that instead of some “text” it contains a magnified version of the picture, clipped to the size of the node.

The `\spy` command does not create the nodes immediately. Rather, the creation of these nodes is postponed till the end of the `spy scope` in which the `\spy` command is used. This is necessary since in order to repeat the whole scope inside the node containing the magnified version, the whole picture needs to be available when this node is created.

A basic question any library for “magnifying things” has to address is how you specify which part of the picture is to be magnified (the *spy-on node*) and where this magnified part is to be shown (the *spy-in node*). There are two possible ways:

1. You specify the size and position of the spy-on node. Then the size of the spy-in node is determined by the size of the spy-on node and the magnification factor – you can still decide where the spy-in node should be placed, but not its size.
2. Alternatively, you specify the size and position of the spy-in node. Then, similarly to the first case, the size of the spy-on node is determined implicitly and you can only decide where the spy-on node should be placed, but not its size.

The `spy` library uses the second method: You specify the size and position of the spy-in nodes, the sizes of the spy-on nodes are then computed automatically.

73.2 Spy Scopes

`/tikz/spy scope=<options>` (default empty)

This option may be used with a `{scope}` or any environment that creates such a scope internally (like `{tikzpicture}`). It has the following effects:

- It resets a number of graphic state parameters, including the color, line style, and others. This is necessary for technical reasons.
- It tells TikZ that the content of the scope should be saved internally in a special box.
- It defines the command `\spy` so that it can be used inside the scope.
- At the end of the scope, the nodes belonging to the `\spy` commands used inside the scope are created.
- The `<options>` are saved in an internal style. Each time `\spy` is used, these `<options>` will be used.
- Three keys are defined that provide useful shortcuts:

`/tikz/size=<dimension>` (no default)

Inside a `spy scope`, this is a shortcut for `minimum size`.

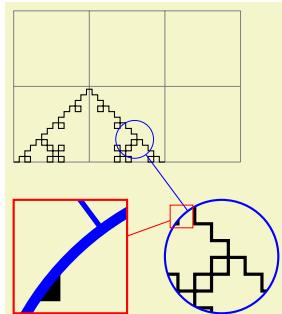
`/tikz/height=<dimension>` (no default)

Inside a `spy scope`, this is a shortcut for `minimum height`.

`/tikz/width=<dimension>` (no default)

Inside a `spy scope`, this is a shortcut for `minimum width`.

It is permissible to nest `spy scopes`. In this case, all `\spy` commands inside the inner `spy scope` only have an effect on material inside the scope, whereas `\spy` commands outside the inner `spy scope` but inside the outer `spy scope` allow you to “spy on the spy”.



```
\usetikzlibrary {decorations.fractals,spy}
\begin{tikzpicture}
  \spy using outlines={rectangle, red, magnification=5,
    size=1.5cm, connect spies};

  \begin{scope}
    \spy using outlines={circle, blue,
      magnification=3, size=1.5cm, connect spies};
    \draw [help lines] (0,0) grid (3,2);
    \draw [decoration=Koch curve type 1]
      decorate{ decorate{ decorate{ (0,0) -- (2,0) }}};

    \spy on (1.6,0.3) in node (zoom) [left] at (3.5,-1.25);
  \end{scope}

  \spy on (zoom.north west) in node [right] at (0,-1.25);
\end{tikzpicture}
```

73.3 The Spy Command

`\spy[<options>] on <coordinate> in node <node options>;`

This command can only be used inside a `spy scope`. Let us start with the syntax:

- The `\spy` command is not a special case of `\path`. Rather, it has a small parser of its own.
- Following the optional `<options>`, you must write `on`, followed by a coordinate. This coordinate will be the center of the area that is to be magnified.
- Following the `<coordinate>`, you must write `in node` followed by some `<node options>`. The syntax for these options is the same as for a normal `node` path command, such as `[left]` or `(foo) [red] at (bar)`. However, `<node options>` are not followed by a curly brace. Rather, the `<node options>` must directly be followed by a semicolon.

The effect of this command is the following: The `<options>`, `<coordinate>`, and `<node options>` are stored internally till the end of the current `spy scope`. This means that, in particular, you can reference any node inside the `spy scope`, even if it is not yet defined when the `\spy` command is given. At the end of the current `spy scope`, two nodes are created, called the *spy-in node* and the *spy-on node*.

- The *spy-in node* is the node that contains a magnified part of the picture (the node *in* which we see on what we spy). This node is, indeed, a normal TikZ node, so you can use all standard options to style this node. In particular, you can specify a shape or a border color or a drop shadow or whatever. The only thing that is special about this node is that instead of containing some normal text, its “text” is the magnified picture.

To be precise, the picture of the `spy scope` is scaled by a certain factor, specified by the `lens` or `magnification` options discussed below, and is shifted in such a way that the `<coordinate>` lies at the center of the spy-on node.

- The *spy-on node* is a node that is centered on the `<coordinate>` and whose size reflects exactly the area shown inside the spy-in node (the node containing *on* what we spy).

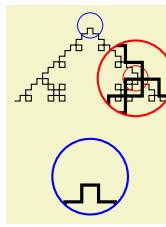
Let us now go over what happens in detail when the two nodes are created:

1. A scope is started. Two sets of options are used with this scope: First, the options passed to the enclosing `spy scope` and then the `<options>` (which will, thus, overrule the options of the `spy scope`).
2. Then, the spy-on node is created. However, we will first discuss the spy-in node.
3. The spy-in node is created after the spy-on node (and, hence, will cover the spy-on node in case they overlap). When this node is created, the `<node options>` are used in addition to the effect caused by the `<options>` and the options of the `{spy scope}`. Additionally, the following style is used:

/tikz/every spy in node (style, no value)

This style is used with every spy-in node.

The position of the node (the `at` option) is set to the `<coordinate>` by default, so that it will cover the to-be-magnified area. You can change this by providing the `at` option yourself:



```
\usetikzlibrary {decorations.fractals,spy}
\begin{tikzpicture}
  \spy using outlines={circle, magnification=3, size=1cm}
    [decoration=Koch curve type 1]
    draw [decoration=Koch curve type 1]
    decorate[ decorate[ decorate[ decorate[ (0,0) -- (2,0) ]]];
  \spy [red] on (1.6,0.3) in node;
  \spy [blue] on (1,1) in node at (1,-1);
\end{tikzpicture}
```

No “text” can be specified for the node. Rather, the “text” shown inside this node is the picture of the current `spy scope`, but canvas-transformed according to the following key:

/tikz/lens=<options> (no default)

The `<options>` should contain transformation commands like `scale` or `rotate`. These transformations are applied to the picture when it is shown inside the spy-on node.

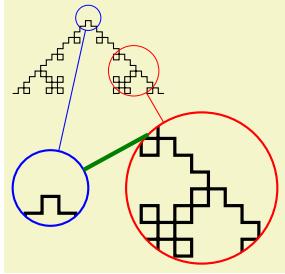
Since the most common transformation is undoubtedly a simple scaling, there is a special style for this:

/tikz/magnification=<number> (no default)

This has the same effect as saying `lens={scale=<number>}`.

Now, usually the size of a node is determined in such a way that it “fits” around the text of the node. For a spy-on node this is not a good approach since the “text” of this node would contain “the whole picture”. Because of this, TikZ acts as if the “text” of the node has zero size. You must then use keys like `minimum size` to cause the node to have a certain size. Note that the key `size` is an abbreviation for `minimum size` inside a spy scope.

You can name the spy-on node in the usual ways. Additionally, the node is (also) always named `tikzspyinnode`. Following the spy scope, you can use this node like any other node:



```
\usetikzlibrary {decorations.fractals,spy}
\begin{tikzpicture}
\begin{scope}
[spy using outlines=fcircle, magnification=3, size=2cm, connect spies]

\draw [decoration=Koch curve type 1]
  decorate{ decorate{ decorate{ (0,0) -- (2,0) }}};

\spy [red] on (1.6,0.3) in node (a) [left] at (3.5,-1.25);

\spy [blue, size=1cm] on (1,1) in node (b) [right] at (0,-1.25);
\end{scope}
\draw [ultra thick, green!50!black] (b) -- (a.north west);
\end{tikzpicture}
```

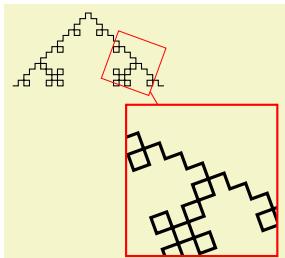
- Once both nodes have been created, the current value of the following key is used to connect them:

`/tikz/spy connection path=<code>` (no default, initially empty)

The `<code>` is executed after the spy-on and spy-in nodes have just been created. Inside this `<code>`, the two nodes can be accessed as `tikzspyinnode` and `tikzspyonnnode`. For example, the key `connect spies` sets this command to

```
\draw[thin] (tikzspyonnnode) -- (tikzspyinnnode);
```

Returning to the creation of the spy-in node: This node is centered on `<coordinate>` (more precisely, its anchor is set to `center` and the `at` option is set to `<coordinate>`). Its size and shape are initially determined in the same way as the size and shape of the spy-on node (unless, of course, you explicitly provide a different shape for, say, the spy-on node locally, which is not really a good idea). Then, additionally, the *inverted* transformation done by the `lens` option is applied, resulting in a node whose size and shape exactly corresponds to the area in the picture that is shown in the spy-on node.



```
\usetikzlibrary {decorations.fractals,spy}
\begin{tikzpicture}
\begin{scope}
[spy using outlines={lens={scale=3,rotate=20}, size=2cm, connect spies}

\draw [decoration=Koch curve type 1]
  decorate{ decorate{ decorate{ (0,0) -- (2,0) }}};

\spy [red] on (1.6,0.3) in node at (2.5,-1.25);
\end{scope}
\end{tikzpicture}
```

Like for the spy-in node, a style can be used to format the spy-on node:

`/tikz/every spy on node` (style, no value)

This style is used with every spy-on node.

The spy-on node is named `tikzspyonnnode` (but, as always, this node is only available after the spy scope). If you have multiple spy-on nodes and you would like to access all of them, you need to use the `name` key inside the `every spy on node` style.

The `inner sep` and `outer sep` of both spy-in and spy-on nodes are set to `0pt`.

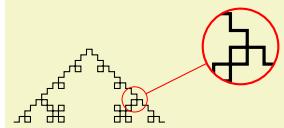
73.4 Predefined Spy Styles

There are some predefined styles that make using the `spy` library easier. The following two styles can be used instead of `spy scope`, they pass their `<options>` directly to `spy scope`. They additionally set up the graphic styles to be used for the spy-in nodes and the spy-on nodes in some special way.

`/tikz/spy using outlines=(options)`

(default empty)

This key creates a `spy scope` in which the spy-in node is drawn, but not filled, using a thick line; and the spy-on node is drawn, but not filled, using a very thin line.

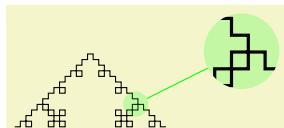


```
\usetikzlibrary {decorations.fractals,spy}
\begin{tikzpicture}
  [spy using outlines={circle, magnification=3, size=1cm, connect spies}]
  \draw [decoration=Koch curve type 1]
    decorate{ decorate{ decorate{ (0,0) -- (2,0) }}};
  \spy [red] on (1.6,0.3) in node at (3,1);
\end{tikzpicture}
```

`/tikz/spy using overlays=(options)`

(default empty)

This key creates a `spy scope` in which both the spy-in node and spy-on nodes are filled, but with the fill opacity set to 20%.



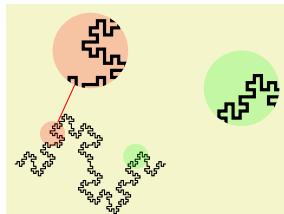
```
\usetikzlibrary {decorations.fractals,spy}
\begin{tikzpicture}
  [spy using overlays={circle, magnification=3, size=1cm, connect spies}]
  \draw [decoration=Koch curve type 1]
    decorate{ decorate{ decorate{ (0,0) -- (2,0) }}};
  \spy [green] on (1.6,0.3) in node at (3,1);
\end{tikzpicture}
```

The following style is useful for connecting the spy-in and the spy-on nodes:

`/tikz/connect spies`

(no value)

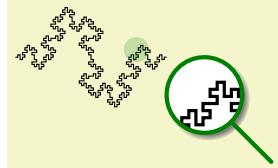
Causes the spy-in and the spy-on nodes to be connected by a thin line.



```
\usetikzlibrary {decorations.fractals}
\begin{tikzpicture}
  [spy using overlays={circle, magnification=3, size=1cm}]
  \draw [decoration=Koch curve type 2]
    decorate{ decorate{ decorate{ (0,0) -- (2,0) }}};
  \spy [green] on (1.6,0.1) in node at (3,1);
  \spy [red,connect spies] on (0.5,0.4) in node at (1,1.5);
\end{tikzpicture}
```

73.5 Examples

Usually, the spy-in node and the spy-on node should have the same shape. However, you might also wish to use the `circle` shape for the spy-on node and the `magnifying glass` shape for the spy-in node:



```
\usetikzlibrary {decorations.fractals,shadows,shapes.symbols,spy}
\tikzset{spy using mag glass/.style={
  spy scope={
    every spy on node/.style={
      circle,
      fill, fill opacity=0.2, text opacity=1},
    every spy in node/.style={
      magnifying glass, circular drop shadow,
      fill=white, draw, ultra thick, cap=round},
    #1
  }}}
\begin{tikzpicture}[spy using mag glass=magnification=3, size=1cm]
  \draw [decoration=Koch curve type 2]
    decorate{ decorate{ decorate{ (0,0) -- (2,0) }}};
  \spy [green!50!black] on (1.6,0.1) in node at (2.5,-0.5);
\end{tikzpicture}
```

With the magnifying glass, you can also put it “on top” of the picture itself:



```
\usetikzlibrary {decorations.fractals,shadows,shapes.symbols,spy}
\begin{tikzpicture}
[spy scope=magnification=4, size=1cm,
 every spy in node/.style={
 magnifying glass, circular drop shadow,
 fill=white, draw, ultra thick, cap=round}]

\draw [decoration=Koch curve type 2]
 decorate{ decorate{ decorate{ (0,0) -- (2,0) }}};

\spy on (1.6,0.1) in node;
\end{tikzpicture}
```

74 SVG-Path Library

TikZ Library `svg.path`

```
\usepgflibrary{svg.path} % LATEX and plain TEX and pure pgf
\usepgflibrary[svg.path] % ConTeXt and pure pgf
\usetikzlibrary{svg.path} % LATEX and plain TEX when using TikZ
\usetikzlibrary[svg.path] % ConTeXt when using TikZ
```

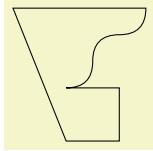
This library defines a command that allows you to specify a path using the SVG-syntax.

`\pgfpathsvg{<path>}`

This command extends the current path by a $\langle path \rangle$ given in the SVG-path-data syntax. This syntax is described in detail in Section 8.3 of the SVG-specification, Version 1.1.

In principle, the complete syntax is supported and the library just provides a parser and a mapping to basic layer commands. For instance, `M 0 10` is mapped to `\pgfpathmoveto{\pgfpoint{0pt}{10pt}}`. There are, however, a few things to be aware of:

- The computation underlying the arc commands `A` and `a` are not numerically stable, which may result in quite imprecise arcs. Bézier curves, both quadratic and cubic, are not affected, neither are arcs spanning degrees that are multiples of 90°.
- The dimensionless units of SVG are always interpreted as points (pt). This is a problem with paths like `M 20000 0`, which will raise an error message since T_EX cannot handle dimensions larger than about 16 000 points.
- All coordinate and canvas transformations apply to the path in the usual fashion.
- The `\pgfpathsvg` command can be freely intermixed with other path commands.



```
\usepgflibrary {svg.path}
\begin{pgfpicture}
  \pgfpathsvg{M 0 0 1 20 0 0 20 -20 0 q 10 0 10 10
              t 10 10 10 10 h -50 z}
  \pgfusepath{stroke}
\end{pgfpicture}
```

75 To Path Library

TikZ Library `topaths`

```
\usetikzlibrary{topaths} % LATEX and plain TEX  
\usetikzlibrary[topaths] % ConTEX
```

This library provides predefined to paths for use with the `to` path operation. After loading this package, you can say for instance `to [loop]` to add a loop to a node.

This library is loaded automatically by TikZ, so you do not need to load it yourself.

75.1 Straight Lines

The following style installs a to path that is simply a straight line from the start coordinate to the target coordinate.

`/tikz/line to` (no value)

Causes a straight line to be added to the path upon a `to` or an `edge` operation.

```
——— \tikz {\draw (0,0) to[line to] (1,0);}
```

75.2 Move-Tos

The following style installs a to path that simply “jumps” to the target coordinate.

`/tikz/move to` (no value)

Causes a move to be added to the path upon a `to` or an `edge` operation.

```
——— —— \tikz \draw (0,0) to[line to] (1,0)  
          to[move to] (2,0) to[line to] (3,0);
```

75.3 Curves

The `curve to` style causes the to path to be set to a curve. The exact way this curve looks can be influenced via a number of options.

`/tikz/curve to` (no value)

Specifies that the to path should be a curve. This curve will leave the start coordinate at a certain angle, which can be specified using the `out` option. It reaches the target coordinate also at a certain angle, which is specified using the `in` option. The control points of the curve are at a certain distance that is computed in different ways, depending on which options are set.

All of the following options implicitly cause the `curve to` style to be installed.

`/tikz/out=<angle>` (no default)

The angle at which the curve leaves the start coordinate. If the start coordinate is a node, the start coordinate is the point on the border of the node at the given `<angle>`. The control point will, thus, lie at a certain distance in the direction `<angle>` from the start coordinate.



```
\begin{tikzpicture}[out=45,in=135]  
  \draw (0,0) to (1,0)  
        (0,0) to (2,0)  
        (0,0) to (3,0);  
\end{tikzpicture}
```

`/tikz/in=<angle>` (no default)

The angle at which the curve reaches the target coordinate.

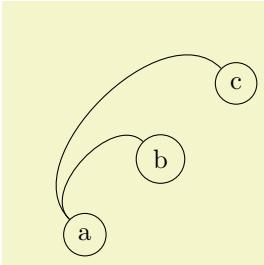
`/tikz/relative=<true or false>` (default `true`)

This option tells TikZ whether the `in` and `out` angles should be considered absolute or relative. Absolute means that an `out` angle of 30° means that the curve leaves the start coordinate at an

angle of 30° relative to the paper (unless, of course, further transformations have been installed). A *relative* angle is, by comparison, measured relative to a straight line from the start coordinate to the target coordinate. Thus, a relative angle of 30° means that the curve will bend to the left from the line going straight from the start to the target. For the target, the relative coordinate is measured in the same manner, namely relative to the line going from the start to the target. Thus, an angle of 150° means that the curve will reach target coming slightly from the left.



```
\begin{tikzpicture}[out=45,in=135,relative]
  \draw (0,0) to (1,0)
    to (2,1)
    to (2,2);
\end{tikzpicture}
```



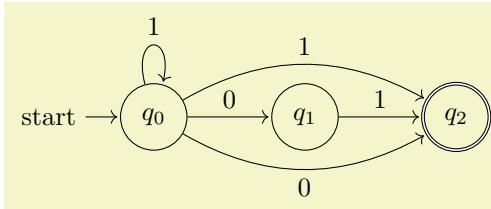
```
\begin{tikzpicture}[out=90,in=90,relative]
  \node [circle,draw] (a) at (0,0) {a};
  \node [circle,draw] (b) at (1,1) {b};
  \node [circle,draw] (c) at (2,2) {c};

  \path (a) edge (b)
    edge (c);
\end{tikzpicture}
```

`/tikz/bend left=<angle>`

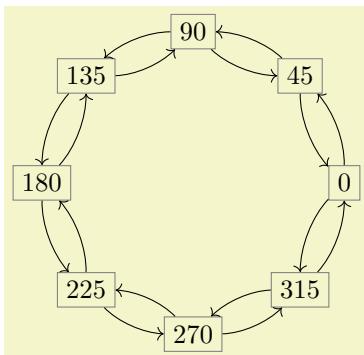
(default last value)

This option sets `out=<angle>,in=180 - <angle>,relative`. If no `<angle>` is given, the last given `bend left` or `bend right` angle is used.



```
\usetikzlibrary {automata,positioning}
\begin{tikzpicture}[shorten >=1pt,node distance=2cm,on grid]
  \node[state,initial] (q_0) {$q_0$};
  \node[state] (q_1) [right=of q_0] {$q_1$};
  \node[state,accepting] (q_2) [right=of q_1] {$q_2$};

  \path[->] (q_0) edge node [above] {0} (q_1)
    edge [loop above] node [above] {1} ();
  \path[->] (q_1) edge node [above] {1} (q_2)
    edge [bend left] node [above] {1} (q_2)
    edge [bend right] node [below] {0} (q_2);
  \path[->] (q_2) edge node [above] {1} (q_2);
\end{tikzpicture}
```



```
\begin{tikzpicture}
\foreach \angle in {0,45,...,315}
  \node[rectangle,draw=black!50] (\angle) at (\angle:2) {\angle};

\foreach \from/\to in {0/45,45/90,90/135,135/180,
                      180/225,225/270,270/315,315/0}
  \path (\from) edge [->,bend right=22,looseness=0.8] (\to)
        edge [<-,bend left=22,looseness=0.8] (\to);
\end{tikzpicture}
```

/tikz/bend right=(*angle*) (default last value)

Works like the `bend left` option, only the bend is to the other side.

/tikz/bend angle=(*angle*) (no default)

Sets the angle to be used by the `bend left` or `bend right`, but without actually selecting the `curve to` or the `relative` option. This is useful for globally specifying a `bend angle` for a whole picture.

/tikz/looseness=(*number*) (no default, initially 1)

This number specifies how “loose” the curve will be. In detail, the following happens: TikZ computes the distance between the start and the target coordinate (if the start and/or target coordinate are nodes, the distance is computed between the points on their border). This distance is then multiplied by a fixed factor and also by the factor `<number>`. The resulting distance, let us call it *d*, is then used as the distance of the control points from the start and target coordinates.

The fixed factor has been chosen in such a way that if `<number>` is 1, if the `in` and `out` angles differ by 90°, then a quarter circle results:



```
\tikz \draw (0,0) to [out=0,in=-90] (1,1);
\tikz \draw (0,0) to [out=0,in=-90,looseness=0.5] (1,1);
```

/tikz/out looseness=(*number*) (no default)

Specifies the looseness factor for the out distance only.

/tikz/in looseness=(*number*) (no default)

Specifies the looseness factor for the in distance only.

/tikz/min distance=(*distance*) (no default)

If the computed distance for the start and target coordinates are below `<distance>`, then `<distance>` is used instead.

/tikz/max distance=(*distance*) (no default)

If the computed distance for the start and target coordinates are above `<distance>`, then `<distance>` is used instead.

/tikz/out min distance=(*distance*) (no default)

The minimum distance set only for the start coordinate.

/tikz/out max distance=(*distance*) (no default)

The maximum distance set only for the start coordinate.

/tikz/in min distance=(*distance*) (no default)

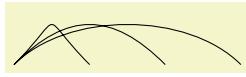
The minimum distance set only for the target coordinate.

/tikz/in max distance=(*distance*) (no default)

The maximum distance set only for the target coordinate.

/tikz/distance=(*distance*) (no default)

Set the minimum and maximum distance to the same value *(distance)*. Note that this causes any computed distance *d* to be ignored and *(distance)* to be used instead.



```
\begin{tikzpicture} [out=45,in=135,distance=1cm]
  \draw (0,0) to (1,0)
    (0,0) to (2,0)
    (0,0) to (3,0);
\end{tikzpicture}
```

/tikz/out distance=(*distance*) (no default)

Sets the minimum and maximum out distance.

/tikz/in distance=(*distance*) (no default)

Sets the minimum and maximum in distance.

/tikz/out control=(*coordinate*) (no default)

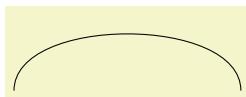
This option causes the *(coordinate)* to be used as the start control point. All computations of *d* are ignored. You can use a coordinate like +(1,0) to specify a point relative to the start coordinate.

/tikz/in control=(*coordinate*) (no default)

This option causes the *(coordinate)* to be used as the target control point.

/tikz/controls=(*coordinate*)and(*coordinate*) (no default)

This option causes the *(coordinate)*s to be used as control points.



```
\tikz \draw (0,0) to [controls=+(90:1) and +(90:1)] (3,0);
```

75.4 Loops

/tikz/loop (no value)

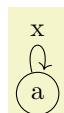
This key is similar to the `curve to` key, but differs in the following ways: First, the actual target coordinate is ignored and the start coordinate is used as the target coordinate. Thus, it is allowed not to provide any target coordinate, which can be useful with unnamed nodes. Second, the `looseness` is set to 8 and the `min distance` to 5mm. These settings result in rather nice loops when the opening angle (difference between `in` and `out`) is 30°.



```
\begin{tikzpicture}
  \node [circle,draw] {a} edge [in=30,out=60,loop] ();
\end{tikzpicture}
```

/tikz/loop above (style, no value)

Sets the `loop` style and sets in and out angles such that loop is above the node. Furthermore, the `above` option is set, which causes a node label to be placed at the correct position.



```
\begin{tikzpicture}
  \node [circle,draw] {a} edge [loop above] node {x} ();
\end{tikzpicture}
```

/tikz/loop below (style, no value)

Works like the previous option.

/tikz/loop left (style, no value)

Works like the previous option.

/tikz/loop right (style, no value)

Works like the previous option.

/tikz/every loop (style, initially `->`, `shorten >=1pt`)

This style is installed at the beginning of every loop.



```
\begin{tikzpicture}[every loop/.style={}]
  \draw (0,0) to [loop above] () to [loop right] ()
        to [loop below] () to [loop left] ();
\end{tikzpicture}
```

76 Through Library

TikZ Library `through`

```
\usetikzlibrary{through} % LATEX and plain TEX  
\usetikzlibrary[through] % ConTeXt
```

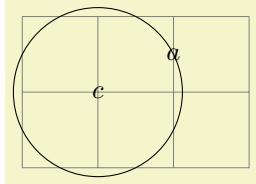
This library defines keys for creating shapes that go through given points.

`/tikz/circle through=(coordinate)`

(no default)

When this key is given as an option to a node, the following happens:

1. The `inner sep` and the `outer sep` are set to zero.
2. The shape is set to `circle`.
3. The `minimum size` is set such that the circle around the center of the node (which is specified using `at`), goes through `(coordinate)`.



```
\usetikzlibrary {through}  
\begin{tikzpicture}  
  \draw[help lines] (0,0) grid (3,2);  
  \node (a) at (2,1.5) {$a$};  
  \node [draw] at (1,1) [circle through={f(a)}] {$c$};  
\end{tikzpicture}
```

77 Tree Library

TikZ Library `trees`

```
\usetikzlibrary{trees} % LATEX and plain TEX
\usetikzlibrary[trees] % ConTEXt
```

This package defines styles to be used when drawing trees.

77.1 Growth Functions

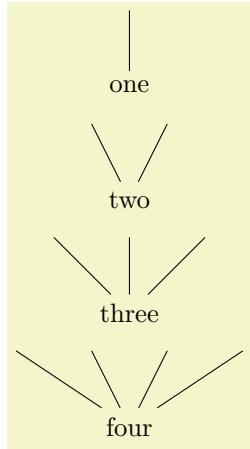
The package `trees` defines two new growth functions. They are installed using the following options:

```
/tikz/grow via three points=one child at (<x>) and two children at (<y>) and (<z>)      (no
default)
```

This option installs a growth function that works as follows: If a parent node has just one child, this child is placed at $\langle x \rangle$. If the parent node has two children, these are placed at $\langle y \rangle$ and $\langle z \rangle$. If the parent node has more than two children, the children are placed at points that are linearly extrapolated from the three points $\langle x \rangle$, $\langle y \rangle$, and $\langle z \rangle$. In detail, the position is $x + \frac{n-1}{2}(y - x) + (c-1)(z - y)$, where n is the number of children and c is the number of the current child (starting with 1).

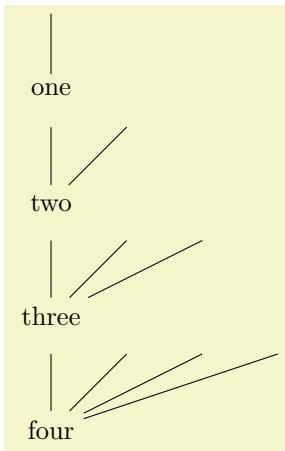
The net effect of all this is that if you have a certain “linear arrangement” in mind and use this option to specify the placement of a single child and of two children, then any number of children will be placed correctly.

Here are some arrangements based on this growth function. We start with a simple “above” arrangement:



```
\usetikzlibrary {trees}
\begin{tikzpicture}[grow via three points=%
  one child at (0,1) and two children at (-.5,1) and (.5,1)]
\node at (0,0) {one} child;
\node at (0,-1.5) {two} child child;
\node at (0,-3) {three} child child child;
\node at (0,-4.5) {four} child child child child;
\end{tikzpicture}
```

The next arrangement places children above, but “grows only to the right”.



```
\usetikzlibrary {trees}
\begin{tikzpicture}[grow via three points=%
  one child at (0,1) and two children at (0,1) and (1,1)]
\node at (0,0) {one} child;
\node at (0,-1.5) {two} child child;
\node at (0,-3) {three} child child child;
\node at (0,-4.5) {four} child child child child;
\end{tikzpicture}
```

In the final arrangement, the children are placed along a line going down and right.