TikZ & PGF

Manual for Version 3.1.10

```
\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\nodeshadowed[#1]#2;{
    \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}}Z};
  \nodeshadowed [at={( 0,8.3)}]
    {\huge \textcolor{green!50!black!50}{\&}};
  \nodeshadowed [at={( 5,8  )},yslant=-0.05]
    {\Huge \textsc{PGF}};
  \nodeshadowed [at={( 0,5  )}]
    {Manual for Version \pgftypesetversion};

  \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

The Ti*k*Z and PGF Packages

Manual for version 3.1.10

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

Till Tantau*
Institut für Theoretische Informatik
Universität zu Lübeck

January 15, 2023

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

I wish to start with the questions "What is Ti*k*Z?" Basically, it just defines a number of TeX 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 Ti*k*Z you "program" your graphics, just as you "program" your document when you use TeX. This also explains the name: Ti*k*Z is a recursive acronym in the tradition of "GNU's Not Unix" and means "Ti*k*Z ist *kein* Zeichenprogramm", which translates to "Ti*k*Z is not a drawing program", cautioning the reader as to what to expect. With Ti*k*Z you get all the advantages of the "TeX-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 Ti*k*Z is, what about "PGF"? As mentioned earlier, Ti*k*Z started out as a project to implement TeX graphics macros that can be used both with pdfLaTeX and also with the classical (PostScript-based) LaTeX. In other words, I wanted to implement a "portable graphics format" for TeX – 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 Ti*k*Z – which has become a whole language of its own.

1.1 The Layers Below Ti*k*Z

It turns out that there are actually *two* layers below Ti*k*Z:

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 `\pgfsys@lineto{10pt}{10pt}` extends the current path to the coordinate $(10\text{pt}, 10\text{pt})$ 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, Ti*k*Z 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 Ti*k*Z frontend all this boils down to a single simple METAFONT-like command:

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

In practice, Ti*k*Z 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 Ti*k*Z, but they are intended more as "technology studies" and less as serious alternatives to Ti*k*Z. In particular, the `pgfpict2e` frontend reimplements the standard LaTeX `{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 Ti*k*Z and almost no one will use the system layer directly, this manual is mainly about Ti*k*Z in the first parts; the basic layer and the system layer are explained at the end.

1.2 Comparison with Other Graphics Packages

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

1. The standard LaTeX `{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 LaTeX'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 Ti*k*Z, `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 Ti*k*Z syntax is more consistent than the `pstricks` syntax as Ti*k*Z 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 `dratex` package is a small graphic package for creating a graphics. Compared to the other package, including Ti*k*Z, it is very small, which may or may not be an advantage.

5. The `metapost` program is a powerful alternative to Ti*k*Z. It used to be an external program, which entailed a bunch of problems, but in LuaTeX 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 Ti*k*Z for users who do not wish to "program" their graphics as is necessary with Ti*k*Z and the other packages above. There is a conversion program that will convert `xfig` graphics to Ti*k*Z.

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.

4. 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 Ti*k*Z 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 Ti*k*Z, 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 Ti*k*Z. 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

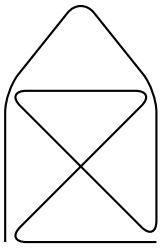
When you need help with PGF and Ti*k*Z, please do the following:

1. Read the manual, at least the part that has to do with your problem.
2. If that does not solve the problem, try having a look at the GitHub development page for PGF and Ti*k*Z (see the title of this document). Perhaps someone has already reported a similar problem and someone has found a solution.
3. 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.
4. 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.
5. *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 Ti*k*Z, 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 Ti*k*Z.



```
\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

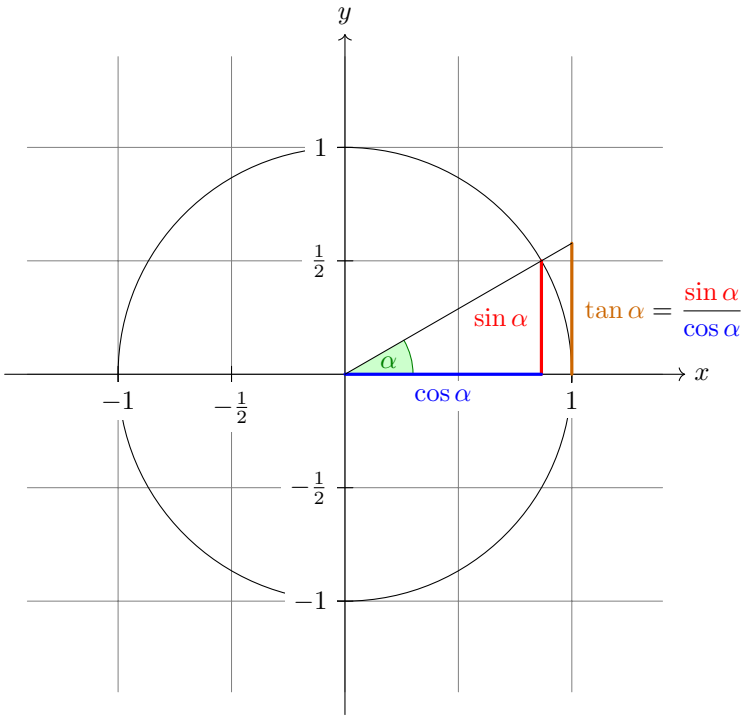
This tutorial is intended for new users of Ti*k*Z. It does not give an exhaustive account of all the features of Ti*k*Z, just of those that you are likely to use right away.

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 Ti*k*Z, which is supposed to stand for "Ti*k*Z ist *kein* Zeichenprogramm". Karl finds this all a bit strange and Ti*k*Z 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 Ti*k*Z's name is intended to warn people that Ti*k*Z 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

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

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

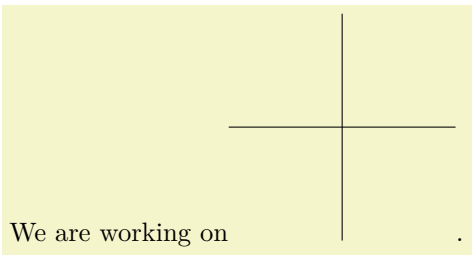
In Ti*k*Z, to draw a picture, at the start of the picture you need to tell TeX or LaTeX that you want to start a picture. In LaTeX this is done using the environment `{tikzpicture}`, in plain TeX you just use `\tikzpicture` to start the picture and `\endtikzpicture` to end it.

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:




```
We are working on
\begin{tikzpicture}
  \draw (-1.5,0) -- (1.5,0);
  \draw (0,-1.5) -- (0,1.5);
\end{tikzpicture}.
```

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) -- (1.5,0)`, which means "a straight line from the point at position $(-1.5, 0)$ to the point at position $(1.5, 0)$". 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 LATEX user, but uses plain TEX since she prefers to do things "the old way". She can also use Ti*k*Z. 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`. Ti*k*Z 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 Ti*k*Z. Instead of `\usepackage{tikz}` he says `\usemodule[tikz]`. Instead of `\begin{tikzpicture}` he writes `\starttikzpicture` and instead of `\end{tikzpicture}` he writes `\stoptikzpicture`.

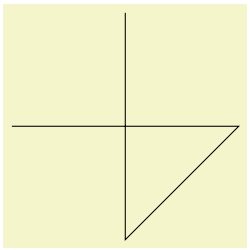
His version of the example looks like this:

```
%% 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 Ti*k*Z 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:



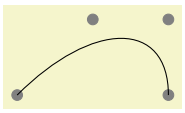
```
\tikz \draw (-1.5,0) -- (1.5,0) -- (0,-1.5) -- (0,1.5);
```

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 Ti*k*Z 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, Ti*k*Z 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.

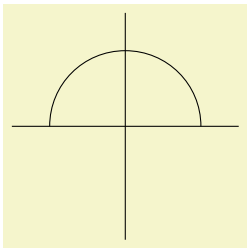
Here is an example (the control points have been added for clarity):



```
\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}
```

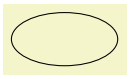
Karl is happy with the result, but finds specifying circles in this way to be extremely awkward. Fortunately, there is a much simpler way.

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];
```

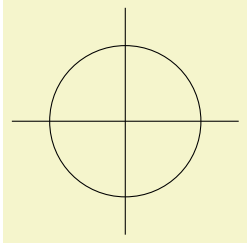
You can also append an ellipse to the path using the `ellipse` operation. Instead of a single radius you can specify two of them:



```
\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 ⬭) 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:

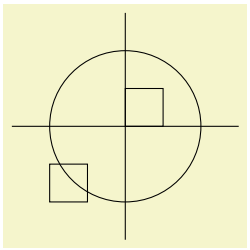


```
\begin{tikzpicture}
  \draw (-1.5,0) -- (1.5,0);
  \draw (0,-1.5) -- (0,1.5);
  \draw (0,0) circle [radius=1cm];
\end{tikzpicture}
```

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 Ti*k*Z 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}
```

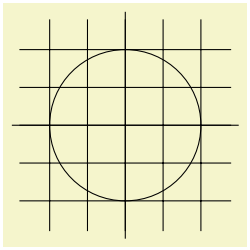
While this may be nice in other situations, this is not really leading anywhere with Karl's problem: First, we would need an awful lot of these rectangles and then there is the border that is not "closed".

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

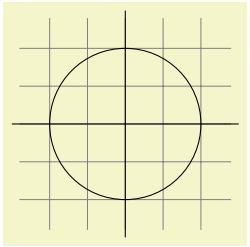
The `grid` path operation adds a grid to the current path. It will add lines making up a grid that fills the rectangle whose one corner is the current point and whose other corner is the point following the `grid` operation. For example, the code `\tikz \draw[step=2pt] (0,0) grid (10pt,10pt);` produces  . 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

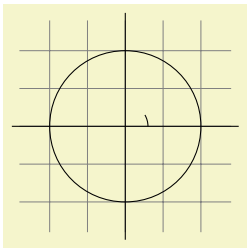
Karl wonders what other options there are that influence how a path is drawn. He saw already that the `color=`⟨*color*⟩ option can be used to set the line's color. The option `draw=`⟨*color*⟩ does nearly the same, only it sets the color for the lines only and a different color can be used for filling (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.

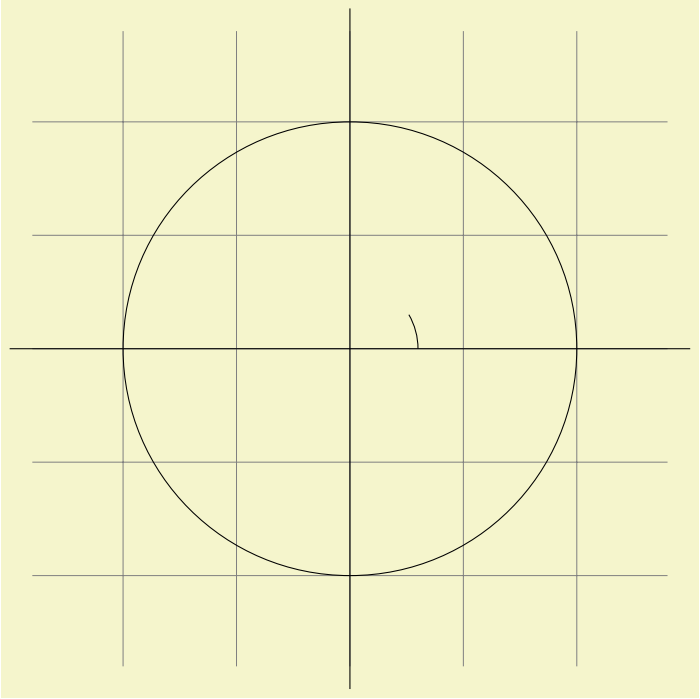
2.10 Arc Path Construction

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".



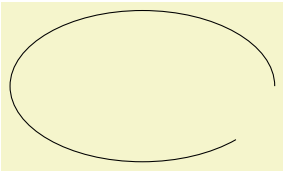
```
\begin{tikzpicture}
  \draw[step=.5cm,gray,very thin] (-1.4,-1.4) grid (1.4,1.4);
  \draw (-1.5,0) -- (1.5,0);
  \draw (0,-1.5) -- (0,1.5);
  \draw (0,0) circle [radius=1cm];
  \draw (3mm,0mm) arc [start angle=0, end angle=30, radius=3mm];
\end{tikzpicture}
```

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}
```

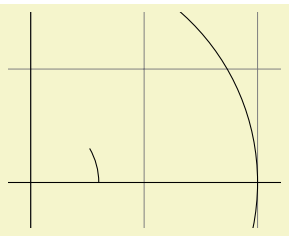
As for circles, you can specify "two" radii in order to get an elliptical arc.



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

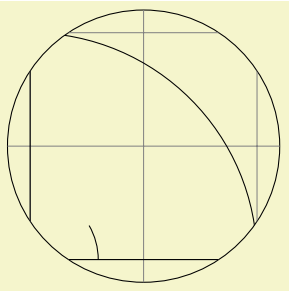
2.11 Clipping a Path

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 Ti*k*Z. 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:

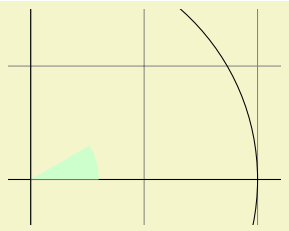
A sine ⌒ curve.

```
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:



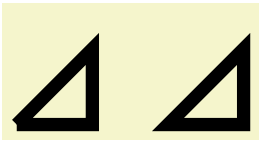
```
\begin{tikzpicture}[scale=3]
  \clip (-0.1,-0.2) rectangle (1.1,0.75);
  \draw[step=.5cm,gray,very thin] (-1.4,-1.4) grid (1.4,1.4);
  \draw (-1.5,0) -- (1.5,0);
  \draw (0,-1.5) -- (0,1.5);
  \draw (0,0) circle [radius=1cm];
  \fill[green!20!white] (0,0) -- (3mm,0mm)
    arc [start angle=0, end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

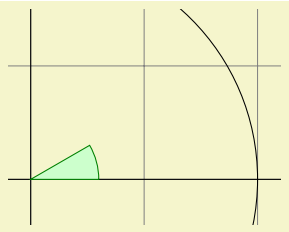
```
\fill[green!20!white] (0,0) -- (3mm,0mm)
  arc [start angle=0, end angle=30, radius=3mm] -- cycle;
```

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

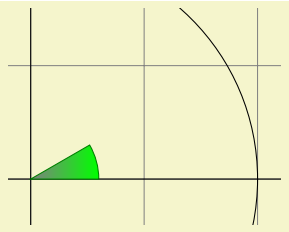
Karl briefly considers the possibility of making the angle "more fancy" by *shading* it. Instead of filling the area with a uniform color, a smooth transition between different colors is used. For this, `\shade` and `\shadedraw`, for shading and drawing at the same time, can be used:

```
\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.

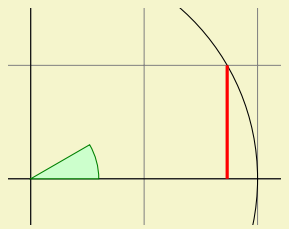
2.15 Specifying Coordinates

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 $(\cos 30°, \sin 30°)$ on the circle".

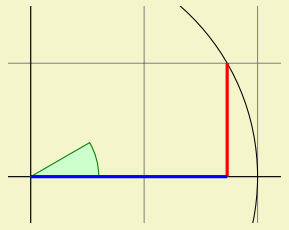
You can add a single `+` sign in front of a coordinate or two of them as in `+(0cm,1cm)` or `++(2cm,0cm)`. Such coordinates are interpreted differently: The first form means "1cm upwards from the previous specified position" and the second means "2cm to the right of the previous specified position, making 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);
\end{tikzpicture}
```

Karl used the fact $\sin 30° = 1/2$. 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 `(30:1cm)` 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 `(`$\langle p \rangle$ `|-` $\langle q \rangle$`)` 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:

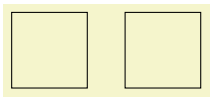


```
\begin{tikzpicture}[scale=3]
  \clip (-0.1,-0.2) rectangle (1.1,0.75);
  \draw[step=.5cm,gray,very thin] (-1.4,-1.4) grid (1.4,1.4);
  \draw (-1.5,0) -- (1.5,0);
  \draw (0,-1.5) -- (0,1.5);
  \draw (0,0) circle [radius=1cm];
  \filldraw[fill=green!20,draw=green!50!black] (0,0) -- (3mm,0mm)
      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 my pen to $(\cos 30°, 1/2)$. Next, there comes another coordinate

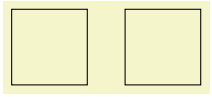
specification, so I move my pen there without drawing anything. This new point is half a unit down from the last position, thus it is at $(\cos 30°, 0)$. Finally, I move the pen to the origin, but this time drawing something (because of the `--`)."

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



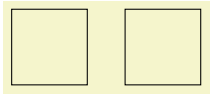
```
\begin{tikzpicture}
  \def\rectanglepath{-- ++(1cm,0cm)  -- ++(0cm,1cm)  -- ++(-1cm,0cm) -- cycle}
  \draw (0,0) \rectanglepath;
  \draw (1.5,0) \rectanglepath;
\end{tikzpicture}
```

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 Ti*k*Z's math engine knows how to compute things like `tan(30)`. Note the added braces since, otherwise, Ti*k*Z'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.

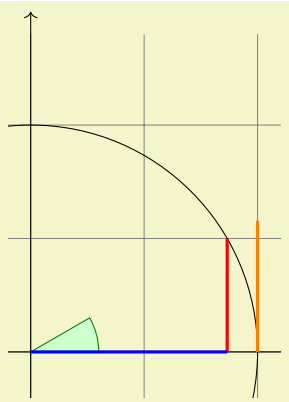
What Karl must do is to create two "invisible" paths that intersect at the position of interest. Creating paths that are not otherwise seen can be done using the `\path` command without any options like `draw` or `fill`. Then, Karl can add the `name path` option to the path for later reference. Once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

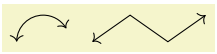
It turns out that adding arrow tips is pretty easy: Karl adds the option `->` to the drawing commands for the axes:



```
\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: →. The shape seems vaguely familiar and, indeed, this is exactly the end of TEX's standard arrow used in something like $f\colon A \to B$.

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 105. 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

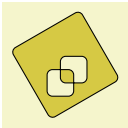
When you specify a coordinate like `(1cm,1cm)`, where is that coordinate placed on the page? To determine the position, Ti*k*Z, TEX, and PDF or PostScript all apply certain transformations to the given coordinate in order to determine the final position on the page.

Ti*k*Z provides numerous options that allow you to transform coordinates in Ti*k*Z'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 Ti*k*Z 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 Ti*k*Z, so if you are familiar with them, feel free to use them. Ti*k*Z 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 Ti*k*Z, but Ti*k*Z includes it automatically.

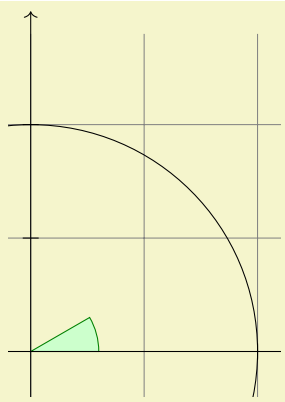
In its basic form, the `\foreach` command is easy to use:

$x = 1$, $x = 2$, $x = 3$,

```
\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:

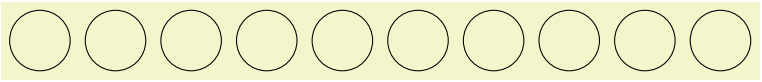


```
\begin{tikzpicture}[scale=3]
  \clip (-0.1,-0.2) rectangle (1.1,1.51);
  \draw[step=.5cm,gray,very thin] (-1.4,-1.4) grid (1.4,1.4);
  \filldraw[fill=green!20,draw=green!50!black] (0,0) -- (3mm,0mm)
      arc [start angle=0, end angle=30, radius=3mm] -- cycle;
  \draw[->] (-1.5,0) -- (1.5,0);
  \draw[->] (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
1,4	2,4	3,4	4,4	5,4
1,3	2,3	3,3	4,3	5,3
1,2	2,2	3,2	4,2	5,2
1,1	2,1	3,1	4,1	5,1

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

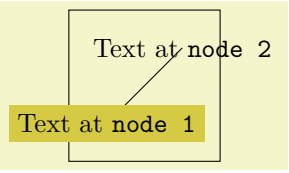
```
\begin{tikzpicture}
  \foreach \x in {1,2,...,5,7,8,...,12}
    \foreach \y in {1,...,5}
    {
      \draw (\x,\y) +(-.5,-.5) rectangle ++(.5,.5);
      \draw (\x,\y) node{\x,\y};
    }
\end{tikzpicture}
```

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

2.21 Adding Text

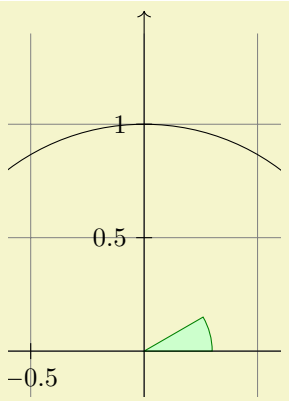
Karl is, by now, quite satisfied with the picture. However, the most important parts, namely the labels, are still missing!

Ti*k*Z 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 Ti*k*Z 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, Ti*k*Z 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:



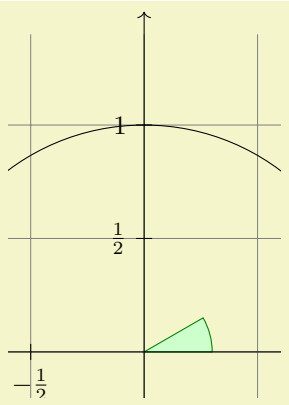

```
\begin{tikzpicture}[scale=3]
  \clip (-0.6,-0.2) rectangle (0.6,1.51);
  \draw[step=.5cm,help lines] (-1.4,-1.4) grid (1.4,1.4);
  \filldraw[fill=green!20,draw=green!50!black] (0,0) -- (3mm,0mm)
    arc [start angle=0, end angle=30, radius=3mm] -- cycle;
  \draw[->] (-1.5,0) -- (1.5,0);   \draw[->]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

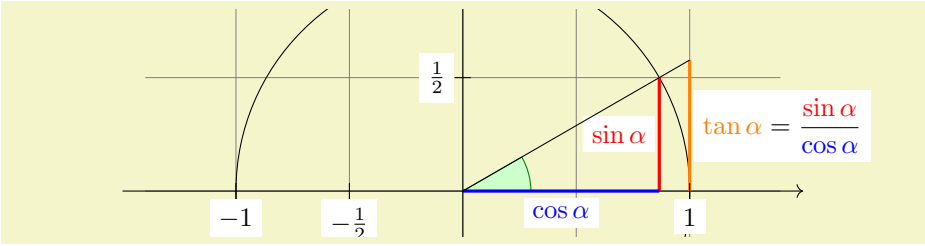
Karl now faces a problem: For the `\foreach` statement, the position `\x` should still be given as `0.5` since Ti*k*Z will not know where `\frac{1}{2}` is supposed to be. On the other hand, the typeset text should really be `\frac{1}{2}`. To solve this problem, `\foreach` offers a special syntax: Instead 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 ⟨*first*⟩`/`⟨*second*⟩. In each iteration, `\x` will be set to ⟨*first*⟩ and `\xtext` will be set to ⟨*second*⟩. If no ⟨*second*⟩ is given, the ⟨*first*⟩ 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}
```

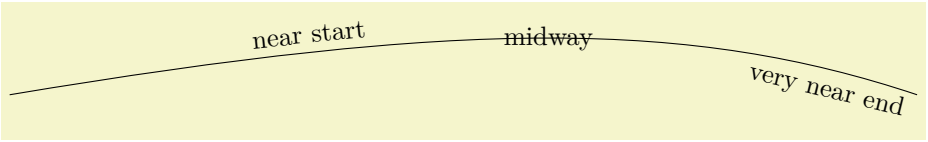
Karl is quite pleased with the result, but his son points out that this is still not perfectly satisfactory: The grid and the circle interfere with the numbers and decrease their legibility. Karl is not very concerned by this (his students do not even notice), but his son insists that there is an easy 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)
    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]
    {$\displaystyle \tan \alpha \color{black}=
      \frac{{\color{red}\sin \alpha}}{\color{blue}\cos \alpha}$} (t);

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

  \draw[xshift=1.85cm]
    node[right,text width=6cm,information text]
    {
      The {\color{anglecolor} angle $\alpha$} is $30^\circ$ in the
      example ($\pi/6$ in radians). The {\color{sincolor}sine of
        $\alpha$}, which is the height of the red line, is
      \[
      {\color{sincolor} \sin \alpha} = 1/2.
      \]
      By the Theorem of Pythagoras ...
    };
\end{tikzpicture}
```

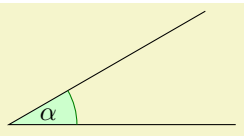
2.22 Pics: The Angle Revisited

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, Ti*k*Z 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.

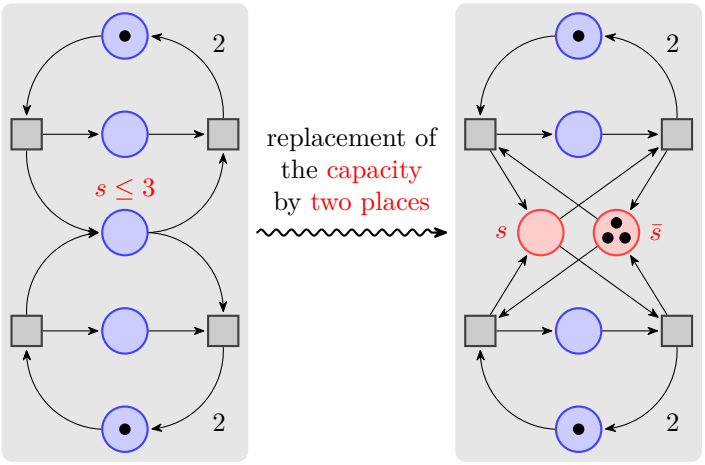
3 Tutorial: A Petri-Net for Hagen

In this second tutorial we explore the node mechanism of Ti*k*Z 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 Ti*k*Z 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:



3.2 Setting up the Environment

For the picture Hagen will need to load the Ti*k*Z 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.meta` 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 LaTeX

When using LaTeX use:

```
\documentclass{article} % say

\usepackage{tikz}
\usetikzlibrary{arrows.meta,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 TeX

When using plain TeX use:

```
%% Plain TeX file
\input tikz.tex
\usetikzlibrary{arrows.meta,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.meta,decorations.pathmorphing,backgrounds,positioning,fit,petri]

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

3.3 Introduction to Nodes

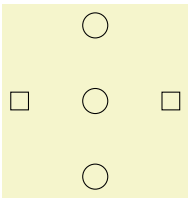
In principle, we already know how to create the graphics that Hagen desires (except perhaps for the snaked line, we will come to that): We start with big light gray rectangle and then add lots of circles and small rectangle, plus some arrows.

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, Ti*k*Z 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`.



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

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

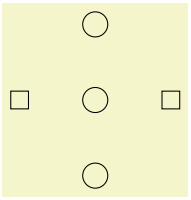
empty {}). So, why do we see anything at all? The answer is the draw option for the node operation: It causes the "shape around the text" to be drawn.

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

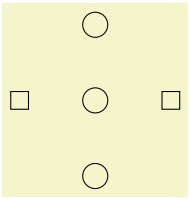
Hagen now understands how the node operation adds nodes to the path, but it seems a bit silly to create a path using the \path operation, consisting of numerous superfluous move-to operations, only to place nodes. He is pleased to learn that there are ways to add nodes in a more sensible manner.

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:

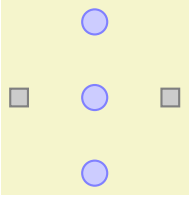


```
\begin{tikzpicture}
  \node at ( 0,2) [circle,draw] {};
  \node at ( 0,1) [circle,draw] {};
  \node at ( 0,0) [circle,draw] {};
  \node at ( 1,1) [rectangle,draw] {};
  \node at (-1,1) [rectangle,draw] {};
\end{tikzpicture}
```

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

3.5 Using Styles

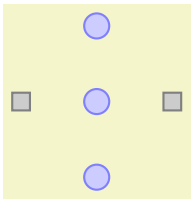
Feeling adventurous, Hagen tries to make the nodes look nicer. In the final picture, the circles and rectangle should be filled with different colors, resulting in the following code:



```
\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.

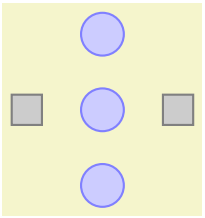
To solve this problem, Hagen uses styles. He defines a style for places and another style for transitions:



```
\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

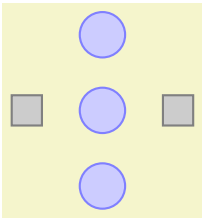
Before Hagen starts naming and connecting the nodes, let us first make sure that the nodes get their final appearance. They are still too small. Indeed, Hagen wonders why they have any size at all, after all, the text is empty. The reason is that Ti*k*Z automatically adds some space around the text. 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.



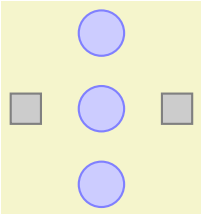
```
\begin{tikzpicture}
  [place/.style={circle,draw=blue!50,fill=blue!20,thick,
                 inner sep=0pt,minimum size=6mm},
   transition/.style={rectangle,draw=black!50,fill=black!20,thick,
                      inner sep=0pt,minimum size=4mm}]
  \node at ( 0,2) [place] {};
  \node at ( 0,1)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

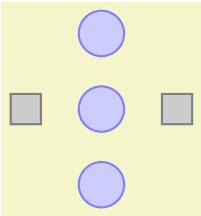
There are two ways to name a node. The first is to use the `name=` option. The second method is to write the desired name in parentheses after the `node` operation. Hagen thinks that this second method seems strange, but he will soon change his opinion.



```
% ... 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.

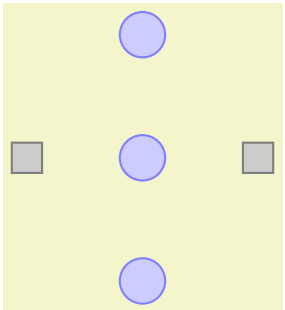
The syntax for the `node` operation is quite liberal with respect to the order in which node names, the `at` specifier, and the options must come. Indeed, you can even have multiple option blocks between the `node` and the text in curly braces, they accumulate. You can rearrange them arbitrarily and 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)                              {};
  \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}
```

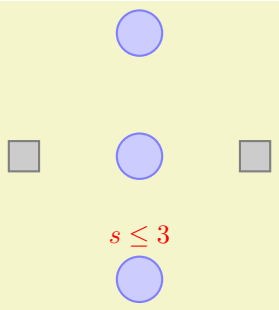
With the `positioning` library loaded, when an option like `below` is followed by `of`, then the position of the node is shifted in such a manner that it is placed at the distance `node distance` in the specified direction of the given direction. The `node distance` is either the distance between the 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:

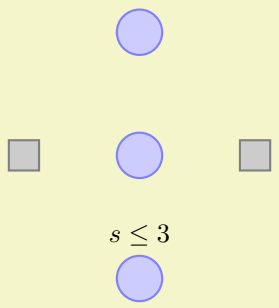
1. Hagen can just add a new node above the `north` anchor of the `semaphore` node.




```
\usetikzlibrary {positioning}
\begin{tikzpicture}
  \node[place]      (waiting)                          {};
  \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]  {};

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

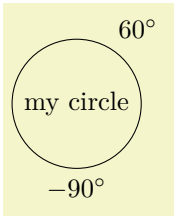
This is a general approach that will "always work".

2. 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:$s\le 3$`. This option is interpreted as follows: We want a node above the `semaphore` node and this node should read "$s \le 3$". Instead of `above` we could also use things like `below left` before the colon or a number like `60`.



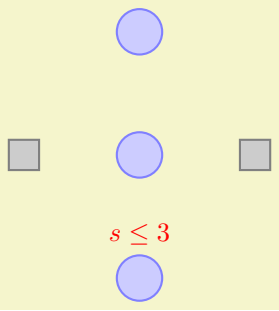

```
\usetikzlibrary {positioning}
\begin{tikzpicture}
  \node[place]      (waiting)                          {};
  \node[place]      (critical)       [below=of waiting]  {};
  \node[place]      (semaphore)      [below=of critical,
                                      label=above:$s\le3$] {};
  \node[transition] (leave critical) [right=of critical] {};
  \node[transition] (enter critical) [left=of 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:$60^\circ$,label=below:$-90^\circ$] {my circle};
```

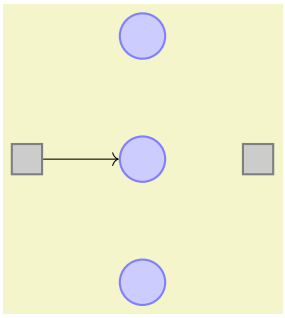
Hagen is not fully satisfied with the `label` option since the label is not red. To achieve this, he has two options: First, he can redefine the `every label` style. Second, he can add options to the label's node. These options are given following the `label=`, so he would write `label=[red]above:$s\le3$`. 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:$s\le3$}`. 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)                          {};
  \node[place]      (critical)       [below=of waiting]  {};
  \node[place]      (semaphore)      [below=of critical,
                                      label=above:$s\le3$] {};
  \node[transition] (leave critical) [right=of critical] {};
  \node[transition] (enter critical) [left=of critical]  {};
\end{tikzpicture}
```

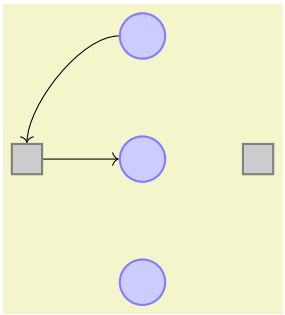
3.10 Connecting Nodes

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:



```
\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.east) -- (critical.west);
\end{tikzpicture}
```

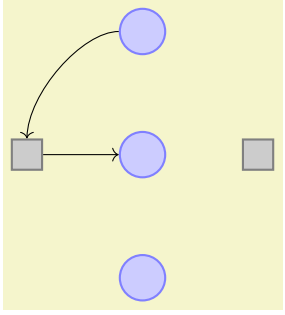
Next, let us tackle the curve from `waiting` to `enter critical`. This can be specified using curves and controls:



```
\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.east) -- (critical.west);
  \draw [->] (waiting.west) .. controls +(left:5mm) and +(up:5mm)
                            .. (enter critical.north);
\end{tikzpicture}
```

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

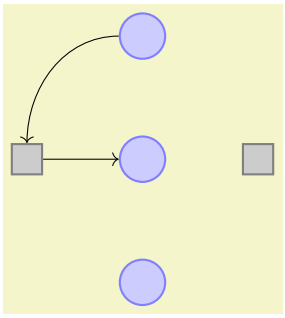
So, let us start improving the code for the edges. First, Hagen can leave out the anchors:



```
\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) -- (critical);
  \draw [->] (waiting) .. controls +(left:8mm) and +(up:8mm)
                       .. (enter critical);
\end{tikzpicture}
```

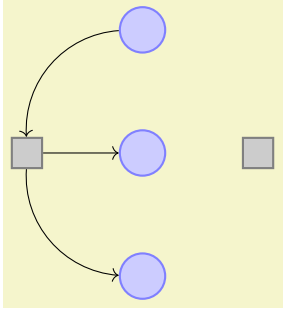
Hagen is a bit surprised that this works. After all, how did Ti*k*Z know that the line from `enter critical` to `critical` should actually start on the borders? Whenever Ti*k*Z encounters a whole node name as a "coordinate", it tries to "be smart" about the anchor that it should choose for this node. Depending on what happens next, Ti*k*Z 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:



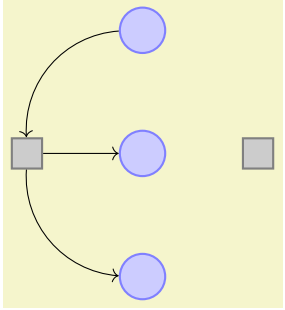
```
\usetikzlibrary {positioning}
\begin{tikzpicture}
  \node[place]      (waiting)                              {};
  \node[place]      (critical)       [below=of waiting]    {};
  \node[place]      (semaphore)      [below=of critical]   {};
  \node[transition] (leave critical) [right=of critical] 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

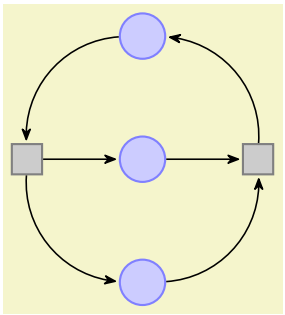
It is now time for Hagen to learn about yet another way of specifying edges: Using the `edge` path operation. This operation is very similar to the `to` operation, but there is one important difference: Like a node the edge generated by the `edge` operation is not part of the main path, but is added 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 [->]                (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.

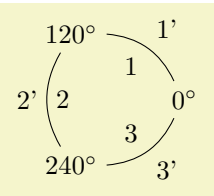
The finishing touch is to introduce two styles `pre` and `post` and to use the `bend angle=45` option to set the bend angle once and for all:



```
\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

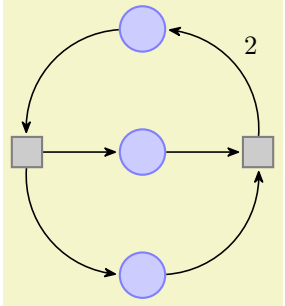
The next thing that Hagen needs to add is the "2" at the arcs. For this Hagen can use Ti*k*Z's automatic node placement: By adding the option `auto`, Ti*k*Z will position nodes on curves and lines in such a way that they are not on the curve but next to it. Adding `swap` will mirror the label with respect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




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

  \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.

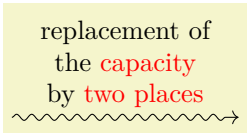
For this he can use a *decoration*. To draw the snaked line, Hagen only needs to set the two options `decoration=snake` and `decorate` on the path. This causes all lines of the path to be replaced by snakes. It is also possible to use snakes only in certain parts of a path, but Hagen will not need this.

```
\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.

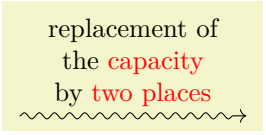

```
\usetikzlibrary {decorations.pathmorphing}
\begin{tikzpicture}
  \draw [->,decorate,
     decoration={snake,amplitude=.4mm,segment length=2mm,post length=1mm}]
    (0,0) -- (3,0);
\end{tikzpicture}
```

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 `\\` command to enforce the line breaks at the desired positions.




```
\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:



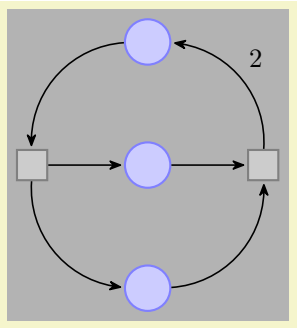

```
\usetikzlibrary {decorations.pathmorphing}
\begin{tikzpicture}
  \draw [->,decorate,
     decoration={snake,amplitude=.4mm,segment length=2mm,post length=1mm}]
    (0,0) -- (3,0)
    node [above,text width=3cm,align=center,midway]
    {
      replacement of the \textcolor{red}{capacity} by
     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

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

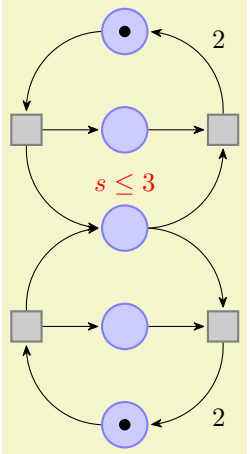
3.14 The Complete Code

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.

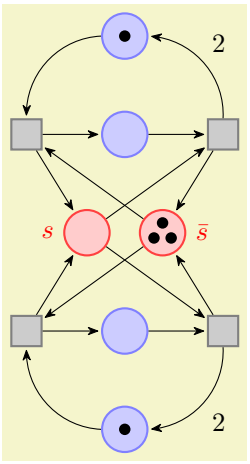
First, Hagen needs less style definitions, but he still needs to specify the colors of places and transitions.

```
\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:$s\le 3$] {};
   \node [place]          (c2) [below=of s]                     {};
   \node [place,tokens=1] (w2) [below=of c2]                    {};

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




```
\usetikzlibrary {arrows.meta,petri,positioning}
  \begin{scope}[xshift=6cm]
    \node [place,tokens=1]      (w1')                                  {};
    \node [place]               (c1') [below=of w1']                   {};
    \node [red place]           (s1') [below=of c1',xshift=-5mm]
            [label=left:$s$]                                           {};
    \node [red place,tokens=3] (s2') [below=of c1',xshift=5mm]
            [label=right:$\bar s$]                                     {};
    \node [place]               (c2') [below=of s1',xshift=5mm]  {};
    \node [place,tokens=1]      (w2') [below=of c2']                   {};

    \node [transition] (e1') [left=of c1'] {}
      edge [pre,bend left]                  (w1')
      edge [post]                           (s1')
      edge [pre]                            (s2')
      edge [post]                           (c1');
    \node [transition] (e2') [left=of c2'] {}
      edge [pre,bend right]                 (w2')
      edge [post]                           (s1')
      edge [pre]                            (s2')
      edge [post]                           (c2');
    \node [transition] (l1') [right=of c1'] {}
      edge [pre]                            (c1')
      edge [pre]                            (s1')
      edge [post]                           (s2')
      edge [post,bend right] node[swap] {2} (w1');
    \node [transition] (l2') [right=of c2'] {}
      edge [pre]                            (c2')
      edge [pre]                            (s1')
      edge [post]                           (s2')
      edge [post,bend left]  node {2}       (w2');
  \end{scope}
```

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

```
  \begin{scope}[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,->,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*

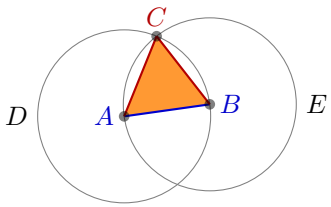
In this third tutorial we have a look at how Ti*k*Z can be used to draw geometric constructions.

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 now, 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 CDB, 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 Ti*k*Z code.

4.1.1 Setting up the Environment

As in the previous tutorials, Euclid needs to load Ti*k*Z, 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]
```

[1] The 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.

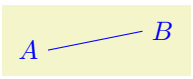
4.1.2 The Line AB

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 Ti*k*Z that does exactly what he needs: It produces a number between -1 and 1. Since Ti*k*Z 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

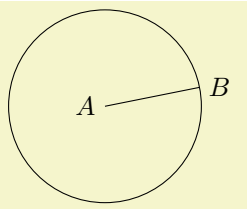
The first tricky construction is the circle around A. We will see later how to do this in a very simple manner, but first let us do it the "hard" way.

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}
```

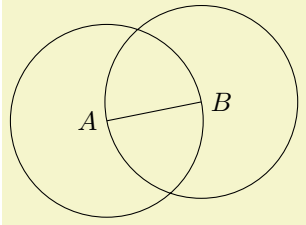
Each assignment in a let operation starts with `\p`, usually followed by a ⟨*digit*⟩. Then comes an equal sign and a coordinate. The coordinate is evaluated and the result is stored internally. From then on you can use the following expressions:

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.

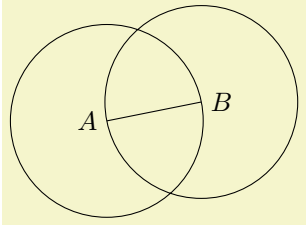
Note that `\p1` is not a coordinate in the usual sense. Rather, it just expands to a string like `10pt,20pt`. So, you cannot write, for instance, `(\p1.center)` since this would just expand to `(10pt,20pt.center)`, which makes no sense.

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:

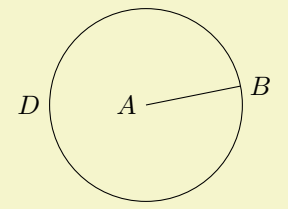



```
\usetikzlibrary {calc}
\begin{tikzpicture}
  \coordinate [label=left:$A$]  (A) at (0,0);
  \coordinate [label=right:$B$] (B) at (1.25,0.25);
  \draw (A) -- (B);

  \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.

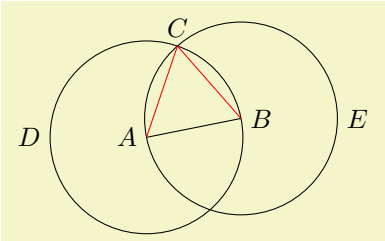



```
\usetikzlibrary {through}
\begin{tikzpicture}
  \coordinate [label=left:$A$]  (A) at (0,0);
  \coordinate [label=right:$B$] (B) at (1.25,0.25);
  \draw (A) -- (B);

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

4.1.4 The Intersection of the Circles

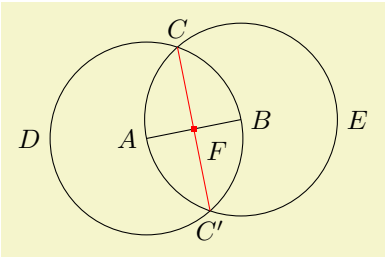
Euclid can now draw the line and the circles. The final problem is to compute the intersection of the two circles. This computation is a bit involved if you want to do it "by hand". Fortunately, the `intersections` library allows us to compute the intersection of arbitrary paths.

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}
```

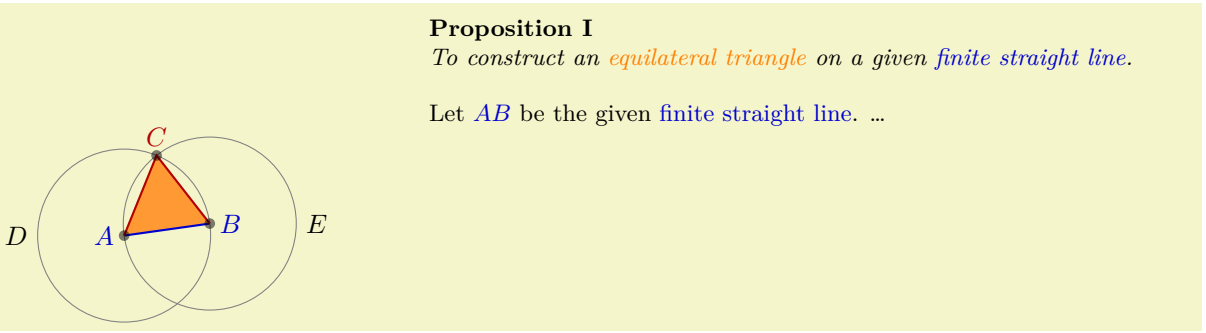
It turns out that this can be further shortened: The `name intersections` takes an optional argument `by`, which lets you specify names for the coordinates and options for them. This creates more compact code. Although Euclid does not need it for the current picture, it is just a small step to computing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

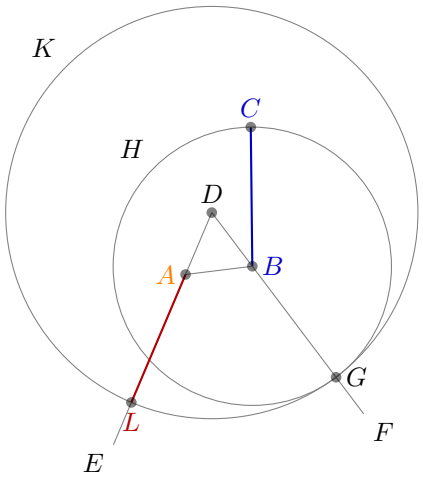
Back to Euclid's code. He introduces a few macros to make life simpler, like a `\A` macro for typesetting a blue A. He also uses the `background` layer for drawing the triangle behind everything at the end.




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

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

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

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

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

  \path [name intersections={of=D and E,by={[label=above:\C]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
    \emph{To construct an \textcolor{triangle}{equilateral triangle}
      on a given \textcolor{input}{finite straight line}.}
    \par\vskip1em
    Let \A\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.

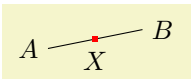
Let A be the given point, and BC the given straight line. It is required to place a straight line equal to the given straight line BC with one end at the point A.
Join the straight line AB from the point A to the point B, and construct the equilateral triangle DAB on it.
Produce the straight lines AE 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.

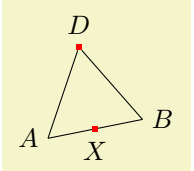
For this reason, TikZ supports some simplifications. First, there is a simple syntax for computing a point that is "partway" on a line from p to q: You place these two points in a coordinate calculation – remember, they start with `($` and end with `$)` – and then combine them using `!`⟨*part*⟩`!`. A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



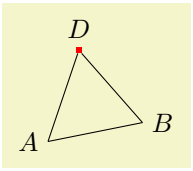

```
\usetikzlibrary {calc}
\begin{tikzpicture}
  \coordinate [label=left:$A$]  (A) at (0,0);
  \coordinate [label=right:$B$] (B) at (1.25,0.25);
  \draw (A) -- (B);
  \node [fill=red,inner sep=1pt,label=below:$X$] (X) at ($ (A)!.5!(B) $) {};
\end{tikzpicture}
```

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 $\sin(60°) \cdot 2$. 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}
```

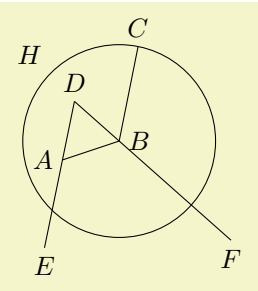
Finally, it is not necessary to explicitly name the point X. Rather, again like in the `xcolor` package, it is possible to chain partway modifiers:




```
\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

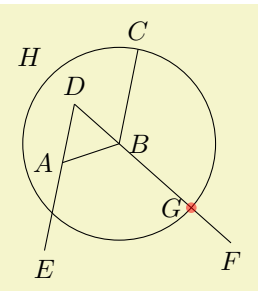
The next step in the construction is to draw a circle around B through C, which is easy enough to do using the `circle through` option. Extending the lines DA and DB can be done using partway calculations, but this time with a part value outside the range $[0, 1]$:




```
\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}
```

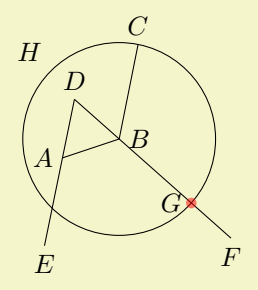
We now face the problem of finding the point G, which is the intersection of the line BF and the circle H. One way is to use yet another variant of the partway computation: Normally, a partway computation has the form $\langle p\rangle\,!\,\langle factor\rangle\,!\,\langle q\rangle$, resulting in the 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:



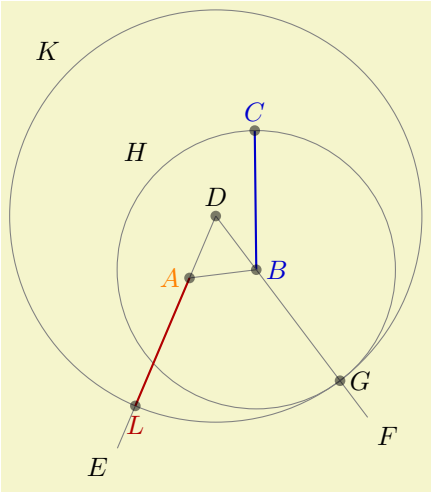

```
\usetikzlibrary {calc,through}
  \node (H) [label=135:$H$,draw,circle through=(C)] at (B) {};
  \path let \p1 = ($ (B) - (C) $) in
    coordinate [label=left:$G$] (G) at ($ (B) ! veclen(\x1,\y1) ! (F) $);
  \fill[red,opacity=.5] (G) circle (2pt);
```

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

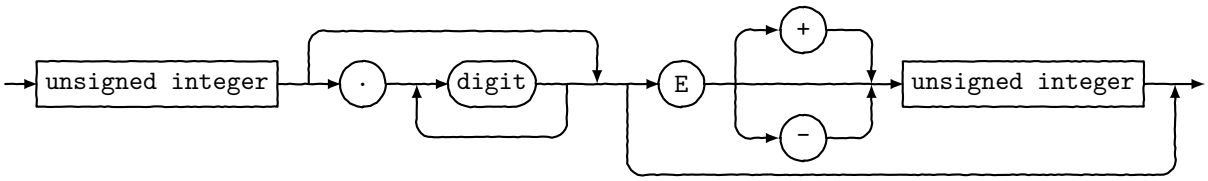



```
\usetikzlibrary {calc,intersections,through}
\begin{tikzpicture}[thick,help lines/.style={thin,draw=black!50}]
  \def\A{\textcolor{orange}{$A$}}   \def\B{\textcolor{input}{$B$}}
  \def\C{\textcolor{input}{$C$}}    \def\D{$D$}
  \def\E{$E$}                       \def\F{$F$}
  \def\G{$G$}         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

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

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

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

  \coordinate [label=above:\D] (D) at ($ (A)!.5!(B) ! {sin(60)*2} ! 90:(B) $);

  \draw [help lines] (D) -- ($ (D)!3.75!(A) $) coordinate [label=-135:\E] (E);
  \draw [help lines] (D) -- ($ (D)!3.75!(B) $) coordinate [label=-45:\F] (F);

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

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

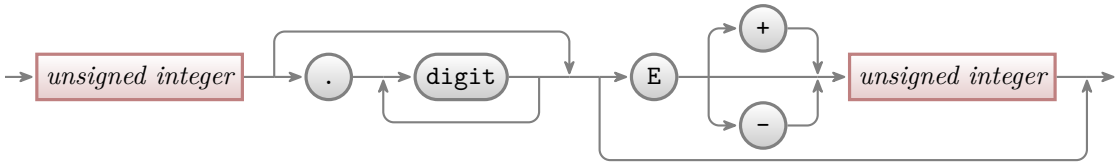
5 Tutorial: Diagrams as Simple Graphs

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 this[2]:



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:

unsigned integer

```
\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

[2]The shown diagram was not scanned, but rather typeset using Ti*k*Z. The jittering lines were created using the `random steps` 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.misc` 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.misc}
\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.misc}
\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.misc}
\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

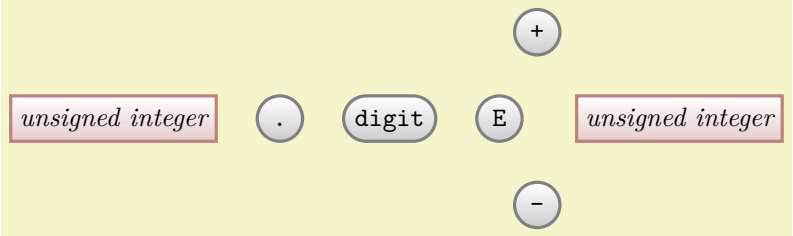
Ilka now has the "styling" of the nodes ready. The next problem is to place them in the right places. There are several ways to do this. The most straightforward is to simply explicitly place the nodes at certain coordinates "calculated by hand". For very simple graphics this is perfectly alright, but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

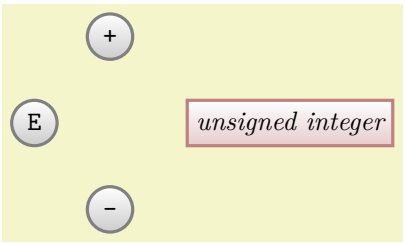
Ilka can use this to draw the place the nodes in a long row:




```
\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}
```

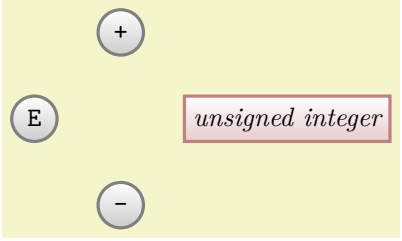
For the plus and minus nodes, Ilka is a bit startled by their placements. Shouldn't they be more to the right? The reason they are placed in that manner is the following: The `north east` anchor of the `E` node lies at the "upper start of the right arc", which, a bit unfortunately in this case, happens 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".

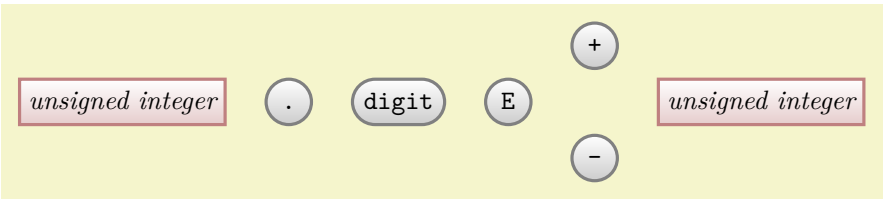

```
\usetikzlibrary {calc,positioning,shapes.misc}
\begin{tikzpicture}[node distance=5mm and 5mm,
    skip loop/.style={to path={-- ++(0,#1) -| (\tikztotarget)}}]
  \node (dot)   [terminal]                        {.};
  \node (digit) [terminal,right=of dot]           {digit};
  \node (E)     [terminal,right=of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

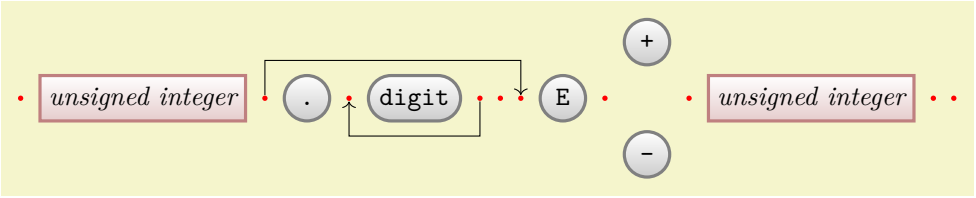
In Ilka's graphic, there will be three rows: One row containing only the plus node, one row containing the main nodes and one row containing only the minus node.




```
\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 [nonterminal] {unsigned integer}; \\
    % 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 (p8) [point]  {}; &    \node (ui2)   [nonterminal] {unsigned integer}; &
    \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}
```

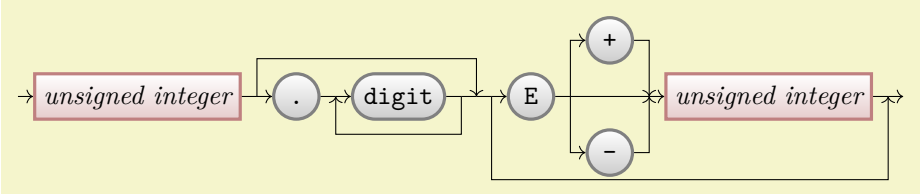
Now, it's only a small step to add all the missing edges.

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):

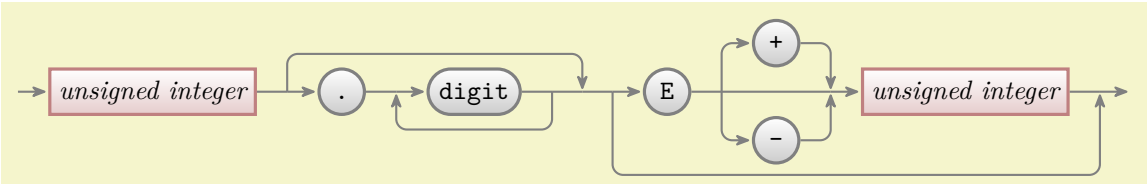


```
\usetikzlibrary {graphs,shapes.misc}
\begin{tikzpicture}[skip loop/.style={to path={-- ++(0,#1) -| (\tikztotarget)}},
                    hv path/.style={to path={-| (\tikztotarget)}},
                    vh path/.style={to path={|- (\tikztotarget)}}]
  \matrix[row sep=1mm,column sep=2mm] { \matrixcontent };

  \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.

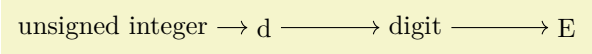
It turns out that edges from a node to a whole group of nodes are quite easy to specify, as shown in the next example. Additionally, by using the `use existing nodes` option, Ilka can also leave out all the parentheses (again, some options have been moved outside to keep the examples shorter):




```
\usetikzlibrary {arrows.meta,graphs,shapes.misc}
\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

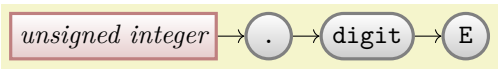
Ilka has heard that the `graph` command is also supposed to make it easy to create nodes, not only to connect them. This is, indeed, correct: When the `use existing nodes` option is not used and when a node name is not surrounded by parentheses, then Ti*k*Z will actually create a node whose name and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




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

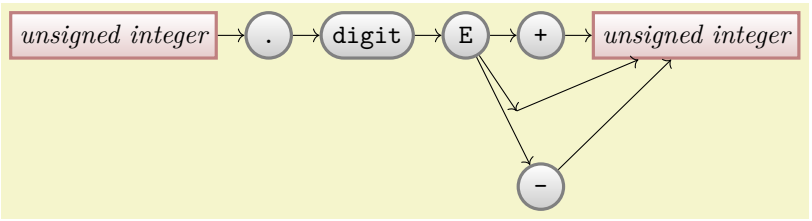
Not quite perfect, but we are getting somewhere. First, let us change the positioning algorithm by saying `grow right sep`, which causes new nodes to be placed to the right of the previous nodes with a certain fixed separation (`1em` by default). Second, we add some options to make the node "look nice".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

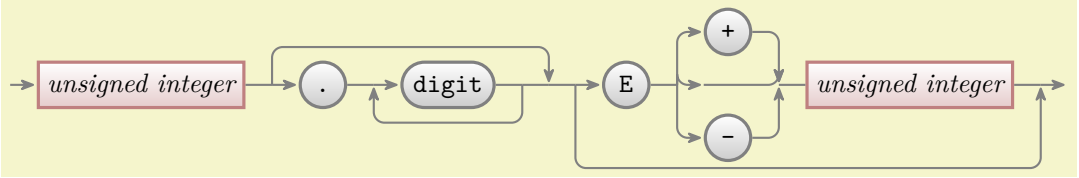



```
\usetikzlibrary {graphs,shapes.misc}
\tikz \graph [grow right sep] {
  unsigned integer  [nonterminal] ->
  "."               [terminal] ->
  digit             [terminal] ->
  E                 [terminal] ->
  {
    "+"             [terminal],
    ""              [coordinate],
    "-"             [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 Ti*k*Z 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.

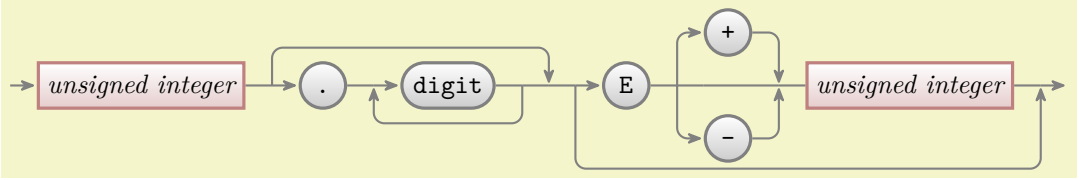
Next, Ilka needs to add some coordinates in between of some nodes where the back-loops should got and she needs to shift the nodes a bit:



```
\usetikzlibrary {arrows.meta,graphs,shapes.misc}
\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 Ti*k*Z'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.misc}
\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 Ti*k*Z.

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.

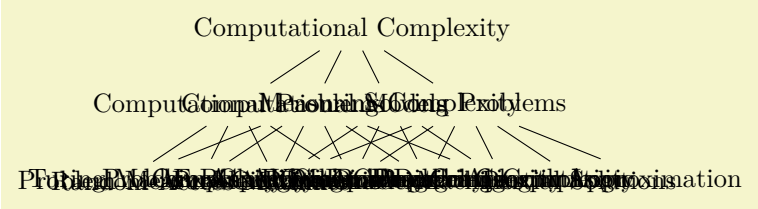
Johannes starts with a tree-like list of concepts that he feels are important in Computational Complexity:

- 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 Ti*k*Z-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} }
    };
```

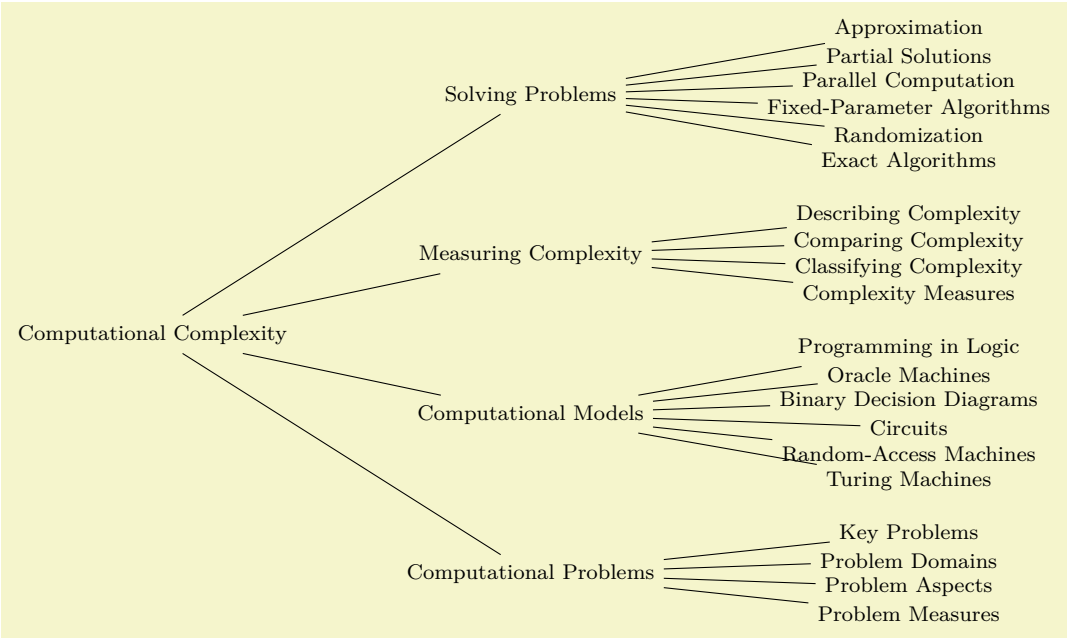
Well, that did not quite work out as expected (although, what, exactly, did one expect?). There are two problems:

1. The overlap of the nodes is due to the fact that Ti*k*Z is not particularly smart when it comes to placing child nodes. Even though it is possible to configure Ti*k*Z to use rather clever placement methods, Ti*k*Z 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:



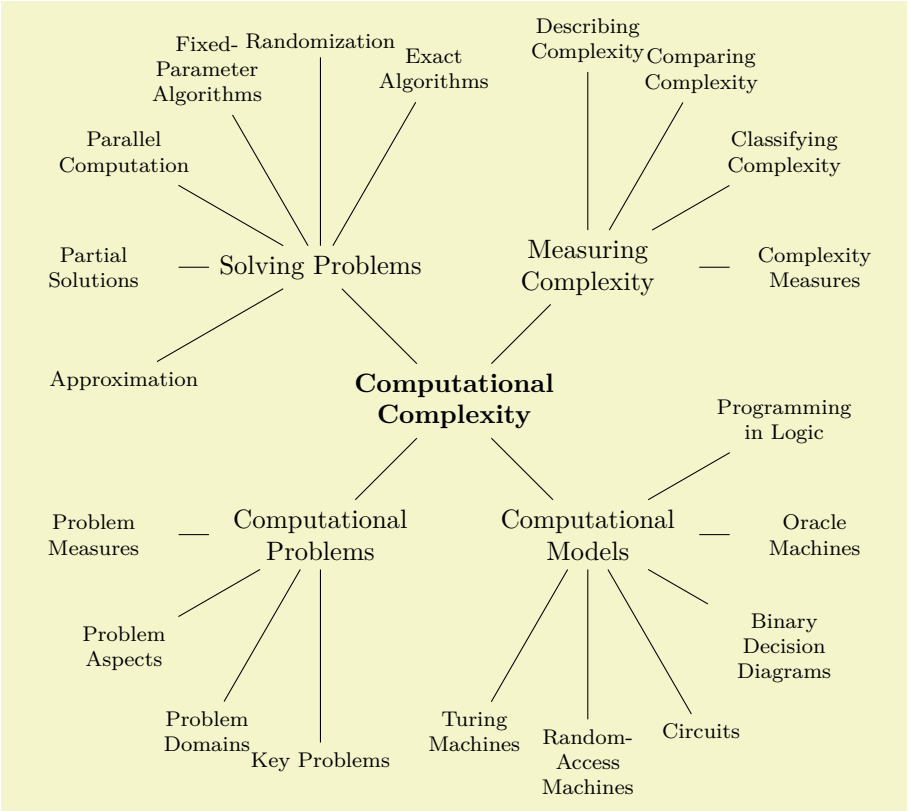

```
\usetikzlibrary {trees}
\tikz [font=\footnotesize,
       grow=right, level 1/.style={sibling distance=6em},
                   level 2/.style={sibling distance=1em}, level distance=5cm]
  \node {Computational Complexity} % root
    child { node {Computational Problems}
      child { node {Problem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

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.

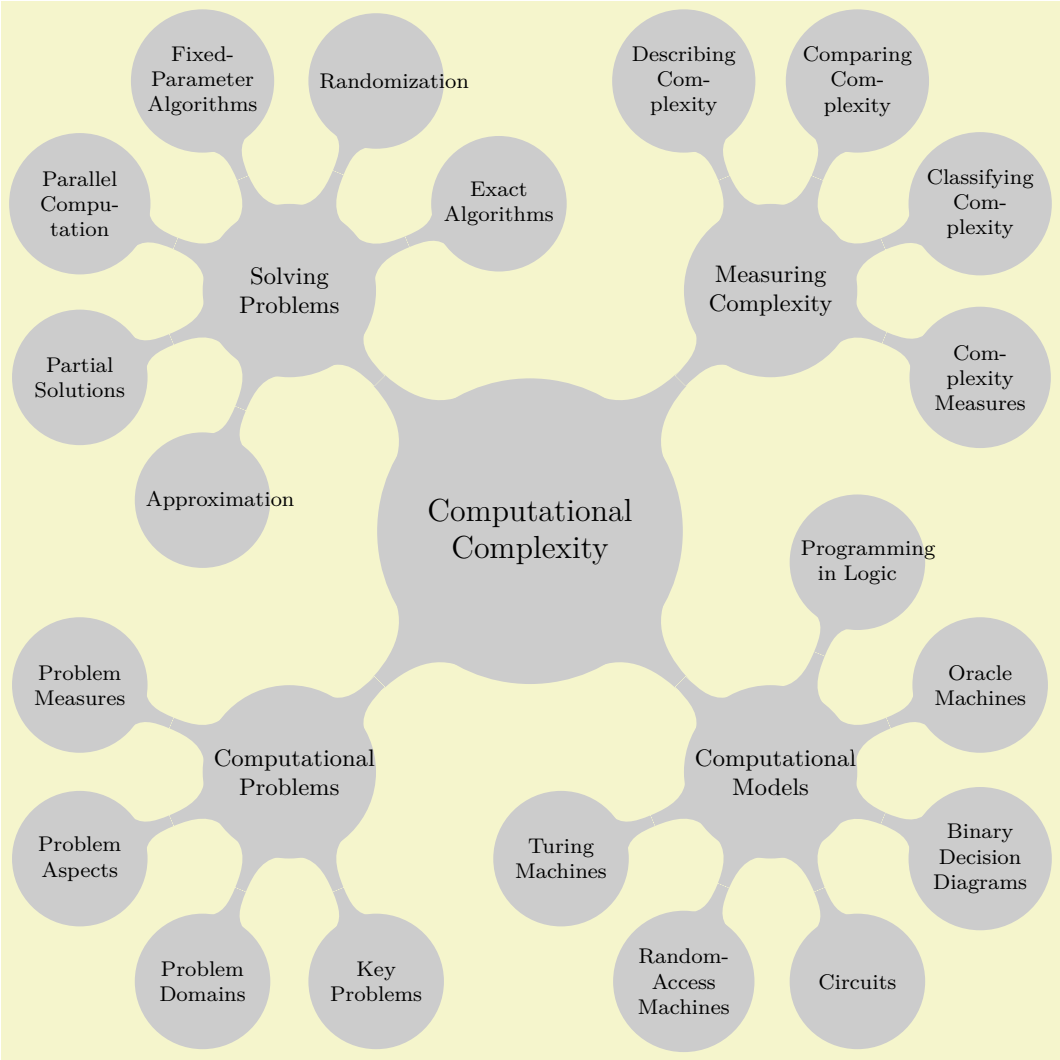
6.3 Creating the Lecture Map

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.

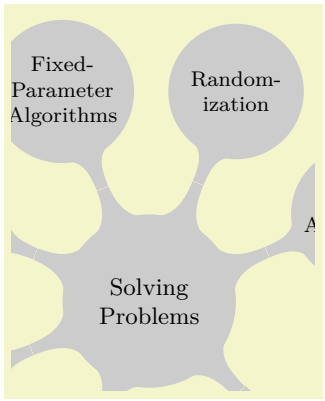
The second step is to add the `concept` option to every node that will, indeed, be a concept of the mindmap. The idea is that some nodes of a tree will be real concepts, while other nodes might just be "simple children". Typically, this is not the case, so you might consider saying `every node/.style=concept`.

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 `\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 `\hskip0pt` inside each node, he uses the `execute at begin node` option to make Ti*k*Z insert this text with every node.



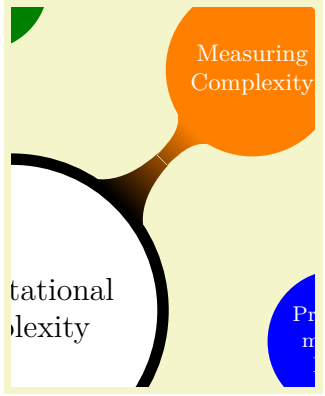

```
\usetikzlibrary {mindmap}
\begin{tikzpicture}
  [mindmap,
   every node/.style={concept, execute at begin node=\hskip0pt},
   concept color=black!20,
   grow cyclic,
   level 1/.append style={level distance=4.5cm,sibling angle=90},
   level 2/.append style={level distance=3cm,sibling angle=45}]
  \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.

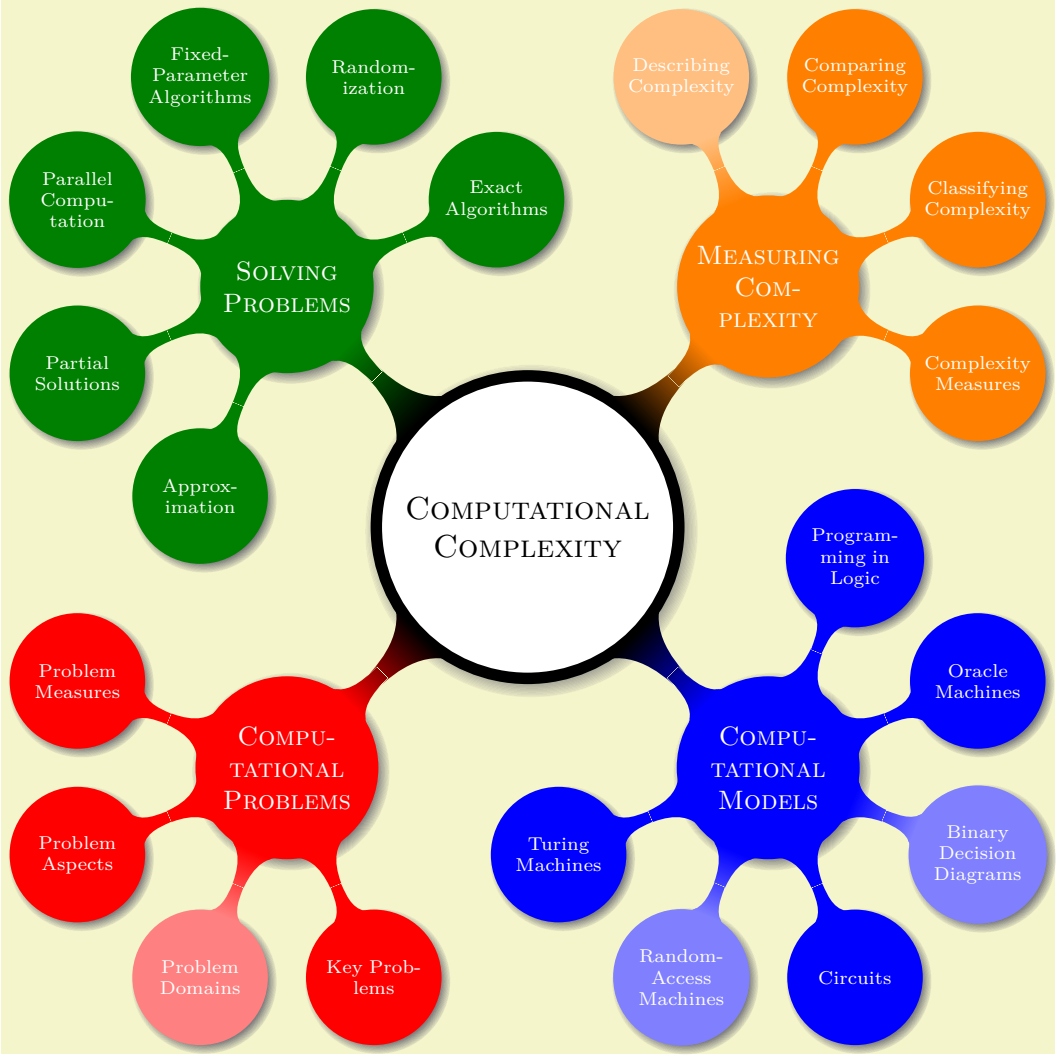
For the root concept Johannes decides to do something special: He sets the concept color to black, sets the line width to a large value, and sets the fill color to white. The effect of this is that the root concept will be encircled with a thick black line and the children are connected to the central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




```
\usetikzlibrary {mindmap}
\begin{tikzpicture}
  [mindmap,
   every node/.style={concept, execute at begin node=\hskip0pt},
   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} }
      ... % 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=\hskip0pt},
    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}
```

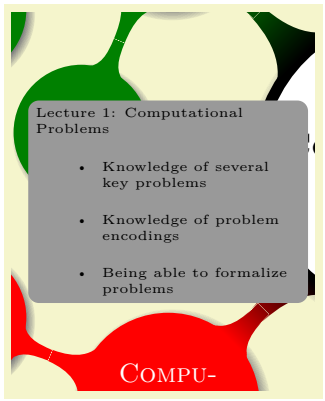
6.4 Adding the Lecture Annotations

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 Ti*k*Z'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} }
  }
...
```

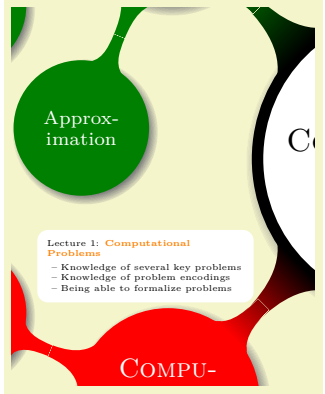
The `annotation` style of the `mindmap` library mainly sets up a rectangular shape of appropriate size. Johannes configures the style by defining `every annotation` appropriately.




```
\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{--}{\topsep=2pt\itemsep=0pt\parsep=0pt
              \parskip=0pt\labelwidth=8pt\leftmargin=8pt
              \itemindent=0pt\labelsep=2pt}
    #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}
\end{tikzpicture}
```

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 777 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:

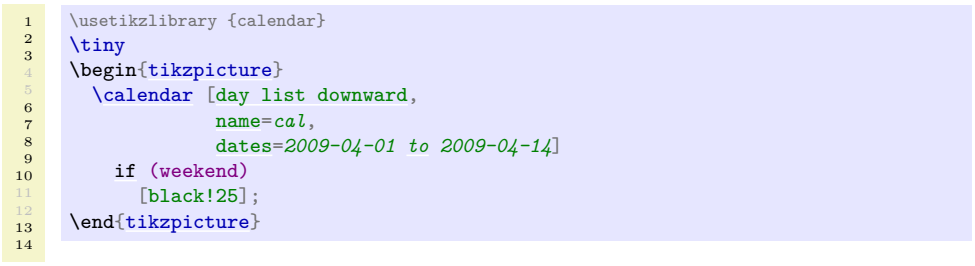

```
\usetikzlibrary {backgrounds,mindmap,shadows}
\begin{tikzpicture}[
  mindmap,
  concept color=black,
  root concept/.append style={
    concept,
    circular drop shadow,
    fill=white, line width=1ex,
    text=black, font=\large\scshape}
  ]

  \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".



```
\usetikzlibrary {calendar}
\tiny
\begin{tikzpicture}
  \calendar [day list downward,
             name=cal,
             dates=2009-04-01 to 2009-04-14]
    if (weekend)
      [black!25];
\end{tikzpicture}
```

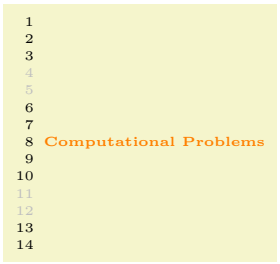
Using the `name` option, we gave a name to the calendar, which will allow us to reference the nodes that make up the individual days of the calendar later on. For instance, the rectangular node containing the `1` that represents April 1st, 2009, can be referenced as `(cal-2009-04-01)`. The `dates` option 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:

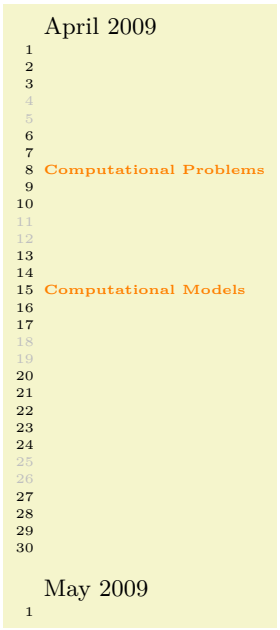
```
\def\lecture#1#2#3#4#5#6{
  % As before:
  \node [annotation, #3, scale=0.65, text width=4cm, inner sep=2mm] at (#4) {
    Lecture #1: \textcolor{orange}{\textbf{#2}}
    \list{--}{\topsep=2pt\itemsep=0pt\parsep=0pt
              \parskip=0pt\labelwidth=8pt\leftmargin=8pt
              \itemindent=0pt\labelsep=2pt}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




```
\usetikzlibrary {calendar}
\tiny
\begin{tikzpicture}
  \calendar [day list downward,
             name=cal,
             dates=2009-04-01 to 2009-04-14]
    if (weekend)
      [black!25];

  % 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 47 for details) and then typesets the month text at a special position at the beginning of each month.




```
\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=\hskip0pt},
    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).

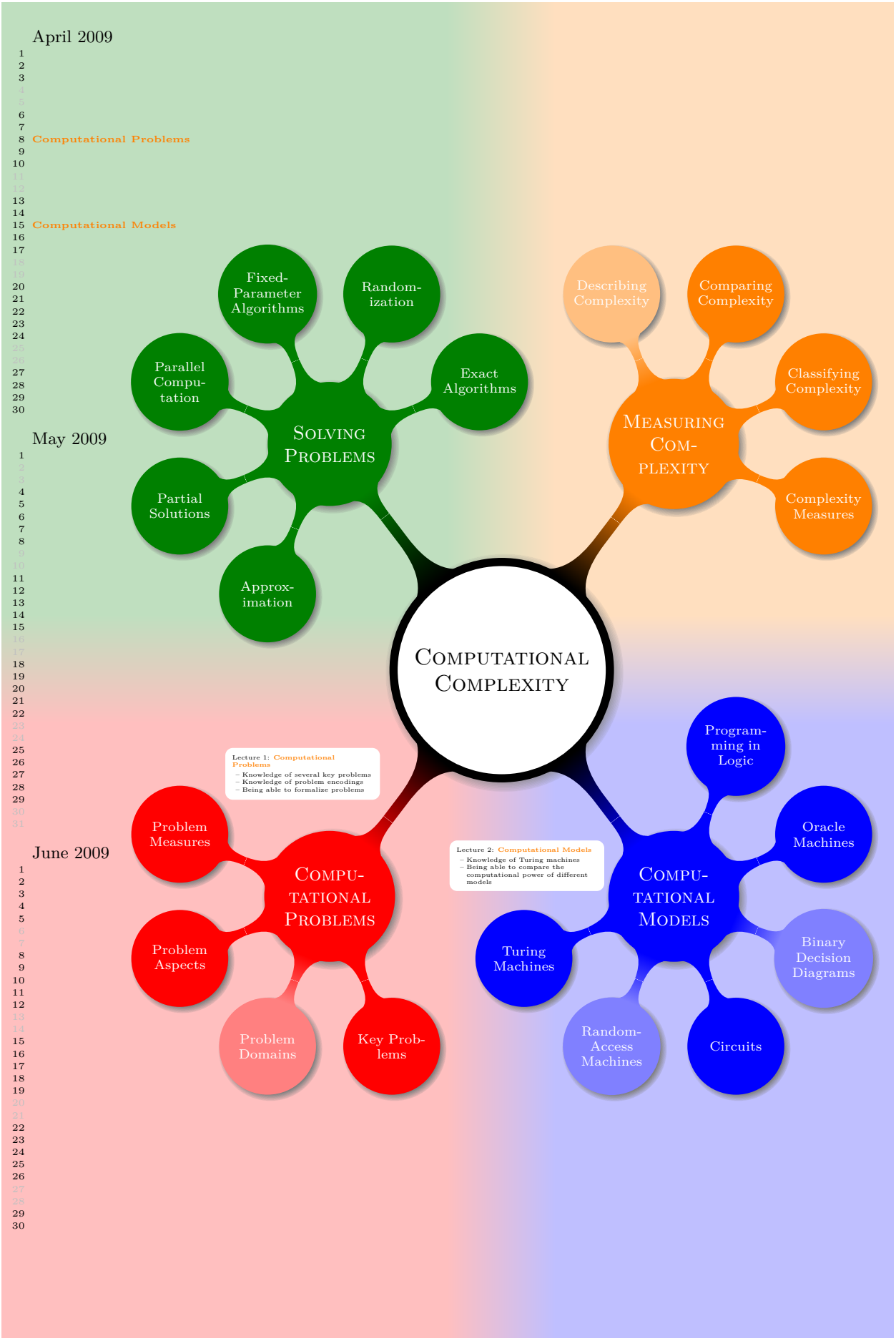
April 2009

Computational Problems

Computational Models

May 2009

June 2009



7 Guidelines on Graphics

The present section is not about PGF or Ti*k*Z, 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 spend 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. LaTeX and TeX 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 GNUPLOT, 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.

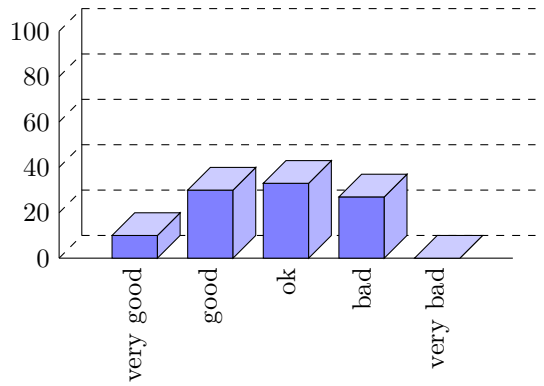
The first question you should ask yourself when creating a plot is: Are there enough data points to merit a plot? If the answer is "not really", use a table.

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:

Rating given	*Participants (out of 50) who gave this rating*	*Percentage*
"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 where 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"?

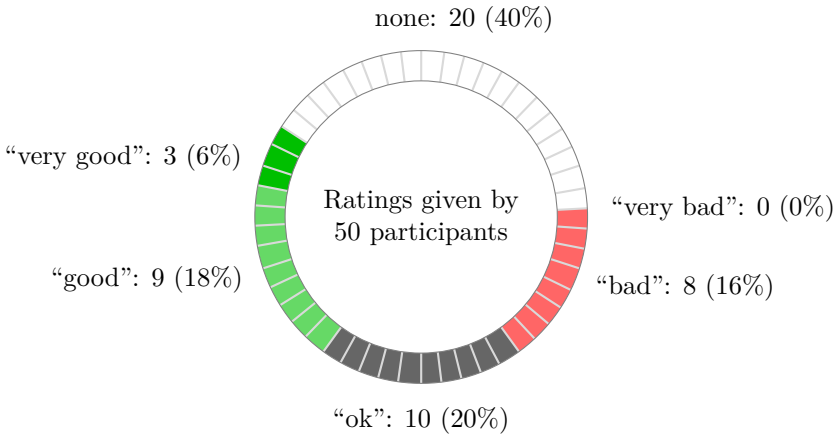
Sadly, the graphic does not allow us to answer *a single one of these questions*. The table answers all of them directly, except for the last one. In essence, the information density of the graphic is very close to zero. The table has a much higher information density; despite the fact that it uses quite 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.

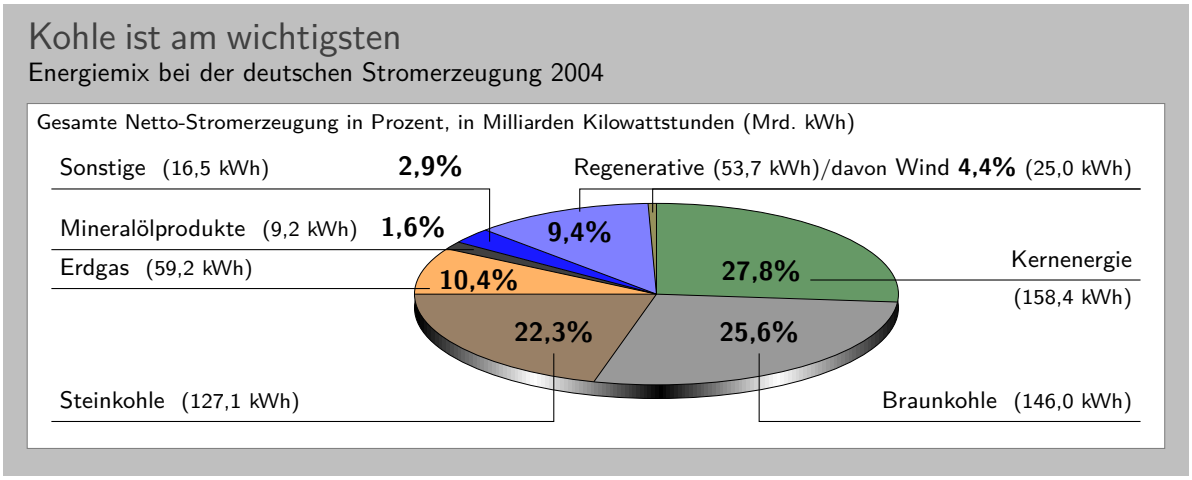
- The three dimensional setup makes it much harder to gauge the height of the bars correctly. Consider the "bad" bar. Is the number this bar stands for more than 20 or less? While the front of the bar is below the 20 line, the back of the bar (which counts) is above.
- It is impossible to tell which 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 Ti*k*Z, 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 readers'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.

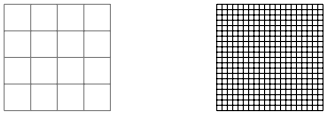
Readers will not read a magazine like a novel. Instead of reading a magazine line-by-line, we use headlines and short abstracts to check whether we want to read a certain article or not. The job of typography is to

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.

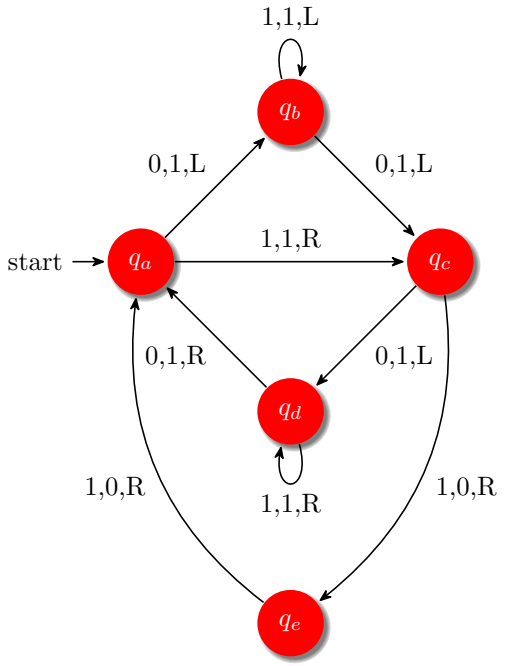
- Background images and shadings distract and only seldomly add anything of importance to a graphic.

- 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)
        (B) edge [loop above] node {1,1,L} (B)
            edge              node {0,1,L} (C)
        (C) edge              node {0,1,L} (D)
            edge [bend left]  node {1,0,R} (E)
        (D) edge [loop below] node {1,1,R} (D)
            edge              node {0,1,R} (A)
        (E) 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 3.1.10 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 pdfTEX.
- `context` with complete functionality, except for graphics inclusion, which works only for pdfTEX.

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 were to find them. If you have a problem with installing, you might wish to have a look at the Debian page or the MiKTEX page first.

8.2.1 Debian

The command "`apt-get install texlive-pictures`" 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-3.1.10`, 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 LaTeX 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 LaTeX 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 LaTeX Project Public License, Version 1.3c 2006-05-20

9.3.1 Preamble

The LaTeX Project Public License (LPPL) is the primary license under which the LaTeX kernel and the base LaTeX 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 TeX-related (such as a LaTeX package), but it is written in such a way that you can use it even if your work is unrelated to TeX.

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 LaTeX3 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 LaTeX 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 'LaTeX-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 LaTeX 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 LaTeX distribution explains the motivation behind the conditions of this license. It explains, for example, why distributing LaTeX under the GNU General Public

License (GPL) was considered inappropriate. Even if your work is unrelated to LaTeX, 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 'LaTeX-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: LATEX, Plain TEX, ConTEXt

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` TEX format, Leslie Lamport's popular LATEX format, and Hans Hangen's ConTEXt format.

10.1.1 Using the LATEX Format

Using PGF and Ti*k*Z with the LATEX 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 LATEX 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 2019 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 TEX Format

When using the plain TEX 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 LATEX format, PGF is not as good at discerning the appropriate configuration for the plain TEX 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 LATEX style files, the plain TEX files like `tikz.tex` also just include the correct `tikz.code.tex` file.

10.1.3 Using the ConTEXt Format

When using the ConTEXt format, you say `\usemodule[pgf]` or `\usemodule[tikz]`. As for the plain TEX 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 ConTEXt support is very similar to the plain TEX 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 `pgfmodulematrix`. 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 LATEX-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 LATEX 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 pdfTEX, 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 (la)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 LaTeX mode it uses graphicx for the graphics inclusion and does not support masking.
2. In plain TeX mode it does not support image inclusion.

File pgfsys-xetex.def

This is a driver file for use with xe(la)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 LaTeX mode it uses graphicx for the graphics inclusion and does not support masking.
2. In plain TeX mode it does not support image inclusion.
3. Shadings are approximated with discrete colors. This typically leads to aliasing patterns in PostScript and PDF viewing applications.
4. Opacity, Transparency Groups, Fadings and Blend Modes are 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 Acrobat Distiller.

10.2.3 Producing PostScript Output

File pgfsys-dvips.def

This is a driver file for use with (la)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 LaTeX mode it uses graphicx for the graphics inclusion. Image masking is supported if the PostScript output is further processed with ps2pdf to produce PDF.
2. In plain TeX mode it does not support image inclusion.
3. Functional shadings are approximated with Type-0 functions (sampled functions), because Type-4 functions are not available in the latest (version 3) PostScript language definition. Due to their fixed resolution, Type-0 functional shadings are of lesser quality at higher zoom levels as compared to functional shadings from PDF producing drivers. Axial and radial shadings are fully supported. The same output quality (smooth shadings) is achieved as with drivers that produce PDF output.
4. Although fully supported, opacity and fadings are PDF features that become visible only after further processing the PostScript output with ps2pdf. Note that newer Ghostscript versions are necessary for producing opacity in the PDF output. Also, beginning with version 9.52 of Ghostscript, command line option -dALLOWPSTRANSPARENCY must be added:

   ```
   ps2pdf -dALLOWPSTRANSPARENCY example.ps
   ```

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 shares the restrictions of the vtex driver, but adds limited opacity support (no transparency groups, fadings and blend modes, though).

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 LaTeX: 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, Ti*k*Z 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 LaTeX 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, Ti*k*Z 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 directed 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 subsubscripts) 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] {`⟨*text*⟩`}`, PGF escapes back to HTML to render the ⟨*text*⟩. 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 picture number-node number`)

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 TEX 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:

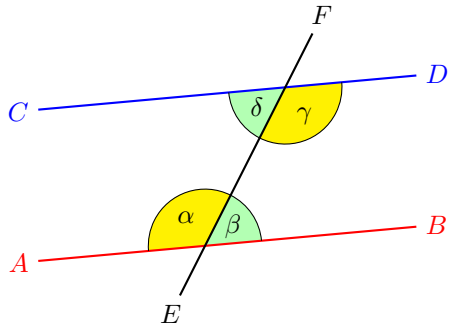
- Text boxes can be placed in the normal way.
- Lines and curves can be drawn (stroked). If they are not horizontal or vertical, they are composed of hundreds of small rectangles.
- Lines of different width are supported.
- Transformations are supported.

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., ${\color{red}AB} \mathbin{\|} \color{blue}CD$,
      then $\alpha = \gamma$ and $\beta = \delta$.
    };
\end{tikzpicture}
```

11 Design Principles

This section describes the design principles behind the Ti*k*Z frontend, where Ti*k*Z means "Ti*k*Z ist *kein* Zeichenprogramm". To use Ti*k*Z, as a LaTeX user say `\usepackage{tikz}` somewhere in the preamble, as a plain TeX user say `\input tikz.tex`. Ti*k*Z'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 Ti*k*Z 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 Ti*k*Z:

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

Ti*k*Z provides a special syntax for specifying points and coordinates. In the simplest case, you provide two TeX 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 Ti*k*Z, your main job is the specification of *paths*. A path is a series of straight or curved lines, which need not be connected. Ti*k*Z 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 ◺ 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.

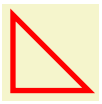
Given a path as in `\path (0,0) rectangle (2ex,1ex);`, you can draw it by adding the `draw` option as in `\path[draw] (0,0) rectangle (2ex,1ex);`, which yields ▭. The `\draw` command is just an abbreviation for `\path[draw]`. To fill a path, use the `fill` option or the `\fill` command, which is an abbreviation 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.

Ti*k*Z allows you to use different colors for filling and stroking.

11.4 Key–Value Syntax for Graphic Parameters

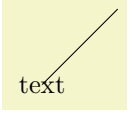
Whenever Ti*k*Z draws or fills a path, a large number of graphic parameters influences the rendering. Examples include the colors used, the dashing pattern, the clipping area, the line width, and many others. In Ti*k*Z, all these options are specified as lists of so called key–value pairs, as in `color=red`, 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

Ti*k*Z 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);
```

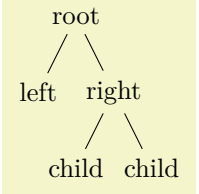
Nodes are inserted at the current position of the path, but either *after* (the default) or *before* the complete path is rendered. When special options are given, as in `\draw (1,1) node[circle,draw] {text};`, the text is not just put at the current position. Rather, it is surrounded by a circle and 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.

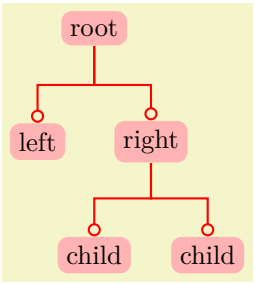
11.6 Special Syntax for Specifying Trees

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.

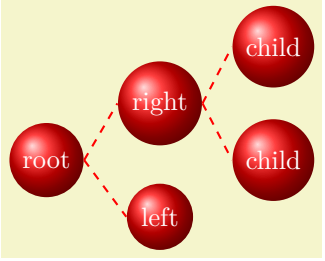



```
\begin{tikzpicture}
  \node {root}
    child {node {left}}
    child {node {right}
      child {node {child}}
      child {node {child}}
    };
\end{tikzpicture}
```

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:



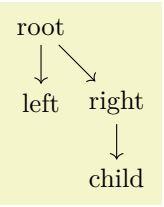

```
\usetikzlibrary {arrows.meta,trees}
\begin{tikzpicture}
  [edge from parent fork down, sibling distance=15mm, level distance=15mm,
   every node/.style={fill=red!30,rounded corners},
   edge from parent/.style={red,-{Circle[open]},thick,draw}]
  \node {root}
      child {node {left}}
      child {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.



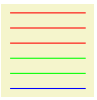

```
\usetikzlibrary {graphs}
\tikz \graph [grow down, branch right] {
  root -> { left, right -> {child, child} }
};
```

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 Ti*k*Z 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

Ti*k*Z 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 99.4.)

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 Ti*k*Z. 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} % LaTeX
\input tikz.tex   % plain TeX
\usemodule[tikz]  % ConTeXt
```

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 ConTEXt together with `dvipdfm`. In this case, you have to write `\def\pgfsysdriver{pgfsys-dvipdfm.def}` *before* you input `tikz.tex`.

`\usetikzlibrary{`⟨*list of libraries*⟩`}`

Once Ti*k*Z 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 ConTEXt 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 ⟨*list of libraries*⟩. 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 ConTEXt users can then use your library.

12.2 Creating a Picture

12.2.1 Creating a Picture Using an Environment

The "outermost" scope of Ti*k*Z 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 Ti*k*Z, 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}⟨animations spec⟩[⟨options⟩]
  ⟨environment contents⟩
\end{tikzpicture}
```

All Ti*k*Z 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 Ti*k*Z, 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 ⟨*options*⟩ 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

$A \twoheadrightarrow B$

```
$A \mathbin{\tikz[baseline] \draw[->>] (0pt,.5ex) -- (3ex,.5ex);} B$
```

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.

Hello ~~world.~~

```
\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 Ti*k*Z 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, Ti*k*Z 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.

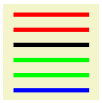
12.3 Using Scopes to Structure a Picture

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

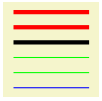
There is a small library that makes using scopes a bit easier:

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, `{ [ultra thick]` actually causes a `\begin{scope}[ultra thick]` to be inserted, and the corresponding closing `}` causes an `\end{scope}` to be inserted.

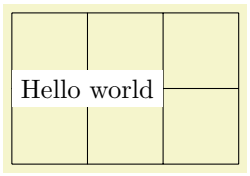
The "certain places" where an opening brace has this special meaning are the following: First, right after the semicolon that ends a path. Second, right after the end of a scope. Third, right at the beginning of a scope, which includes the beginning of a picture. Also note that some square bracket must 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 cumbersome to create a `\begin{scope}` and `\end{scope}` pair just for a single command, the `\scoped` command may be useful:

\scoped⟨*animations spec*⟩[⟨*options*⟩]⟨*path command*⟩

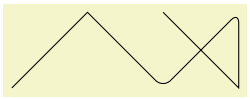
This command works like `\tikz`, only you can use it inside a `{tikzpicture}`. It will take the following ⟨*path command*⟩ and put it inside a `{scope}` with the ⟨*options*⟩ set. The ⟨*path command*⟩ may either be a single command ended by a semicolon or it may contain multiple commands, but then they 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{`⟨*options*⟩`}`

This command will process the ⟨*options*⟩ using the `\pgfkeys` command, documented in detail in Section 87, 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 87 once more.

When a pair ⟨*key*⟩=⟨*value*⟩ is processed, the following happens:

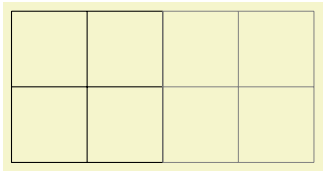
1. If the ⟨*key*⟩ is a full key (starts with a slash) it is handled directly as described in Section 87.
2. Otherwise (which is usually the case), it is checked whether `/tikz/`⟨*key*⟩ is a key and, if so, it is executed.
3. Otherwise, it is checked whether `/pgf/`⟨*key*⟩ is a key and, if so, it is executed.
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.



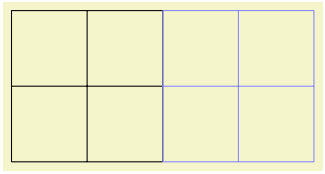
```
\begin{tikzpicture}
  \draw             (0,0) grid +(2,2);
  \draw[help lines] (2,0) grid +(2,2);
\end{tikzpicture}
```

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}
```

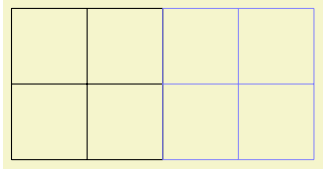
The meaning of the curious `/.style` is the following: "The key `my style` should not be used here but, rather, be defined. So, set up things such that using the key `my style` will, in the following, have the same effect as if we had written `draw=red,fill=red!20` instead."

Returning to the help lines example, suppose we prefer blue help lines. This could be achieved as follows:



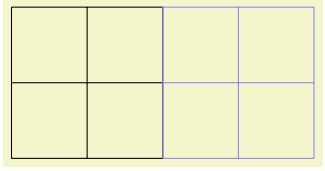
```
\begin{tikzpicture}[help lines/.style={blue!50,very thin}]
  \draw             (0,0) grid +(2,2);
  \draw[help lines] (2,0) grid +(2,2);
\end{tikzpicture}
```

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}
```

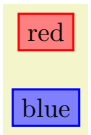
Since styles are just special cases of `pgfkeys`'s general style facility, you can actually do quite a bit more. Let us start with adding options to an already existing style. This is done using `/.append style` instead of `/.style`:



```
\begin{tikzpicture}[help lines/.append style=blue!50]
  \draw             (0,0) grid +(2,2);
  \draw[help lines] (2,0) grid +(2,2);
\end{tikzpicture}
```

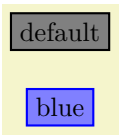
In the above example, the option `blue!50` is appended to the style `help lines`, which now has the same effect as `black!50,very thin,blue!50`. Note that two colors are set, so the last one will "win". There also exists a handler called `/.prefix style` that adds something at the beginning of the style.

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 87.

13 Specifying Coordinates

13.1 Overview

A *coordinate* is a position on the canvas on which your picture is drawn. Ti*k*Z uses a special syntax for specifying coordinates. Coordinates are always put in round brackets. The general syntax is `([`⟨*options*⟩`]`⟨*coordinate specification*⟩`)`.

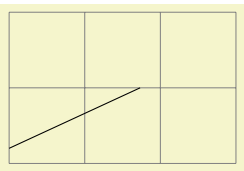
The ⟨*coordinate specification*⟩ specifies coordinates using one of many different possible *coordinate systems*. Examples are the Cartesian coordinate system or polar coordinates or spherical coordinates. No matter which coordinate system is used, in the end, a specific point on the canvas is represented 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*⟩`)`.

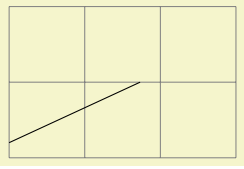
Implicitly The explicit specification is often too verbose when numerous coordinates should be given. Because of this, for the coordinate systems that you are likely to use often a special syntax is provided. Ti*k*Z will notice when you use a coordinate specified in a special syntax and will choose the correct coordinate system automatically.

Here is an example in which explicit the coordinate systems are specified explicitly:



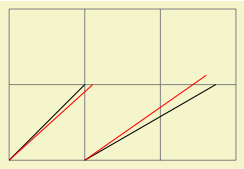
```
\begin{tikzpicture}
  \draw[help lines] (0,0) grid (3,2);
  \draw (canvas cs:x=0cm,y=2mm)
     -- (canvas polar cs:radius=2cm,angle=30);
\end{tikzpicture}
```

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`

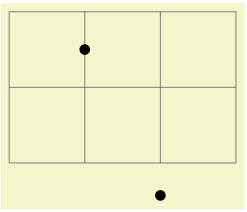
The simplest way of specifying a coordinate is to use the `canvas` coordinate system. You provide a dimension d_x using the `x=` option and another dimension d_y using the `y=` option. The position on the canvas is located at the position that is d_x to the right and d_y above the origin.

/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*⟩.

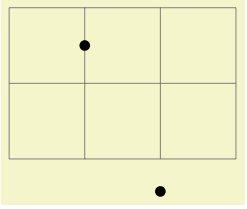
/tikz/cs/y=⟨*dimension*⟩ (no default, initially `0pt`)

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`)

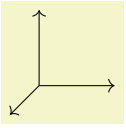
Factor by which the x-vector is multiplied.

/tikz/cs/y=⟨*factor*⟩ (no default, initially `0`)

Works like `x`.

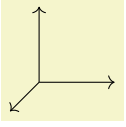
/tikz/cs/z=⟨*factor*⟩ (no default, initially `0`)

Works like `x`.



```
\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.

`/tikz/cs/radius=`⟨*dimension*⟩ (no default)

The distance from the origin.

`/tikz/cs/x radius=`⟨*dimension*⟩ (no default)

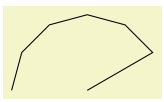
A polar coordinate is, after all, just a point on a circle of the given ⟨*radius*⟩. When you provide an x-radius and also a y-radius, you specify an ellipse instead of a circle. The `radius` option has the same effect as specifying identical `x radius` and `y radius` options.

`/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 ⌴. 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)

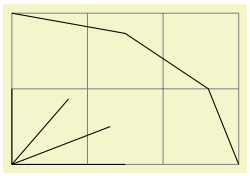
A factor by which the x-vector and y-vector are multiplied prior to forming the ellipse.

/tikz/cs/x radius=⟨*dimension*⟩ (no default)

A specific factor by which only the x-vector is multiplied.

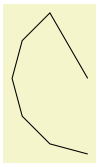
/tikz/cs/y radius=⟨*dimension*⟩ (no default)

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, …, v_n and numbers α_1, α_2, …, α_n. Then the barycentric coordinate specified by these vectors and numbers is

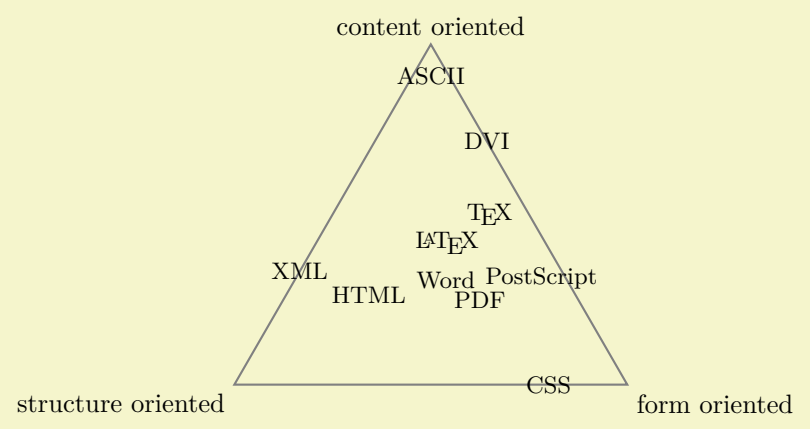
$$\frac{\alpha_1 v_1 + \alpha_2 v_2 + \cdots + \alpha_n v_n}{\alpha_1 + \alpha_2 + \cdots + \alpha_n}$$

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 Ti*k*Z 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.

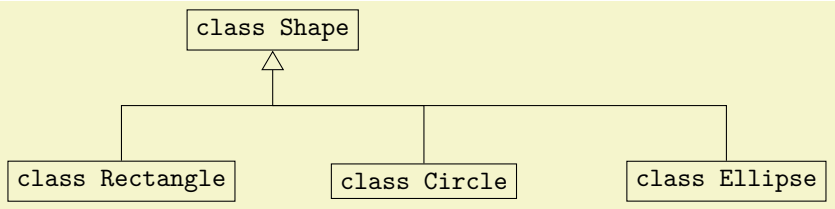
You can use three options to specify which coordinate you mean:

`/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)

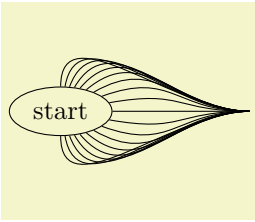
Specifies an anchor of the node. Here is an example:




```
\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}
```

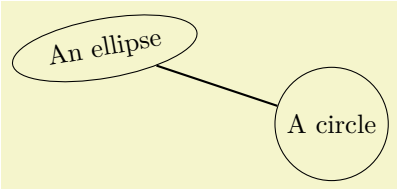
/tikz/cs/angle=⟨*degrees*⟩ (no default)

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, Ti*k*Z will calculate an appropriate border position for you. Here is an example:



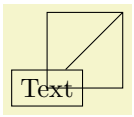

```
\usetikzlibrary {shapes.geometric}
\begin{tikzpicture}
  \path (0,0)  node(a) [ellipse,rotate=10,draw] {An ellipse}
        (3,-1) node(b) [circle,draw]            {A circle};
  \draw[thick] (node cs:name=a) -- (node cs:name=b);
\end{tikzpicture}
```

Ti*k*Z will be reasonably clever at determining the border points that you "mean", but, naturally, this may fail in some situations. If Ti*k*Z 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.

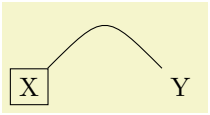
Note that if you use an automatic coordinate for both the start and the end of a line-to, as in `--(node cs:name=b)--`, then *two* border coordinates are computed with a move-to between them. This is usually exactly what you want.

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);
```

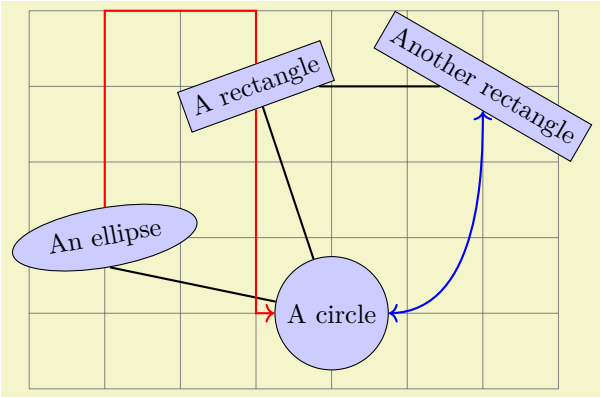
Similarly, in the following examples both control points are $(1,1)$:




```
\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]
  \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 Ti*k*Z 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)

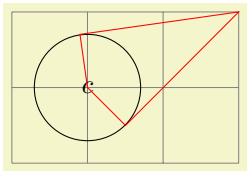
This key specifies the point through which the tangent should go.

/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 Ti*k*Z 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:

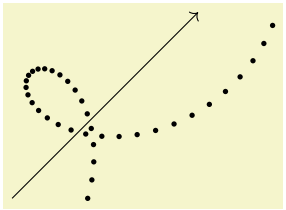
\tikzdeclarecoordinatesystem{⟨*name*⟩}{⟨*code*⟩}

This command declares a new coordinate system named ⟨*name*⟩ that can later on be used by writing (⟨*name*⟩ `cs:`⟨*arguments*⟩). When Ti*k*Z 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,...,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 `intersections` 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.

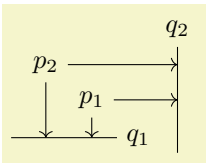
`/tikz/cs/vertical line through={(`⟨*coordinate*⟩`)}` (no default)

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 `(`⟨*p*⟩ `|-` ⟨*q*⟩`)` or `(`⟨*q*⟩ `-|` ⟨*p*⟩`)`.

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).

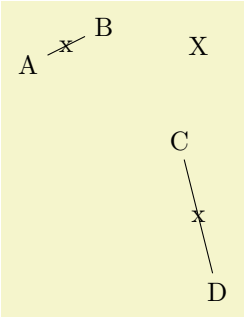
The most useful application of the syntax is to draw a line up to some point on a vertical or horizontal line. Here is an example:




```
\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 `(`⟨*c*⟩ `|-` ⟨*d*⟩`)` the coordinates ⟨*c*⟩ and ⟨*d*⟩ 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 ({$(A)!.5!(B)$} -| {$(C)!.5!(D)$}) {X};
\end{tikzpicture}
```

13.3.2 Intersections of Arbitrary Paths

Ti*k*Z Library `intersections`

```
\usetikzlibrary{intersections} % LaTeX and plain TeX
\usetikzlibrary[intersections] % ConTeXt
```

This library enables the calculation of intersections of two arbitrary paths. However, due to the low accuracy of TEX, 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 Ti*k*Z, 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 the is useful for the intersection computation):

`/tikz/name path=`⟨*name*⟩ (no default)
`/tikz/name path global=`⟨*name*⟩ (no default)

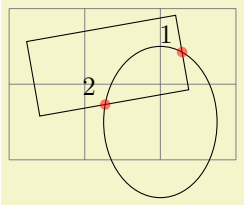
The effect of this key is that, after the path has been constructed, just before it is used, it is associated with ⟨*name*⟩. For `name path`, this association survives beyond the final semi-colon of the path but not the end of the surrounding scope. For `name path global`, the association will survive 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}
    (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)

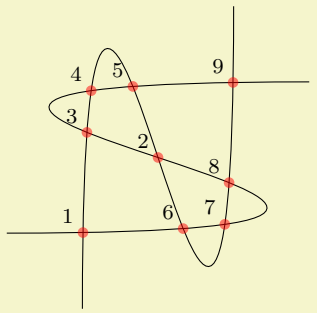
This key is used to specify the names of the paths to use for the intersection.

`/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*⟩.

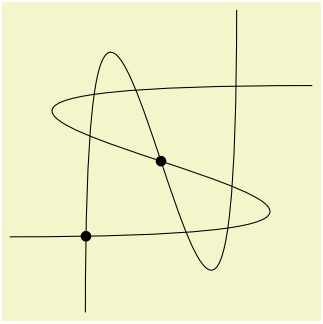



```
\usetikzlibrary {intersections}
\begin{tikzpicture}
  \clip (-2,-2) rectangle (2,2);
  \draw [name path=curve 1] (-2,-1) .. controls (8,-1) and (-8,1) .. (2,1);
  \draw [name path=curve 2] (-1,-2) .. controls (-1,8) and (1,-8) .. (1,2);

  \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

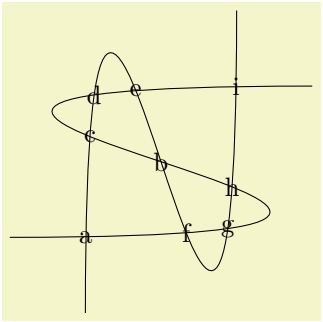
be named by the first element of the ⟨*comma-separated list*⟩. What happens is that the ⟨*comma-separated list*⟩ is passed to the `\foreach` statement and for ⟨*list member*⟩ a coordinate is created at the already-named intersection.



```
\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*⟩.

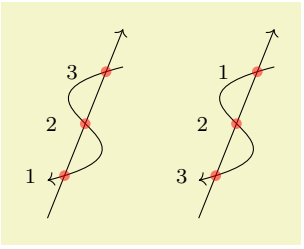
In case an element of the ⟨*comma-separated list*⟩ starts with options in square brackets, these options are used when the coordinate is created. A coordinate name can still, but need not, follow the options. This makes it easy to add labels to intersections:




```
\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:...],[label=center:i]}}];
\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.

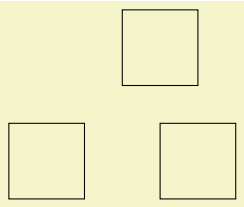



```
\usetikzlibrary {intersections}
\begin{tikzpicture}
\clip (-0.5,-0.75) rectangle (3.25,2.25);
\foreach \pathname/\shift in {line/0cm, curve/2cm}{
  \tikzset{xshift=\shift}
  \draw [->, name path=curve] (1,1.5) .. controls (-1,1) and (2,0.5) .. (0,0);
  \draw [->, name path=line]  (0,-.5) -- (1,2) ;
  \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

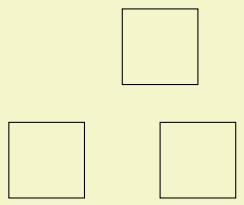
13.4.1 Specifying Relative Coordinates

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}
```

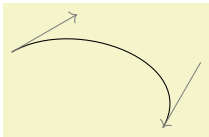
Instead of `++` you can also use a single `+`. This also specifies a relative coordinate, but it does not "update" the current point for subsequent usages of relative coordinates. Thus, you can use this notation to specify numerous points, all relative to the same "initial" point:



```
\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°:



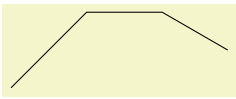
```
\begin{tikzpicture}
  \draw (1,0) .. controls +(30:1cm) and +(60:1cm) .. (3,-1);
  \draw[gray,->] (1,0) -- +(30:1cm);
  \draw[gray,<-] (3,-1) -- +(60:1cm);
\end{tikzpicture}
```

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)

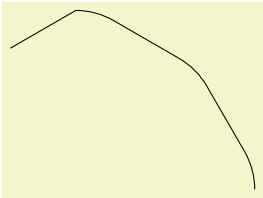
This key can be given as an option to a ⟨*coordinate*⟩ as in the following example:



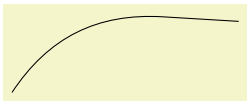
```
\tikz \draw (0,0) -- (1,1) -- ([turn]-45:1cm) -- ([turn]-30:1cm);
```

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 …

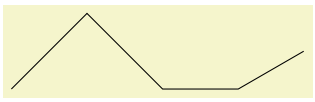


```
\tikz [delta angle=30, radius=1cm]
  \draw (0,0) arc [start angle=0]  -- ([turn]0:1cm)
              arc [start angle=30] -- ([turn]0:1cm)
              arc [start angle=60] -- ([turn]30:1cm);
```



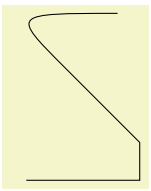
```
\tikz \draw (0,0) to [bend left] (2,1) -- ([turn]0:1cm);
```

...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);
```

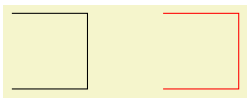
13.4.3 Relative Coordinates and Scopes

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.

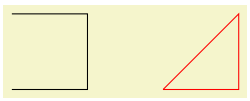
`/tikz/current point is local`=⟨*boolean*⟩ (no default, initially `false`)

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 behavior 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 Ti*k*Z 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.

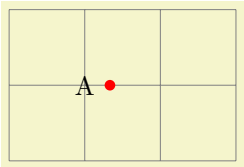
13.5 Coordinate Calculations

Ti*k*Z 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:

`([`⟨*options*⟩`]$`⟨*coordinate computation*⟩`$)`.

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 ⟨*coordinate computation*⟩ has the following structure:

1. It starts with

   ⟨*factor*⟩`*`⟨*coordinate*⟩⟨*modifiers*⟩

2. This is optionally followed by `+` or `-` and then another

   ⟨*factor*⟩`*`⟨*coordinate*⟩⟨*modifiers*⟩

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.

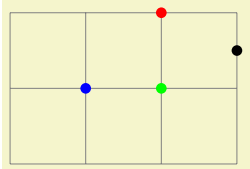
13.5.2 The Syntax of Factors

The ⟨*factor*⟩s are optional and detected by checking whether the ⟨*coordinate computation*⟩ starts with a `(`. Also, after each ± a ⟨*factor*⟩ is present if, and only if, the `+` or `-` sign is not directly followed by `(`.

If a ⟨*factor*⟩ is present, it is evaluated using the `\pgfmathparse` macro. This means that you can use pretty complicated computations inside a factor. A ⟨*factor*⟩ may even contain opening parentheses, which creates a complication: How does TikZ know where a ⟨*factor*⟩ ends and where a coordinate starts? For instance, if the beginning of a ⟨*coordinate computation*⟩ is `2*(3+4`…, it is not clear whether `3+4` is part of a ⟨*coordinate*⟩ or part of a ⟨*factor*⟩. Because of this, the following rule is used: Once it has been determined, that a ⟨*factor*⟩ is present, in principle, the ⟨*factor*⟩ 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 ⟨*factor*⟩ in curly braces. This can be used whenever it is unclear where the ⟨*factor*⟩ would end.

Here are some examples of coordinate specifications that consist of exactly one ⟨*factor*⟩ and one ⟨*coordinate*⟩:



```
\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

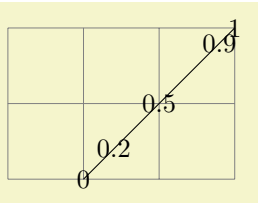
A ⟨*coordinate*⟩ can be followed by different ⟨*modifiers*⟩. The first kind of modifier is the *partway modifier*. The syntax (which is loosely inspired by Uwe Kern's `xcolor` package) is the following:

⟨*coordinate*⟩`!`⟨*number*⟩`!`⟨*angle*⟩`:`⟨*second coordinate*⟩

One could write for instance

```
(1,2)!.75!(3,4)
```

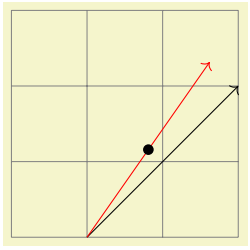
The meaning of this is: "Use the coordinate that is three quarters on the way from `(1,2)` to `(3,4)`." In general, ⟨*coordinate x*⟩`!`⟨*number*⟩`!`⟨*coordinate y*⟩ yields the coordinate $(1 - \langle number\rangle)\langle coordinate\ x\rangle + \langle number\rangle\langle coordinate\ y\rangle$. Note that this is a bit different from the way the ⟨*number*⟩ is interpreted in the `xcolor` package: First, you use a factor between 0 and 1, not a percentage, and, second, as the ⟨*number*⟩ approaches 1, we approach the second coordinate, not the first. It is permissible to use a ⟨*number*⟩ that is smaller than 0 or larger than 1. The ⟨*number*⟩ is evaluated using the `\pgfmathparse` command and, thus, it can involve complicated computations.



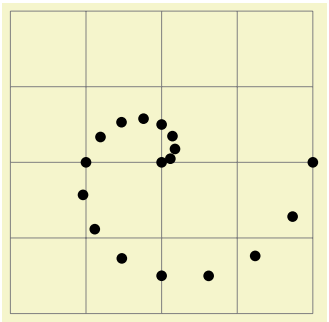

```
\usetikzlibrary {calc}
\begin{tikzpicture}
  \draw [help lines] (0,0) grid (3,2);

  \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 ⟨*second coordinate*⟩ may be prefixed by an ⟨*angle*⟩, separated with a colon, as in `(1,1)!.5!60:(2,2)`. The general meaning of ⟨*a*⟩`!`⟨*factor*⟩`!`⟨*angle*⟩`:`⟨*b*⟩ is: "First, consider the line from ⟨*a*⟩ to ⟨*b*⟩. Then rotate this line by ⟨*angle*⟩ *around the point* ⟨*a*⟩. Then the two endpoints of this line will be ⟨*a*⟩ and some point ⟨*c*⟩. Use this point ⟨*c*⟩ for the subsequent computation, namely the partway computation."

Here are two examples:

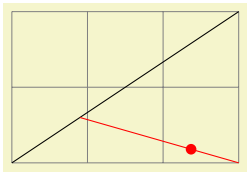


```
\usetikzlibrary {calc}
\begin{tikzpicture}
  \draw [help lines] (0,0) grid (3,3);

  \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 {0,0.125,...,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.



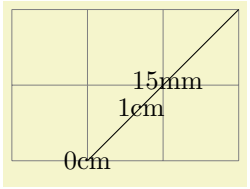
```
\usetikzlibrary {calc}
\begin{tikzpicture}
  \draw [help lines] (0,0) grid (3,2);

  \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 ⟨*dimension*⟩ (something like `1cm`) instead of a ⟨*factor*⟩ (something like `0.5`):

⟨*coordinate*⟩`!`⟨*dimension*⟩`!`⟨*angle*⟩`:`⟨*second coordinate*⟩

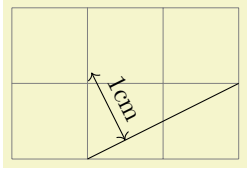
When you write ⟨*a*⟩`!`⟨*dimension*⟩`!`⟨*b*⟩, this means the following: Use the point that is distanced ⟨*dimension*⟩ from ⟨*a*⟩ on the straight line from ⟨*a*⟩ to ⟨*b*⟩. Here is an example:




```
\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 ⟨*angle*⟩, the ⟨*second coordinate*⟩ is rotated by this much around the ⟨*coordinate*⟩ before it is used.

The combination of an ⟨*angle*⟩ 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

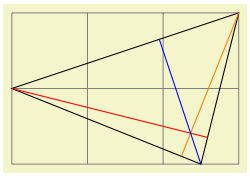
The projection modifier is also similar to the above modifiers: It also gives a point on a line from the ⟨*coordinate*⟩ to the ⟨*second coordinate*⟩. However, the ⟨*number*⟩ or ⟨*dimension*⟩ is replaced by a ⟨*projection coordinate*⟩:

⟨*coordinate*⟩`!`⟨*projection coordinate*⟩`!`⟨*angle*⟩`:`⟨*second coordinate*⟩

Here is an example:

```
(1,2)!(0,5)!(3,4)
```

The effect is the following: We project the ⟨*projection coordinate*⟩ orthogonally onto the line from ⟨*coordinate*⟩ to ⟨*second coordinate*⟩. This makes it easy to compute projected points:



```
\usetikzlibrary {calc}
\begin{tikzpicture}
  \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 Ti*k*Z 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".

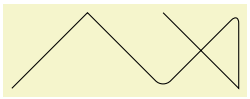
At any point where Ti*k*Z expects a path operation, you can also give some graphic options, which is a list of options in brackets, such as `[rounded corners]`. These options can have different effects:

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);
```

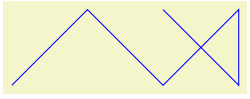
   Another example are the transformation options, which also apply only to subsequent coordinates.

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:



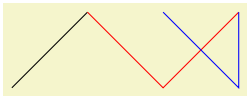
```
\tikz \draw (0,0) -- (1,1)
          {[rounded corners] -- (2,0) -- (3,1)}
          -- (3,0) -- (2,1);
```

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] (1,1) -- (2,0) -- (3,1);
      \draw [color=blue] (3,1) -- (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.

Note: When scanning for path operations TikZ expands tokens looking for valid path operations. This however implies that these tokens has to be fully expandable up to the point where it results in a valid path operation.

`/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:

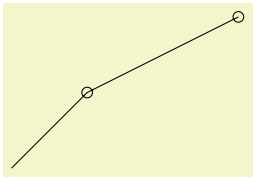
`/tikz/every path` (style, initially empty)

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.



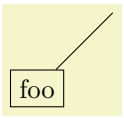
```
\tikz [c/.style={insert path={circle[radius=2pt]}}]
  \draw (0,0) -- (1,1) [c] -- (3,2) [c];
```

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)

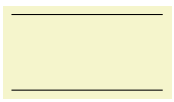
Works like `append after command`, only the accumulation order is inverse: The ⟨*path*⟩ is added before any earlier paths added using either `append after command` or `prefix after command`.

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*⟩ … ;

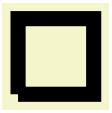
The move-to operation normally starts a path at a certain point. This does not cause a line segment to be created, but it specifies the starting point of the next segment. If a path is already under construction, that is, if several segments have already been created, a move-to operation will start a 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.

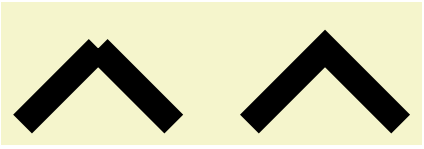
14.2 The Line-To Operation

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}
```

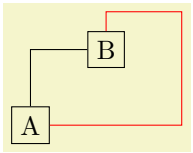
Writing `cycle` instead of a coordinate at the end of a path operation is possible with all path operations that end with a coordinate (such as `--` or `..` or `sin` or `grid`, but not `graph` or `plot`). In all cases, the effect is that the coordinate of the last moveto is used as the coordinate expected by the path operation and that a smooth join is added. (What actually happens 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 operations means "first vertical, then horizontal".

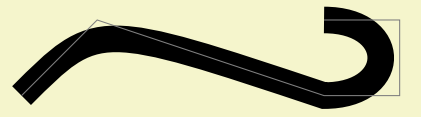
14.3 The Curve-To Operation

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.

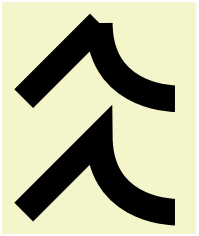
If the "`and`⟨*d*⟩" part is not given, d is assumed to be equal to c.



```
\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:



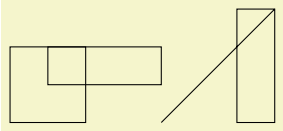
```
\begin{tikzpicture}[line width=10pt]
  \draw (0,0) -- (1,1) (1,1) .. controls (1,0) and (2,0) .. (2,0);
  \draw [yshift=-1.5cm]
        (0,0) -- (1,1)       .. controls (1,0) and (2,0) .. (2,0);
\end{tikzpicture}
```

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

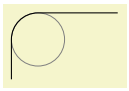
All of the path construction operations mentioned up to now are influenced by the following option:

`/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.

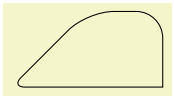
```
\tikz \draw [rounded corners] (0,0) -- (1,1)
          -- (2,0) .. controls (3,1) .. (4,0);
```

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[`⟨*options*⟩`]` …;

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 from 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`.

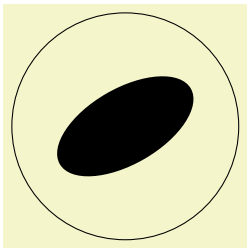
`/tikz/radius=`⟨*value*⟩ (no default)

Sets the `x radius` and `y radius` simultaneously.

`/tikz/at=`⟨*coordinate*⟩ (no default)

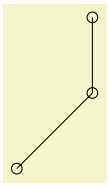
If this option is explicitly set inside the ⟨*options*⟩ (or indirectly via the `every circle` style), the ⟨*coordinate*⟩ is used as the center of the circle instead of the current point. Setting `at` to some value in an enclosing scope has no effect.

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}}
```

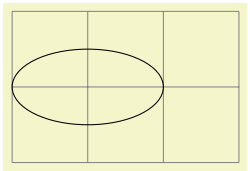
You can then say `circle [r=1cm]` or `circle [rx=1,ry=1.5]`. The reason Ti*k*Z uses the longer names by default is that it encourages people to write more readable code.

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.

Ti*k*Z 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 ⟨*options*⟩. 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.

`/tikz/start angle=`⟨*degrees*⟩ (no default)

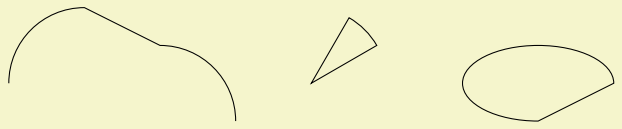
Sets the start angle.

`/tikz/end angle=`⟨*degrees*⟩ (no default)

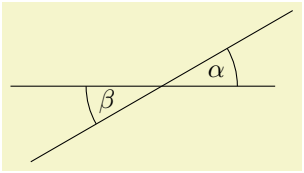
Sets the end angle.

`/tikz/delta angle=`⟨*degrees*⟩ (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}
```

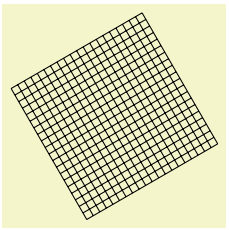
There also exists a shorter syntax for the arc operation, namely `arc` begin directly followed by `(`⟨*start angle*⟩`:`⟨*end angle*⟩`:`⟨*radius*⟩`)`. However, this syntax is harder to read, so the normal syntax should be preferred in general.

14.8 The Grid Operation

You can add a grid to the current path using the `grid` path operation.

`\path` … `grid[`⟨*options*⟩`]`⟨*corner or cycle*⟩ …`;`

This operations adds a grid filling a rectangle whose two corners are given by ⟨*corner*⟩ 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 not very natural to do so.) 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.

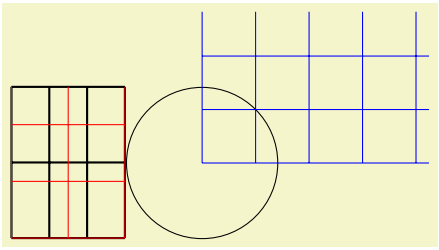


```
\tikz[rotate=30] \draw[step=1mm] (0,0) grid (2,2);
```

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.



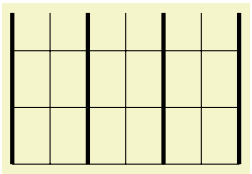
```
\begin{tikzpicture}[x=.5cm]
  \draw[thick] (0,0) grid [step=1]     (3,2);
  \draw[red]   (0,0) grid [step=.75cm] (3,2);
\end{tikzpicture}
\begin{tikzpicture}
  \draw        (0,0) circle [radius=1];
  \draw[blue]  (0,0) grid [step=(45:1)] (3,2);
\end{tikzpicture}
```

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 or the following two vectors: The first vector is either $(\langle x\text{-}grid\text{-}step\text{-}number\rangle, 0)$ or $(\langle x\text{-}grid\text{-}step\text{-}dimension\rangle, 0\text{pt})$, the second vector is $(0, \langle y\text{-}grid\text{-}step\text{-}number\rangle)$ or $(0\text{pt}, \langle y\text{-}grid\text{-}step\text{-}dimension\rangle)$.

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.

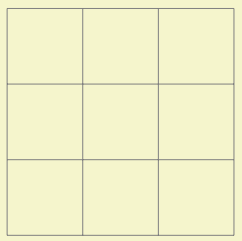
It is important to note that the grid is always "phased" such that it contains the point $(0, 0)$ if that point happens to be inside the rectangle. Thus, the grid does *not* always have an intersection at the corner points; this occurs only if the corner points are multiples of the stepping. Note that 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:



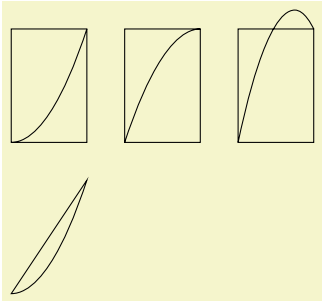
```
\tikz \draw[help lines] (0,0) grid (3,3);
```

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: $\vee$.

`\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)
                        (0,0) parabola                 (1,1.5);
  \draw[xshift=1.5cm]   (0,0) rectangle                (1,1.5)
                        (0,0) parabola[bend at end]    (1,1.5);
  \draw[xshift=3cm]     (0,0) rectangle                (1,1.5)
                        (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.

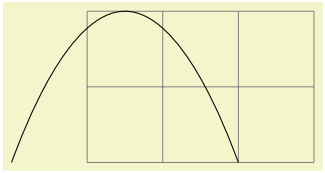
There is one special property of the ⟨*coordinate*⟩: When a relative coordinate is given like `+(0,0)`, the position relative to this coordinate is "flexible". More precisely, this position lies somewhere on a line from the old current point to the new current point. The exact position depends on the 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:

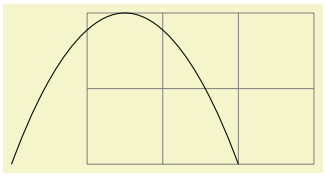


```
\begin{tikzpicture}
  \draw[help lines] (0,0) grid (3,2);
  \draw (-1,0) parabola[bend pos=0.5] bend +(0,2) +(3,0);
\end{tikzpicture}
```

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)

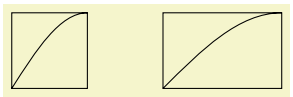
This places the bend at the end of a parabola.

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*⟩ … ;

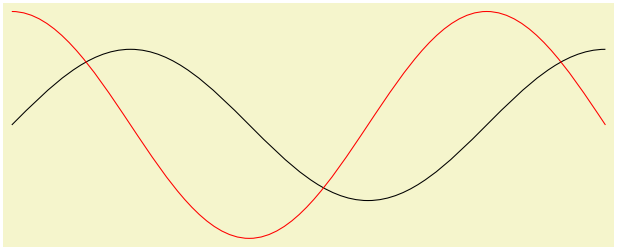
The effect of `sin` is to draw a scaled and shifted version of a sine curve in the interval $[0, \pi/2]$. The scaling and shifting is done in such a way that the start of the sine curve in the interval is at the old current point and that the end of the curve in the interval is at ⟨*coordinate*⟩. Here 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 $\pi/2$.

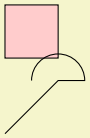
14.11 The SVG Operation

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[`⟨*options*⟩`]{`⟨*path data*⟩`}` … `;`

This operation adds the path specified in the ⟨*path data*⟩ 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 ⟨*options*⟩ apply locally to this path operation, typically you will use them to set up, say, some transformations.

upper left



```
\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 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 instable. 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 behavior of this command are a bit complex, they are described in the separated Section 22.

14.13 The To Path Operation

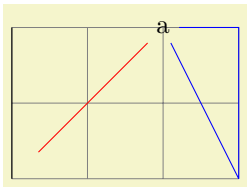
The `to` operation is used to add a user-defined path from the previous coordinate to the following coordinate. When you write `(a) to (b)`, a straight line is added from `a` to `b`, exactly as if you had written `(a) -- (b)`. However, if you write `(a) to [out=135,in=45] (b)` a curve is added to the 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[`⟨*options*⟩`]` ⟨*nodes*⟩ ⟨*coordinate or cycle*⟩ … `;`

This path operation inserts the path currently set via the `to path` option at the current position. The ⟨*options*⟩ can be used to modify (perhaps implicitly) the `to path` and to set up how the path will be rendered.

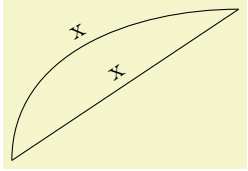
Before the `to path` is inserted, a number of macros are set up that can "help" the `to path`. These are `\tikztostart`, `\tikztotarget`, and `\tikztonodes`; they are explained in the following.

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 `to`s:

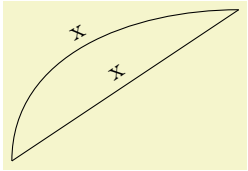



```
\begin{tikzpicture}
  \draw (0,0) to node [sloped,above] {x} (3,2);

  \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:

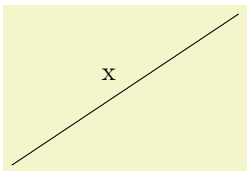



```
\begin{tikzpicture}
  \draw (0,0) to [edge node={node [sloped,above] {x}}] (3,2);

  \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)

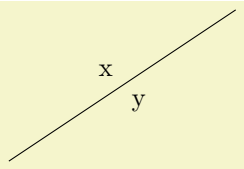
A shorthand for `edge node={node[auto]{`⟨*text*⟩`}}`.




```
\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*⟩`}}`.



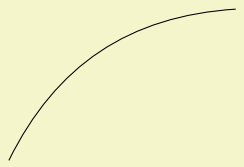

```
\tikz \draw (0,0) to [edge label=x, edge label'=y] (3,2);
```

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:

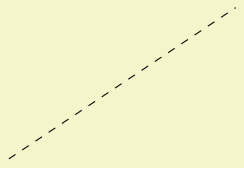
`/tikz/every to` (style, initially empty)

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 ⟨*path*⟩ will expand to

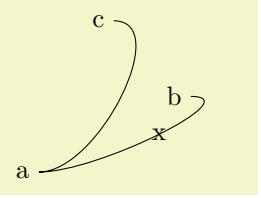
```
(a) -- (b) node[red] {x}
```

It is not possible to specify the path

```
-- \tikztonodes (\tikztotarget)
```

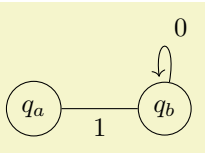
since Ti*k*Z does not allow one to have a macro after `--` that expands to a node.

Now let us have a look at how we can modify the ⟨*path*⟩ 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           node[below]       {1} (b)
                to [my loop] node[above right] {0} (b);
\end{tikzpicture}
```

`/tikz/execute at begin to=`⟨*code*⟩ (no default)

The ⟨*code*⟩ 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=`⟨*code*⟩ (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)

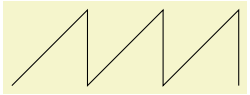
This style is installed at the beginning of every to.

There are a number of predefined `to path`s, see Section 74 for a reference.

14.14 The Foreach Operation

`\path` … `foreach`⟨*variables*⟩`[`⟨*options*⟩`]` `in` ⟨*list*⟩ `{`⟨*path commands*⟩`}` … `;`

The `foreach` operation can be used to repeatedly insert the ⟨*path commands*⟩ into the current path. Naturally, the ⟨*path commands*⟩ should internally reference some of the ⟨*variables*⟩ 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 88 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. It requires the `calc` library, see Section 13.5.

`\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 Ti*k*Z, 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*3cm/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*⟩`}`

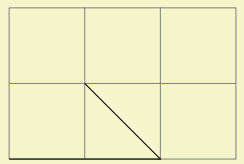
When this macro is used on the left-hand side of an `=`-sign in a let operation, it has no effect and is just there for readability. When the macro is used on the right-hand side of an `=`-sign or in the body of the let operation, then it expands to the value stored in the ⟨*number register*⟩. This will 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*⟩`=`⟨*coordinate*⟩

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*⟩`}`

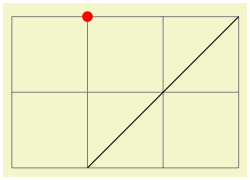
This macro expands just to the x-part of the point register. If we say as above, as we did above, `let \p1=(1pt,1pt+2pt) in ...`, then inside the `...` part the macro `\x1` expands to `1pt`.

`\y{`⟨*point register*⟩`}`

Works like `\x`, only for the y-part.

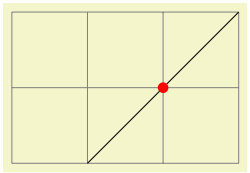
Note that the above macros are available only inside a let operation.

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 `|-` notation, this could be written much more compactly.



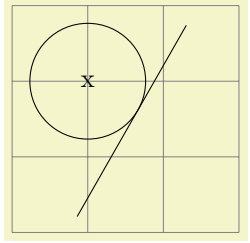
```
\usetikzlibrary {calc}
\begin{tikzpicture}
  \draw [help lines] (0,0) grid (3,2);

  \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 and 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.

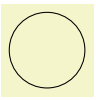
14.20 The Attribute Animation Operation

`\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}
\tikz \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 Ti*k*Z'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 Ti*k*Z 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 Ti*k*Z 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 Ti*k*Z 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 121.

`/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 (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 LaTeX users, this is just a normal "LaTeX-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 Ti*k*Z does not know, like `red!20`, gets a "second chance" as a color name.

For plain TeX users, it is not so easy to specify colors since plain TeX 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 supported[3].

  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 TeX 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.

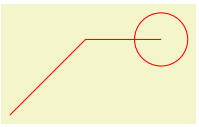
There are numerous parameters that influence how a line is drawn, like the thickness or the dash pattern. These options are explained below.

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.

[3] ConTeXt users should be aware that `\definecolor` has a different meaning in ConTeXt. 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}
```

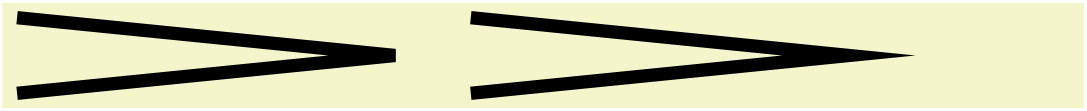
/tikz/line join=⟨*type*⟩ (no default, initially `miter`)

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}
```

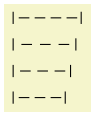
/tikz/dash=⟨*dash pattern*⟩`phase`⟨*dash phase*⟩ (no default)

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);
```

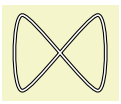
15.3.3 Graphic Parameters: Draw Opacity

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 Ti*k*Z 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}
```

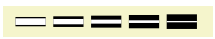
/tikz/double distance=⟨*dimension*⟩ (no default, initially 0.6pt)

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.

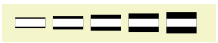
```
\begin{tikzpicture}
  \draw[very thick,double]                (0,0) arc (180:90:1cm);
  \draw[very thick,double distance=2pt]   (1,0) arc (180:90:1cm);
  \draw[thin,double distance=2pt]         (2,0) arc (180:90:1cm);
\end{tikzpicture}
```

/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.



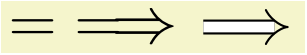
```
\begin{tikzpicture}[double distance between line centers=3pt]
  \foreach \lw in {0.5,1,1.5,2,2.5}
    \draw[line width=\lw pt,double] (\lw,0) -- ++(4mm,0);
\end{tikzpicture}
```



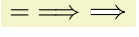
```
\begin{tikzpicture}[double distance=3pt]
  \foreach \lw in {0.5,1,1.5,2,2.5}
    \draw[line width=\lw pt,double] (\lw,0) -- ++(4mm,0);
\end{tikzpicture}
```

/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 subpaths, 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:

```
\usetikzlibrary {arrows.meta,bending}
\tikz \path[tips, -{Latex[open,length=10pt,bend]}] (0,0) to[bend left] (1,0);
```

```
\usetikzlibrary {arrows.meta,bending}
\tikz \draw[tips, -{Latex[open,length=10pt,bend]}] (0,0) to[bend left] (1,0);
```

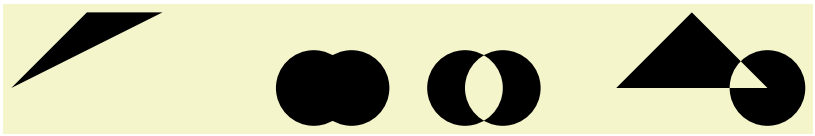
15.5 Filling a Path

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.

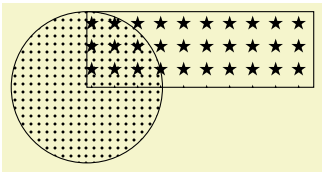
Patterns are not overly flexible. In particular, it is not possible to change the size or orientation of a pattern without declaring a new pattern. For complicated cases, it may be easier to use two nested `\foreach` statements to simulate a pattern, but patterns are rendered *much* more quickly than 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.

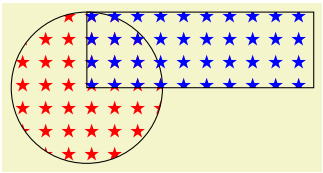
Strangely, no ⟨*name*⟩s are permissible by default. You need to load for instance the `patterns` library, see Section 62, to install predefined patterns.



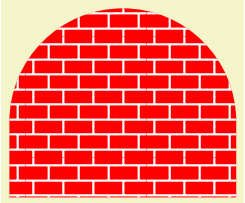
```
\usetikzlibrary {patterns}
\begin{tikzpicture}
  \draw[pattern=dots] (0,0) circle (1cm);
  \draw[pattern=fivepointed stars] (0,0) rectangle (3,1);
\end{tikzpicture}
```

`/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}
```



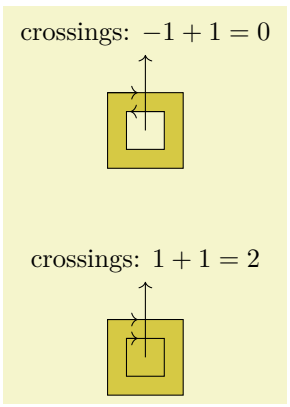
```
\usetikzlibrary {patterns}
\begin{tikzpicture}
  \def\mypath{(0,0) -- +(0,1) arc (180:0:1.5cm) -- +(0,-1)}
  \fill   [red]                                  \mypath;
  \pattern[pattern color=white,pattern=bricks] \mypath;
\end{tikzpicture}
```

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.



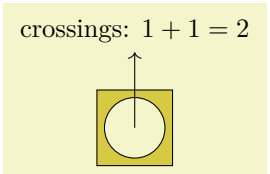

```
\begin{tikzpicture}
  \filldraw[fill=yellow!80!black]
  % Clockwise rectangle
  (0,0) -- (0,1) -- (1,1) -- (1,0) -- cycle
  % Counter-clockwise rectangle
  (0.25,0.25) -- (0.75,0.25) -- (0.75,0.75) -- (0.25,0.75) -- cycle;

  \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.

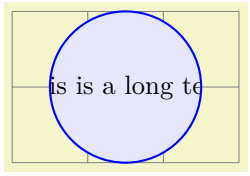
As with other keys like `fill` or `draw` this option needs to be given on a path, setting the `path picture` outside a path has no effect (the path picture is cleared at the beginning of each path).

The ⟨*code*⟩ can be any normal Ti*k*Z code like `\draw ...` or `\node ...`. As always, when you include an external graphic, you need to put it inside a `\node`.

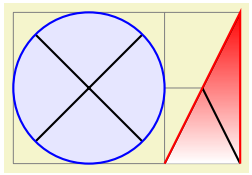
Note that no special actions are taken to transform the origin in any way. This means that the coordinate `(0,0)` is still where is was when the path was being constructed and not – as one might expect – at the lower left corner of the path. However, you can use the following special node to access the size of the path:

Predefined node `path picture bounding box`

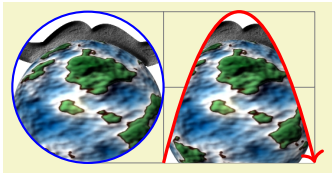
This node is of shape `rectangle`. Its size and position are those of `current path bounding box` just before the ⟨*code*⟩ of the path picture started to be executed. The ⟨*code*⟩ can construct its own paths, so accessing the `current path bounding box` inside the ⟨*code*⟩ yields the bounding box of 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=brave-gnu-world-logo,draw=blue,thick]
        (0,1) circle (1);
\draw [path image=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);
```

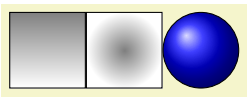
For some shadings it is not really clear how they can "fill" the path. For example, the `ball` shading normally looks like this:  . How is this supposed to shade a rectangle? Or a triangle?

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 69.

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 69 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 114.3.

15.8 Establishing a Bounding Box

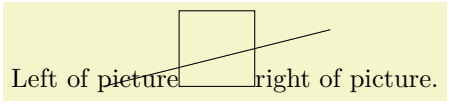
PGF is reasonably good at keeping track of the size of your picture and reserving just the right amount of space for it in the main document. However, in some cases you may want to say things like "do not count this for the picture size" or "the picture is actually a little large". For this you can use 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:

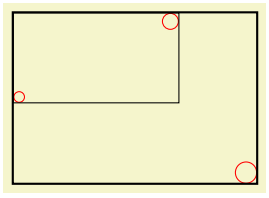

```
Left of picture
\begin{tikzpicture}
  \useasboundingbox (0,0) rectangle (3,1);
  \fill (.75,.25) circle (.5cm);
\end{tikzpicture}
right of picture.
```

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.



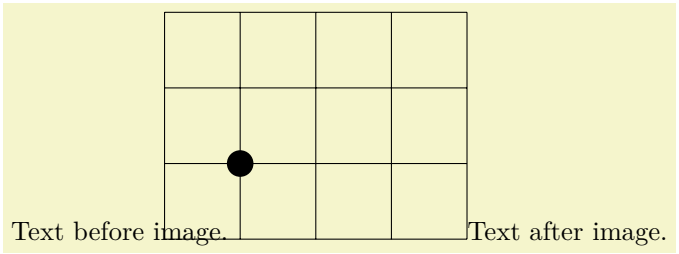
```
\begin{tikzpicture}
  \draw[red] (0,0) circle (2pt);
  \draw[red] (2,1) circle (3pt);

  \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`)

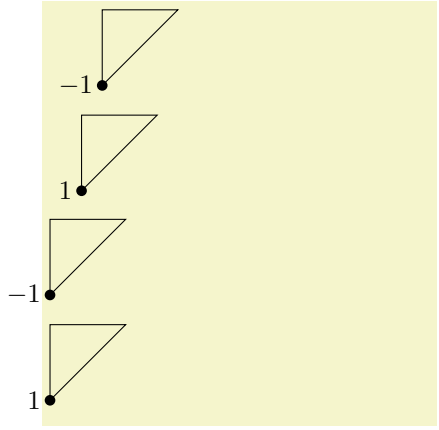
The `trim left` key tells PGF to discard everything which is left of the provided ⟨*dimension or coordinate*⟩. Here, ⟨*dimension*⟩ is a single x coordinate of the picture and ⟨*coordinate*⟩ is a point with x and y coordinates (but only its x coordinate will be used). The effect is the same as if you 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`:



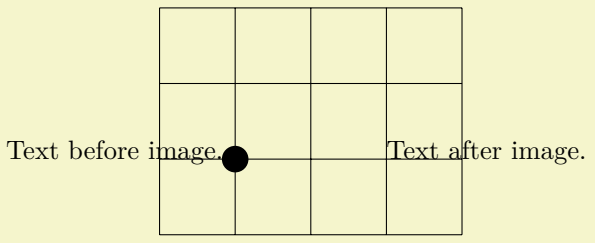

```
\begin{tikzpicture}
    \draw (0,1) -- (0,0) -- (1,1) -- cycle;
    \fill (0,0) circle (2pt);
    \node[left] at (0,0) {$-1$};
\end{tikzpicture}
\par
\begin{tikzpicture}
    \draw (0,1) -- (0,0) -- (1,1) -- cycle;
    \fill (0,0) circle (2pt);
    \node[left] at (0,0) {$1$};
\end{tikzpicture}
\par
\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)

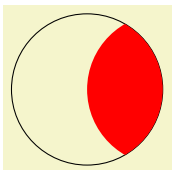
Clipping path means that all painting on the page is restricted to a certain area. This area need not be rectangular, rather an arbitrary path can be used to specify this area. The `clip` option, explained below, is used to specify the region that is to be used for clipping.

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.

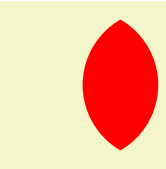
The clipping path is a graphic state parameter, so it will be reset at the end of the current scope. Multiple clippings accumulate, that is, clipping is always done against the intersection of all clipping areas that have been specified inside the current scopes. The only way of enlarging the clipping 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

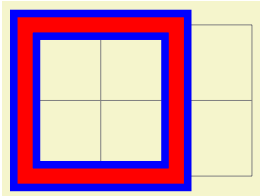
If more than one of the basic actions like drawing, clipping and filling are requested, they are automatically applied in a sensible order: First, a path is filled, then drawn, and then clipped (although it took Apple two major revisions of their operating system to get this right…). Sometimes, however, 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.

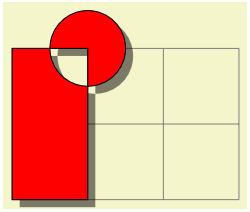
Here is an example in which the path consists of a rectangle. The main action is to draw this path in red (which is why we see a red rectangle). However, the preaction is to draw the path in blue, which is why we see a blue rectangle behind the red rectangle.



```
\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 preformed, 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:

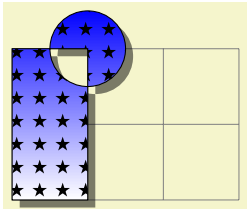


```
\begin{tikzpicture}
  \draw[help lines] (0,0) grid (3,2);
  \draw
    [preaction={fill=black,opacity=.5,
                transform canvas={xshift=1mm,yshift=-1mm}}]
    [fill=red] (0,0) rectangle (1,2)
               (1,2) circle (5mm);
\end{tikzpicture}
```

Naturally, you would normally create a style `shadow` that contains the above code. The `shadows` library, see Section 70, contains predefined shadows of this kind.

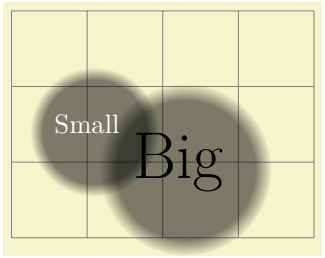
It is possible to use the `preaction` option multiple times. In this case, for each use of the `preaction` option, the path is used again (thus, the ⟨*options*⟩ do not accumulate in a single usage of the path). The path is used in the order of `preaction` options given.

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:



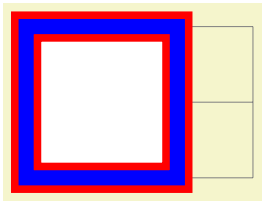

```
\usetikzlibrary {fadings,patterns}
\begin{tikzpicture}
  [
    % Define an interesting style
    button/.style={
      % First preaction: Fuzzy shadow
      preaction={fill=black,path fading=circle with fuzzy edge 20 percent,
                 opacity=.5,transform canvas={xshift=1mm,yshift=-1mm}},
      % 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.

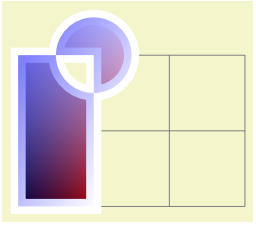
If both pre- and postactions are specified, then the preactions are taken first, then the main action, and then the post actions.

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}
```

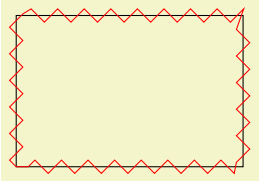
In another example, we use a postaction to "colorize" a path:



```
\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}
```

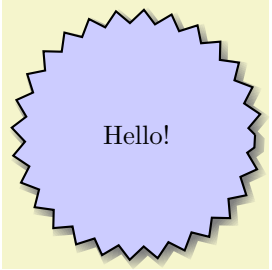
15.11 Decorating and Morphing a Path

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:

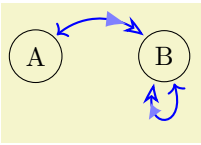



```
\usetikzlibrary {decorations.pathmorphing,shadows}
\begin{tikzpicture}
  \node [circular drop shadow={shadow scale=1.05},minimum size=3.13cm,
         decorate, decoration=zigzag,
         fill=blue!20,draw,thick,circle] {Hello!};
\end{tikzpicture}
```

16 Arrows

16.1 Overview

Ti*k*Z allows you to add (multiple) arrow tips to the end of lines as in ⟶» or in ⟶►. 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}
\tikz {
  \node [circle,draw] (A)              {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);
}
```

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 105, 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 105.

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 Ti*k*Z. 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 Ti*k*Z 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 Ti*k*Z 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.

```
% No path, no arrow tips:
\tikz [<->] \draw;
```

```
% Degenerate path, draw arrow tips (but no path, it is degenerate...)
\tikz [<->] \draw (0,0);
```

```
% Degenerate path, tips=proper suppresses arrows
\tikz [<->] \draw [tips=proper] (0,0);
```

```
% Normal case:
\tikz [<->] \draw (0,0) -- (1,0);
```

```
% Two subpaths, only second gets tips
\tikz [<->] \draw (0,0) -- (1,0) (2,0) -- (3,0);
```

```
% Two subpaths, second degenerate, but still gets tips
\tikz [<->] \draw (0,0) -- (1,0) (2,0);
```

```
% Two subpaths, second degenerate, proper suppresses them
\tikz [<->] \draw [tips=on proper draw] (0,0) -- (1,0) (2,0);
```

```
% 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);
```

One common pitfall when arrow tips are added to a path should be addressed right here at the beginning: When Ti*k*Z 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 that the arrow tip itself, strange things may result. As will be explained in Section 16.3.8, Ti*k*Z 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, Ti*k*Z 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.

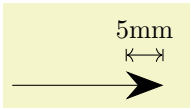
Some of the keys explained in the following are defined in the library `arrows.meta`, others are always available. This has to do with the question of whether the arrow key needs to be supported directly in the PGF core or not. In general, the following explanations assume that `arrows.meta` has been loaded.

16.3.1 Size

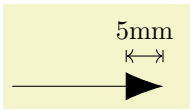
The most important configuration parameter of an arrow tip is undoubtedly its size. The following two keys are the main keys that are important in this context:

`/pgf/arrow keys/length`=⟨*dimension*⟩ ⟨*line width factor*⟩ ⟨*outer factor*⟩ (no default)

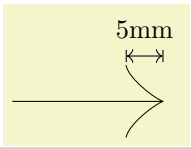
This parameter is usually the most important parameter that governs the size of an arrow tip: The ⟨*dimension*⟩ that you provide dictates the distance from the "very tip" of the arrow to its "back end" along the line:




```
\usetikzlibrary {arrows.meta}
\tikz{
  \draw [-{Stealth[length=5mm]}] (0,0) -- (2,0);
  \draw [|<->|] (1.5,.4) -- node[above=1mm] {5mm} (2,.4);
}
```




```
\usetikzlibrary {arrows.meta}
\tikz{
  \draw [-{Latex[length=5mm]}] (0,0) -- (2,0);
  \draw [|<->|] (1.5,.4) -- node[above=1mm] {5mm} (2,.4);
}
```

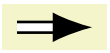



```
\usetikzlibrary {arrows.meta}
\tikz{
  \draw [-{Classical TikZ Rightarrow[length=5mm]}] (0,0) -- (2,0);
  \draw [|<->|] (1.5,.6) -- node[above=1mm] {5mm} (2,.6);
}
```

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 ⟨*dimension*⟩ + ⟨*line width factor*⟩ · 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 `3pt 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 $w_i = 3\text{pt}$, $w_o = 1\text{pt}$, and $w_t = 5\text{pt}$. It is not immediately clear which of these line widths should be considered as w in the above formula ⟨*dimension*⟩ + ⟨*line width factor*⟩ · 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 $w = \langle \textit{outer factor} \rangle w_o + (1 - \langle \textit{outer factor} \rangle) w_t$. 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 $w_t = 5\text{pt}$. Since $w = 5\text{pt}$, 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, $w = w_o = 1\text{pt}$ 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}
\tikz \draw [line width=1pt, double distance=3pt,
             arrows = {-Latex[length=0pt 3 0]}] (0,0) -- (1,0);
```

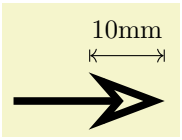
```
\usetikzlibrary {arrows.meta}
\tikz \draw [line width=1pt, double distance=3pt,
             arrows = {-Latex[length=0pt 3 .5]}] (0,0) -- (1,0);
```



```
\usetikzlibrary {arrows.meta}
\tikz \draw [line width=1pt, double distance=3pt,
             arrows = {-Latex[length=0pt 3 1]} ] (0,0) -- (1,0);
```

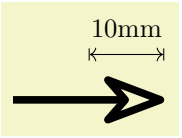
The Exact Length. For an arrow tip kind that is just an outline that is filled with a color, the specified length should *exactly* equal the distance from the tip to the back end. However, when the arrow tip is drawn by stroking a line, it is no longer obvious whether the `length` should refer to the 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}
\tikz{
  \draw [line width=1mm, -{Stealth[length=10mm, open]}]
          (0,0) -- (2,0);
  \draw [|<->|] (2,.6) -- node[above=1mm] {10mm} ++(-10mm,0);
}
```

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 want to change the mode.




```
\usetikzlibrary {arrows.meta}
\tikz{
  \draw [line width=1mm, -{Stealth[length=10mm, open, round]}]
          (0,0) -- (2,0);
  \draw [|<->|] (2,.6) -- node[above=1mm] {10mm} ++(-10mm,0);
}
```

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}
\tikz \draw [arrows = {-Latex[width=10pt, length=10pt]}] (0,0) -- (1,0);
```



```
\usetikzlibrary {arrows.meta}
\tikz \draw [arrows = {-Latex[width=0pt 10, length=10pt]}] (0,0) -- (1,0);
```

`/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 $19\text{pt} = 5\text{pt} + 2 \cdot 7\text{pt}$.


```
\usetikzlibrary {arrows.meta}
\tikz \draw [arrows = {-Latex[width'=0pt .5, length=10pt]}] (0,0) -- (1,0);
```

```
\usetikzlibrary {arrows.meta}
\tikz \draw [arrows = {-Latex[width'=0pt .5, length=15pt]}] (0,0) -- (1,0);
```

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}
\tikz \draw [arrows = {-Stealth[length=10pt, inset=5pt]}] (0,0) -- (1,0);
```

```
\usetikzlibrary {arrows.meta}
\tikz \draw [arrows = {-Stealth[length=10pt, inset=2pt]}] (0,0) -- (1,0);
```

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 13/40th 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}
\tikz \draw [arrows = {-Stealth[inset=0pt, angle=90:10pt]}] (0,0) -- (1,0);
```

```
\usetikzlibrary {arrows.meta}
\tikz \draw [arrows = {-Stealth[inset=0pt, angle=30:10pt]}] (0,0) -- (1,0);
```

`/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}
\tikz \draw [arrows = {-Stealth[inset=0pt, length=10pt, angle'=90]}]
            (0,0) -- (1,0);
```

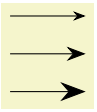
```
\usetikzlibrary {arrows.meta}
\tikz \draw [arrows = {-Stealth[inset=0pt, length=10pt, angle'=30]}]
            (0,0) -- (1,0);
```

16.3.2 Scaling

In the previous section we saw that there are many options for getting "fine control" overt 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, Ti*k*Z 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*⟩.



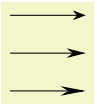
```
\usetikzlibrary {arrows.meta}
\tikz {
  \draw [arrows = {-Stealth[]}]           (0,1)   -- (1,1);
  \draw [arrows = {-Stealth[scale=1.5]}] (0,0.5) -- (1,0.5);
  \draw [arrows = {-Stealth[scale=2]}]    (0,0)   -- (1,0);
}
```

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`)

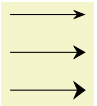
This factor works like `scale`, only it is applied only to dimensions "along the axis of the arrow", that is, to the length and to the inset, but not to the width.



```
\usetikzlibrary {arrows.meta}
\tikz {
  \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);
}
```

`/pgf/arrows keys/scale width`=⟨*factor*⟩ (no default, initially `1`)

Like `scale length`, but for dimensions related to the width.



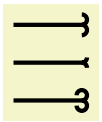
```
\usetikzlibrary {arrows.meta}
\tikz {
  \draw [arrows = {-Stealth[]}]                 (0,1)   -- (1,1);
  \draw [arrows = {-Stealth[scale width=1.5]}] (0,0.5) -- (1,0.5);
  \draw [arrows = {-Stealth[scale width=2]}]   (0,0)   -- (1,0);
}
```

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/arrow 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:



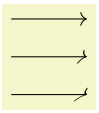
```
\usetikzlibrary {arrows.meta}
\tikz [ultra thick] {
  \draw [arrows = {-Hooks[]}]          (0,1)   -- (1,1);
  \draw [arrows = {-Hooks[arc=90]}]    (0,0.5) -- (1,0.5);
  \draw [arrows = {-Hooks[arc=270]}]   (0,0)   -- (1,0);
}
```

16.3.4 Slanting

You can "slant" arrow tips using the following key:

`/pgf/arrow keys/slant`=⟨*factor*⟩ (no default, initially 0)

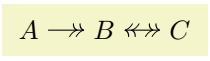
Slanting is used to create an "italics" effect for arrow tips: All arrow tips get "slanted" a little bit relative to the axis of the arrow:



```
\usetikzlibrary {arrows.meta}
\tikz {
  \draw [arrows = {->[]}]          (0,1)   -- (1,1);
  \draw [arrows = {->[slant=.5]}] (0,0.5) -- (1,0.5);
  \draw [arrows = {->[slant=1]}]   (0,0)   -- (1,0);
}
```

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:



```
\usetikzlibrary {arrows.meta,graphs}
\tikz [>={[slant=.3] To[] To[]}]
  \graph [math nodes] { A -> B <-> C };
```

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}
\tikz [ultra thick] \draw [arrows = {-Stealth[reversed]}] (0,0) -- (1,0);
```

```
\usetikzlibrary {arrows.meta}
\tikz [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}
\tikz [ultra thick] \draw [arrows = {-Stealth[harpoon]}] (0,0) -- (1,0);
```

```
\usetikzlibrary {arrows.meta}
\tikz [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 do 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 behavior, 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);
```

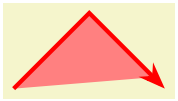
```
\usetikzlibrary {arrows.meta}
\tikz [ultra thick] \draw [red, arrows = {-Stealth[color=black]}] (0,0) -- (1,0);
```

Similar to the `color` option used in normal Ti*k*Z options, you may omit the `color=` part of the option. Whenever an ⟨*arrow key*⟩ is encountered that Ti*k*Z 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.

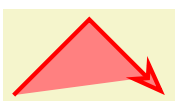
`/pgf/arrow keys/open` (no value)

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 Ti*k*Z 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).

Ti*k*Z 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 there values explicitly for an arrow tip:

`/pgf/arrow keys/line cap=`⟨*round* or *butt*⟩ (no default)

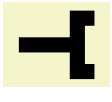
Sets the line cap of all lines that are drawn in the arrow to a round cap or a butt cap. (Unlike for normal lines, the `rect` cap is not allowed.) Naturally, this key has no effect for arrows whose paths are closed.

Each arrow tip has a default value for the line cap, which can be overruled using this option.

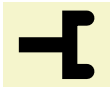
Changing the cap should have no effect on the size of the arrow. However, it will have an effect on where the exact "tip" of the arrow is since this will always be exactly at the end of the arrow:

```
\usetikzlibrary {arrows.meta}
\tikz [line width=2mm]
  \draw [arrows = {-Computer Modern Rightarrow[line cap=butt]}]
        (0,0) -- (1,0);
```

```
\usetikzlibrary {arrows.meta}
\tikz [line width=2mm]
  \draw [arrows = {-Computer Modern Rightarrow[line cap=round]}]
        (0,0) -- (1,0);
```



```
\usetikzlibrary {arrows.meta}
\tikz [line width=2mm]
  \draw [arrows = {-Bracket[reversed,line cap=butt]}]
        (0,0) -- (1,0);
```



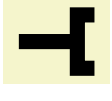
```
\usetikzlibrary {arrows.meta}
\tikz [line width=2mm]
  \draw [arrows = {-Bracket[reversed,line cap=round]}]
        (0,0) -- (1,0);
```

`/pgf/arrow keys/line join=`⟨*round* or *miter*⟩ (no default)

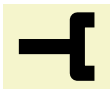
Sets the line join to round or miter (`bevel` is not allowed). This time, the key only has an effect on paths that have "corners" in them. The same rules as for `line cap` apply: the size is not affects, but the tip end is:

```
\usetikzlibrary {arrows.meta}
\tikz [line width=2mm]
  \draw [arrows = {-Computer Modern Rightarrow[line join=miter]}]
        (0,0) -- (1,0);
```

```
\usetikzlibrary {arrows.meta}
\tikz [line width=2mm]
  \draw [arrows = {-Computer Modern Rightarrow[line join=round]}]
        (0,0) -- (1,0);
```



```
\usetikzlibrary {arrows.meta}
\tikz [line width=2mm]
  \draw [arrows = {-Bracket[reversed,line join=miter]}]
        (0,0) -- (1,0);
```



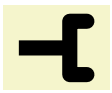
```
\usetikzlibrary {arrows.meta}
\tikz [line width=2mm]
  \draw [arrows = {-Bracket[reversed,line join=round]}]
        (0,0) -- (1,0);
```

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}
\tikz [line width=2mm]
  \draw [arrows = {-Computer Modern Rightarrow[round]}] (0,0) -- (1,0);
```

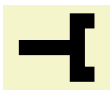


```
\usetikzlibrary {arrows.meta}
\tikz [line width=2mm]
  \draw [arrows = {-Bracket[reversed,round]}] (0,0) -- (1,0);
```

/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}
\tikz [line width=2mm]
  \draw [arrows = {-Computer Modern Rightarrow[sharp]}] (0,0) -- (1,0);
```



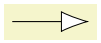
```
\usetikzlibrary {arrows.meta}
\tikz [line width=2mm]
  \draw [arrows = {-Bracket[reversed,sharp]}] (0,0) -- (1,0);
```

You can also set the width of lines used inside arrow tips:

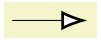
/pgf/arrow keys/line width=⟨*dimension*⟩ ⟨*line width factor*⟩ ⟨*outer factor*⟩ (no default)

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}
\tikz \draw [arrows = {-Latex[line width=0.1pt, fill=white, length=10pt]}] (0,0) -- (1,0);
```



```
\usetikzlibrary {arrows.meta}
\tikz \draw [arrows = {-Latex[line width=1pt, fill=white, length=10pt]}] (0,0) -- (1,0);
```

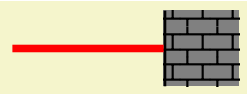
/pgf/arrow keys/line width'=⟨*dimension*⟩ ⟨*length factor*⟩ (no default)

Works like `line width` only the factor is with respect to the `length`.

16.3.8 Bending and Flexing

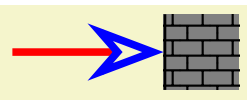
Up to now, we have only added arrow tip to the end of straight lines, which is in some sense "easy". Things get far more difficult, if the line to which we wish to end an arrow tip is curved. In the following, we have a look at the different actions that can be taken and how they can be configured.

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:

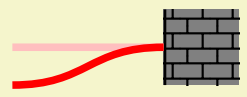
1. Notice that the red line no longer goes all the way to the wall. Indeed, the red line ends more or less exactly where it meets the blue line, leaving the arrow tip empty. Now, recall that the red line was supposed to be the path `(-2,0)--(1,0)`; however, this path has obviously become much shorter (by 6.25mm to be precise). This effect is called *path shortening* in Ti*k*Z.

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 Ti*k*Z.

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 Ti*k*Z 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):



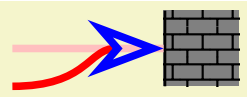
```
\begin{tikzpicture}
  \wall
  \draw [red!25,line width=1mm] (-1,0) -- (1,0);
  \draw [red,line width=1mm] (-1,-.5) .. controls (0,-.5) and (0,0) .. (1,0);
\end{tikzpicture}
```

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 Ti*k*Z 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.

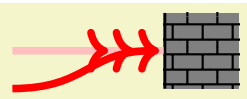
In order to add our arrow tip to the curved path, our first objective is to "shorten" the path by 6.25mm. Unfortunately, this is now much more difficult than for a straight path. When the `quick` option is added to an arrow tip (it is also the default if no special libraries are loaded), we cheat somewhat: 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}
```

They 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

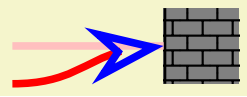
```
\usetikzlibrary{bending} % LaTeX and plain TeX
\usetikzlibrary[bending] % ConTeXt
```

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*:

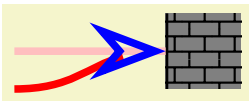


```
\usetikzlibrary {arrows.meta,bending}
\begin{tikzpicture}
  \wall
  \draw [red!25,line width=1mm] (-1,0) -- (1,0);
  \draw [red,line width=1mm,-{Stealth[length=1cm,open,blue,flex]}]
        (-1,-.5) .. controls (0,-.5) and (0,0) .. (1,0);
\end{tikzpicture}
```

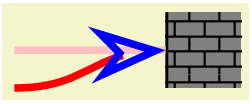
As can be seen in the example, the `flex` option gives a result that is visually pleasing and does not deform the path.

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".

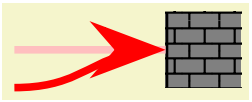


```
\usetikzlibrary {arrows.meta,bending}
\begin{tikzpicture}
  \wall
  \draw [red!25,line width=1mm] (-1,0) -- (1,0);
  \draw [red,line width=1mm,-{Stealth[length=1cm,open,blue,flex=0]}]
        (-1,-.5) .. controls (0,-.5) and (0,0) .. (1,0);
\end{tikzpicture}
```



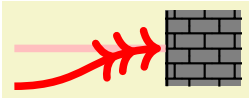
```
\usetikzlibrary {arrows.meta,bending}
\begin{tikzpicture}
  \wall
  \draw [red!25,line width=1mm] (-1,0) -- (1,0);
  \draw [red,line width=1mm,-{Stealth[length=1cm,open,blue,flex=.5]}]
        (-1,-.5) .. controls (0,-.5) and (0,0) .. (1,0);
\end{tikzpicture}
```

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}
```

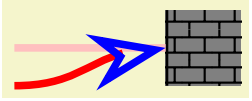
If there are several arrow tips on a path, the `flex` option positions them independently, so that each of them lies optimally on the path:



```
\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`)

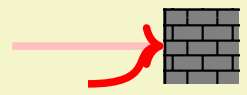
The `flex'` key is almost identical to the `flex` key. The only difference is that a factor of `1` corresponds to rotating the arrow tip so that the instead of the visual end, the "ultimate back end" of the arrow tip lies on the red path. In the example instead of having the arrow tip at a distance of `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.

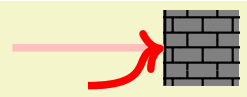
The main use of this option is not so much with an arrow tip like `Stealth` but rather with tips like the standard `>` in the context of a strongly curved line:



```
\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:

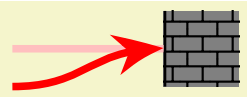


```
\usetikzlibrary {arrows.meta,bending}
\begin{tikzpicture}
  \wall
  \draw [red!25,line width=1mm] (-1,0) -- (1,0);
  \draw [red,line width=1mm,-{Computer Modern Rightarrow[flex'=.75]}]
        (0,-.5) .. controls (1,-.5) and (0.5,0) .. (1,0);
\end{tikzpicture}
```

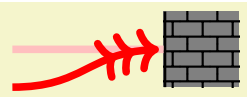
In the example, I used `flex'=.75` so as not to overpronounce the effect. Usually, you will have to fiddle with it sometime to get the "perfectly aligned arrow tip", but a value of `.75` is usually a good start.

`/pgf/arrow keys/bend` (no value)

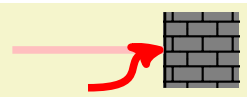
Bending an arrow tip is a radical solution to the problem of positioning arrow tips on a curved line: The arrow tip is no longer "rigid" but the drawing itself will now bend along the curve. This has the advantage that all the problems of flexing with wrong tangents and overflexing disappear. The downsides are longer computation times (bending an arrow is *much* more expensive that 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:

⟨*start specification*⟩`-`⟨*end specification*⟩

We now have a closer look at what these specifications may look like. The general syntax of the ⟨*start specification*⟩ is as follows:

`[`⟨*options for all tips*⟩`]` ⟨*first arrow tip spec*⟩ ⟨*second arrow tip spec*⟩ ⟨*third arrow tip spec*⟩ ...

As can be seen, an arrow tip specification may start with some options in brackets. If this is the case, the ⟨*options for all tips*⟩ 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 ⟨*arrow tip kind name*⟩`[`⟨*options*⟩`]`.
2. It can be of the form ⟨*shorthand*⟩`[`⟨*options*⟩`]`.
3. It can be of the form ⟨*single char shorthand*⟩`[`⟨*options*⟩`]`. 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 ⟨*arrow tip kind name*⟩ to the sequence of arrow tips with the ⟨*options*⟩ applied to it at the start (for the ⟨*start specification*⟩) or at the end (for the ⟨*end specification*⟩). Note that for the ⟨*start specification*⟩ the first arrow tip specified in this way will be at the very start of the curve, while for the ⟨*end specification*⟩ 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 ⟨*options*⟩ 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 ⟨*arrow tip kind name*⟩, 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 Ti*k*Z 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 Ti*k*Z 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 Ti*k*Z 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 `o`? 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 Ti*k*Z 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]` 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:

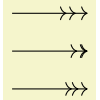

```
\tikz \draw [<<<->>>>] (0,0) -- (2,0);
```

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`)

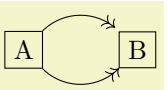
When a sequence of arrow tips is specified in an arrow tip specification for the end of the line, the arrow tips are normally arranged in such a way that the tip of each arrow ends exactly at the "back end" of the next arrow tip (for start specifications, the ordering is inverted, of course). Now, when 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]     >}] (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}
\tikz {
  \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);
}
```

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);
```

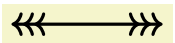


```
\tikz \draw [->__>] (0,0) -- (1,0);
```

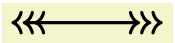
However, using the `sep` option will be faster than using the `_` arrow tip and it also allows you to specify the desired length directly.

16.4.3 Specifying the Line End

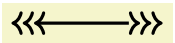
In the previous examples of sequences of arrow tips, the line of the path always ended at the last of the arrow tips (for end specifications) or at the first of the arrow tips (for start specifications). Often, this is what you may want, but not always. Fortunately, it is quite easy to specify the desired 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) are at the first arrow following `.` (for start specifications).



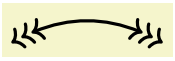
```
\tikz [very thick] \draw [<<<->>>] (0,0) -- (2,0);
```



```
\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:

```
\usetikzlibrary {arrows.meta}
\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` are 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}
\tikz [foo /.tip = {Stealth[sep]. >>}]
  \draw [-foo] (0,0) -- (2,0);
```

```
\usetikzlibrary {arrows.meta}
\tikz [foo /.tip = {Stealth[sep] Latex[sep]},
       bar /.tip = {Stealth[length=10pt,open]}]
  \draw [-{foo[red] . bar}] (0,0) -- (2,0);
```

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 behavior 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}
\tikz \draw [Stealth-Stealth] (0,0) -- (1,0);
```

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}
\tikz [<-> /.tip = Stealth] \draw [<->>] (0,0) -- (1,0);
```



```
\usetikzlibrary {arrows.meta}
\tikz [<-> /.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 Ti*k*Z (more precisely, PGF) actually uses the following definition internally:

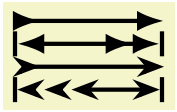
```
>-< /.tip = >[reversed]
```

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:

```
\tikz [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*
`Arc Barb[]`	thin	**thick**	
`Bar[]`	thin	**thick**	
`Bracket[]`	thin	**thick**	
`Hooks[]`	thin	**thick**	
`Parenthesis[]`	thin	**thick**	
`Straight Barb[]`	thin	**thick**	
`Tee Barb[]`	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 TeX.

Appearance of the below at line width	*0.4pt*	*0.8pt*	*1.6pt*
`Classical TikZ Rightarrow[]`	thin	**thick**	
`Computer Modern Rightarrow[]`	thin	**thick**	
`Implies[]` on double line	thin	**thick**	
`To[]`	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 ⟶.

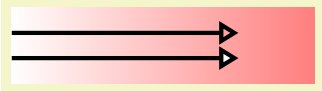
Appearance of the below at line width	*0.4pt*	*0.8pt*	*1.6pt*
`Circle[]`	thin	**thick**	
`Diamond[]`	thin	**thick**	
`Ellipse[]`	thin	**thick**	
`Kite[]`	thin	**thick**	
`Latex[]`	thin	**thick**	

`Latex[round]`	thin	**thick**	
`Rectangle[]`	thin	**thick**	
`Square[]`	thin	**thick**	
`Stealth[]`	thin	**thick**	
`Stealth[round]`	thin	**thick**	
`Triangle[]`	thin	**thick**	
`Turned Square[]`	thin	**thick**	

Here are the "open" variants:

Appearance of the below at line width	*0.4pt*	*0.8pt*	*1.6pt*
`Circle[open]`	thin	**thick**	
`Diamond[open]`	thin	**thick**	
`Ellipse[open]`	thin	**thick**	
`Kite[open]`	thin	**thick**	
`Latex[open]`	thin	**thick**	
`Latex[round,open]`	thin	**thick**	
`Rectangle[open]`	thin	**thick**	
`Square[open]`	thin	**thick**	
`Stealth[open]`	thin	**thick**	
`Stealth[round,open]`	thin	**thick**	
`Triangle[open]`	thin	**thick**	
`Turned Square[open]`	thin	**thick**	

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}
\tikz {
  \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);
}
```

- *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 ➠»».

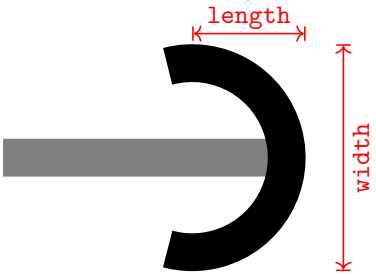
Appearance of the below at line width	*1ex*	*1em*
`Butt Cap[]`		
`Fast Round[]`		
`Fast Triangle[]`		
`Round Cap[]`		
`Triangle Cap[]`		

- *Special* arrow tips are used for some specific purpose and do not fit into the above categories.

Appearance of the below at line width	*0.4pt*	*0.8pt*	*1.6pt*
`Rays[]`	thin	**thick**	
`Rays[n=8]`	thin	**thick**	

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	*0.4pt*	*0.8pt*	*1.6pt*
`Arc Barb[]`	thin	**thick**	
`Arc Barb[sep] Arc Barb[]`	thin	**thick**	
`Arc Barb[sep] . Arc Barb[]`	thin	**thick**	
`Arc Barb[arc=120]`	thin	**thick**	
`Arc Barb[arc=270]`	thin	**thick**	
`Arc Barb[length=2pt]`	thin	**thick**	
`Arc Barb[length=2pt,width=5pt]`	thin	**thick**	
`Arc Barb[line width=2pt]`	thin	**thick**	
`Arc Barb[reversed]`	thin	**thick**	
`Arc Barb[round]`	thin	**thick**	
`Arc Barb[slant=.3]`	thin	**thick**	
`Arc Barb[left]`	thin	**thick**	
`Arc Barb[right]`	thin	**thick**	
`Arc Barb[harpoon,reversed]`	thin	**thick**	
`Arc Barb[red]`	thin	**thick**	

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 `Tee Barb` 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.

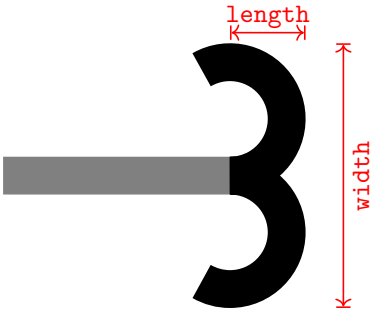
Appearance of the below at line width	*0.4pt*	*0.8pt*	*1.6pt*
`Bracket[]`	thin	**thick**	
`Bracket[sep] Bracket[]`	thin	**thick**	
`Bracket[sep] . Bracket[]`	thin	**thick**	
`Bracket[reversed]`	thin	**thick**	
`Bracket[round]`	thin	**thick**	
`Bracket[slant=.3]`	thin	**thick**	
`Bracket[left]`	thin	**thick**	
`Bracket[right]`	thin	**thick**	
`Bracket[harpoon,reversed]`	thin	**thick**	
`Bracket[red]`	thin	**thick**	

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 that 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	*0.4pt*	*0.8pt*	*1.6pt*
`Hooks[]`	thin	**thick**	
`Hooks[sep] Hooks[]`	thin	**thick**	
`Hooks[sep] . Hooks[]`	thin	**thick**	
`Hooks[arc=120]`	thin	**thick**	
`Hooks[arc=270]`	thin	**thick**	
`Hooks[length=2pt]`	thin	**thick**	
`Hooks[length=2pt,width=5pt]`	thin	**thick**	
`Hooks[line width=2pt]`	thin	**thick**	
`Hooks[reversed]`	thin	**thick**	
`Hooks[round]`	thin	**thick**	
`Hooks[slant=.3]`	thin	**thick**	
`Hooks[left]`	thin	**thick**	
`Hooks[right]`	thin	**thick**	
`Hooks[harpoon,reversed]`	thin	**thick**	
`Hooks[red]`	thin	**thick**	

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 `Arc Barb` 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.

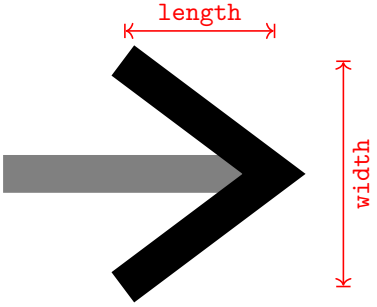
Appearance of the below at line width	*0.4pt*	*0.8pt*	*1.6pt*
`Parenthesis[]`	thin	**thick**	
`Parenthesis[sep] Parenthesis[]`	thin	**thick**	
`Parenthesis[sep] . Parenthesis[]`	thin	**thick**	
`Parenthesis[reversed]`	thin	**thick**	
`Parenthesis[round]`	thin	**thick**	
`Parenthesis[slant=.3]`	thin	**thick**	
`Parenthesis[left]`	thin	**thick**	
`Parenthesis[right]`	thin	**thick**	
`Parenthesis[harpoon,reversed]`	thin	**thick**	
`Parenthesis[red]`	thin	**thick**	

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 | *0.4pt* | *0.8pt* | *1.6pt*

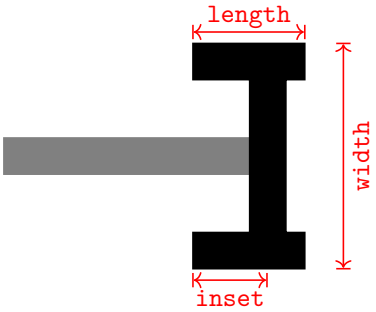
`Straight Barb[]` — thin — **thick**
`Straight Barb[] Straight Barb[]` — thin — **thick**
`Straight Barb[] . Straight Barb[]` — thin — **thick**
`Straight Barb[length=5pt]` — thin — **thick**
`Straight Barb[length=5pt,width=5pt]` — thin — **thick**
`Straight Barb[line width=2pt]` — thin — **thick**
`Straight Barb[reversed]` — thin — **thick**
`Straight Barb[angle=60:2pt 3]` — thin — **thick**
`Straight Barb[round]` — thin — **thick**
`Straight Barb[slant=.3]` — thin — **thick**
`Straight Barb[left]` — thin — **thick**
`Straight Barb[right]` — thin — **thick**
`Straight Barb[harpoon,reversed]` — thin — **thick**
`Straight Barb[red]` — thin — **thick**

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 | *0.4pt* | *0.8pt* | *1.6pt*

`Tee Barb[]` — thin — **thick**
`Tee Barb[sep] Tee Barb[]` — thin — **thick**
`Tee Barb[sep] . Tee Barb[]` — thin — **thick**
`Tee Barb[inset=0pt]` — thin — **thick**
`Tee Barb[inset'=0pt 1]` — thin — **thick**
`Tee Barb[line width=2pt]` — thin — **thick**
`Tee Barb[round]` — thin — **thick**
`Tee Barb[round,inset'=0pt 1]` — thin — **thick**
`Tee Barb[slant=.3]` — thin — **thick**
`Tee Barb[left]` — thin — **thick**
`Tee Barb[right]` — thin — **thick**
`Tee Barb[harpoon,reversed]` — thin — **thick**
`Tee Barb[red]` — thin — **thick**

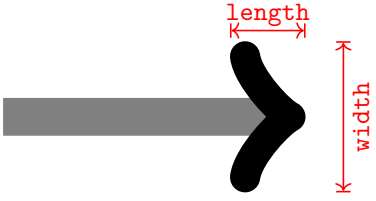
The following options have no effect: `open`, `fill`.

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 Ti*k*Z 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 →. This new version

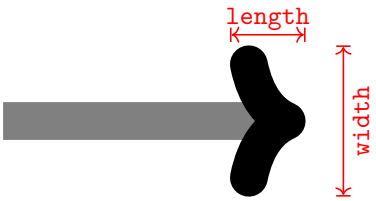
is also the default used as `>` and as `To`, now.

Appearance of the below at line width	*0.4pt*	*0.8pt*	*1.6pt*
`Classical TikZ Rightarrow[]`	thin	**thick**	
`Classical TikZ Rightarrow[sep] Classical TikZ Rightarrow[]`	thin	**thick**	
`Classical TikZ Rightarrow[sep] . Classical TikZ Rightarrow[]`	thin	**thick**	
`Classical TikZ Rightarrow[length=3pt]`	thin	**thick**	
`Classical TikZ Rightarrow[sharp]`	thin	**thick**	
`Classical TikZ Rightarrow[slant=.3]`	thin	**thick**	
`Classical TikZ Rightarrow[left]`	thin	**thick**	
`Classical TikZ Rightarrow[right]`	thin	**thick**	
`Classical TikZ Rightarrow[harpoon,reversed]`	thin	**thick**	
`Classical TikZ Rightarrow[red]`	thin	**thick**	

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" → 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 `>`).

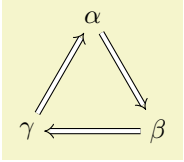
Appearance of the below at line width	*0.4pt*	*0.8pt*	*1.6pt*
`Computer Modern Rightarrow[]`	thin	**thick**	
`Computer Modern Rightarrow[sep] Computer Modern Rightarrow[]`	thin	**thick**	
`Computer Modern Rightarrow[sep] . Computer Modern Rightarrow[]`	thin	**thick**	
`Computer Modern Rightarrow[length=3pt]`	thin	**thick**	
`Computer Modern Rightarrow[sharp]`	thin	**thick**	
`Computer Modern Rightarrow[slant=.3]`	thin	**thick**	
`Computer Modern Rightarrow[left]`	thin	**thick**	
`Computer Modern Rightarrow[right]`	thin	**thick**	
`Computer Modern Rightarrow[harpoon,reversed]`	thin	**thick**	
`Computer Modern Rightarrow[red]`	thin	**thick**	

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 (⟹). A typical use of this arrow tip is




```
\usetikzlibrary {arrows.meta,graphs}
\tikz \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	*0.4pt*	*0.8pt*	*1.6pt*
`Implies[]` on double line	thin	**thick**	
`Implies[red]` on double line	thin	**thick**	

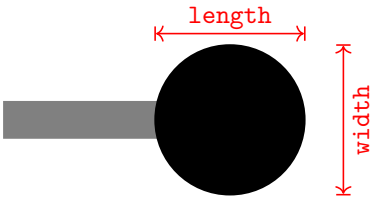
Arrow Tip Kind `To`

This is a shorthand for `Computer Modern Rightarrow` when the `arrows.meta` library is loaded. Otherwise, it is a shorthand for the classical Ti*k*Z 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	*0.4pt*	*0.8pt*	*1.6pt*
`Circle[]`	thin	**thick**	
`Circle[sep] Circle[]`	thin	**thick**	
`Circle[sep] . Circle[]`	thin	**thick**	
`Circle[open]`	thin	**thick**	
`Circle[length=3pt]`	thin	**thick**	
`Circle[slant=.3]`	thin	**thick**	
`Circle[left]`	thin	**thick**	
`Circle[right]`	thin	**thick**	
`Circle[red]`	thin	**thick**	

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	*0.4pt*	*0.8pt*	*1.6pt*
`Diamond[]`	thin	**thick**	
`Diamond[] Diamond[]`	thin	**thick**	
`Diamond[] . Diamond[]`	thin	**thick**	
`Diamond[open]`	thin	**thick**	
`Diamond[length=10pt]`	thin	**thick**	
`Diamond[round]`	thin	**thick**	
`Diamond[slant=.3]`	thin	**thick**	
`Diamond[left]`	thin	**thick**	
`Diamond[right]`	thin	**thick**	
`Diamond[red]`	thin	**thick**	
`Diamond[fill=red!50]`	thin	**thick**	

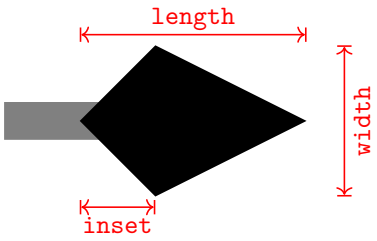
Arrow Tip Kind `Ellipse`

This is a shorthand for a "circle" that is twice as wide as high.

Appearance of the below at line width	*0.4pt*	*0.8pt*	*1.6pt*
`Ellipse[]`	thin	**thick**	
`Ellipse[sep] Ellipse[]`	thin	**thick**	
`Ellipse[sep] . Ellipse[]`	thin	**thick**	
`Ellipse[open]`	thin	**thick**	
`Ellipse[length=10pt]`	thin	**thick**	
`Ellipse[round]`	thin	**thick**	
`Ellipse[slant=.3]`	thin	**thick**	
`Ellipse[left]`	thin	**thick**	
`Ellipse[right]`	thin	**thick**	
`Ellipse[red]`	thin	**thick**	
`Ellipse[fill=red!50]`	thin	**thick**	

Arrow Tip Kind `Kite`

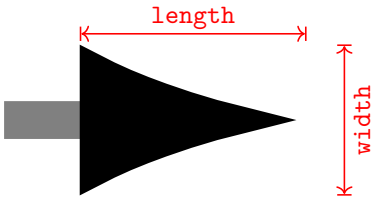
This arrow tip consists of four lines that form a "kite". The `inset` prescribed how far the width-axis of the kite is removed from the back end. Note that the inset cannot be negative, use a `Stealth` arrow tip for this.



Appearance of the below at line width	*0.4pt*	*0.8pt*	*1.6pt*
`Kite[]`	thin	**thick**	
`Kite[sep] Kite[]`	thin	**thick**	
`Kite[sep] . Kite[]`	thin	**thick**	
`Kite[open]`	thin	**thick**	
`Kite[length=6pt,width=4pt]`	thin	**thick**	
`Kite[length=6pt,width=4pt,inset=1.5pt]`	thin	**thick**	
`Kite[round]`	thin	**thick**	
`Kite[slant=.3]`	thin	**thick**	
`Kite[left]`	thin	**thick**	
`Kite[right]`	thin	**thick**	
`Kite[red]`	thin	**thick**	

Arrow Tip Kind `Latex`

This arrow tip is the same as the arrow tip used in LaTeX'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.



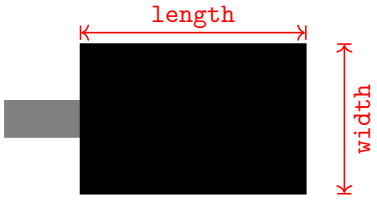
Appearance of the below at line width	*0.4pt*	*0.8pt*	*1.6pt*
`Latex[]`	thin	**thick**	
`Latex[sep] Latex[]`	thin	**thick**	
`Latex[sep] . Latex[]`	thin	**thick**	
`Latex[open]`	thin	**thick**	
`Latex[length=4pt]`	thin	**thick**	
`Latex[round]`	thin	**thick**	
`Latex[slant=.3]`	thin	**thick**	
`Latex[left]`	thin	**thick**	
`Latex[right]`	thin	**thick**	
`Latex[red]`	thin	**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	*0.4pt*	*0.8pt*	*1.6pt*
`Rectangle[]`	thin	**thick**	
`Rectangle[sep] Rectangle[]`	thin	**thick**	
`Rectangle[sep] . Rectangle[]`	thin	**thick**	
`Rectangle[open]`	thin	**thick**	
`Rectangle[length=4pt]`	thin	**thick**	
`Rectangle[round]`	thin	**thick**	

`Rectangle[slant=.3]`	thin	**thick**	
`Rectangle[left]`	thin	**thick**	
`Rectangle[right]`	thin	**thick**	
`Rectangle[red]`	thin	**thick**	

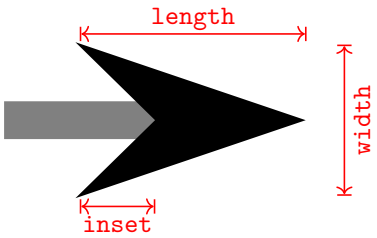
Arrow Tip Kind `Square`

An instance of the `Rectangle` whose width is identical to the length.

Appearance of the below at line width	*0.4pt*	*0.8pt*	*1.6pt*
`Square[]`	thin	**thick**	
`Square[sep] Square[]`	thin	**thick**	
`Square[sep] . Square[]`	thin	**thick**	
`Square[open]`	thin	**thick**	
`Square[length=4pt]`	thin	**thick**	
`Square[round]`	thin	**thick**	
`Square[slant=.3]`	thin	**thick**	
`Square[left]`	thin	**thick**	
`Square[right]`	thin	**thick**	
`Square[red]`	thin	**thick**	

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 is 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	*0.4pt*	*0.8pt*	*1.6pt*
`Stealth[]`	thin	**thick**	
`Stealth[sep] Stealth[]`	thin	**thick**	
`Stealth[sep] . Stealth[]`	thin	**thick**	
`Stealth[open]`	thin	**thick**	
`Stealth[length=6pt,width=4pt]`	thin	**thick**	
`Stealth[length=6pt,width=4pt,inset=1.5pt]`	thin	**thick**	
`Stealth[round]`	thin	**thick**	
`Stealth[slant=.3]`	thin	**thick**	
`Stealth[left]`	thin	**thick**	
`Stealth[right]`	thin	**thick**	
`Stealth[red]`	thin	**thick**	

Arrow Tip Kind `Triangle`

An instance of a `Kite` with zero inset.

Appearance of the below at line width	*0.4pt*	*0.8pt*	*1.6pt*
`Triangle[]`	thin	**thick**	
`Triangle[sep] Triangle[]`	thin	**thick**	
`Triangle[sep] . Triangle[]`	thin	**thick**	
`Triangle[open]`	thin	**thick**	
`Triangle[length=4pt]`	thin	**thick**	
`Triangle[angle=45:1pt 3]`	thin	**thick**	
`Triangle[angle=60:1pt 3]`	thin	**thick**	
`Triangle[angle=90:1pt 3]`	thin	**thick**	
`Triangle[round]`	thin	**thick**	
`Triangle[slant=.3]`	thin	**thick**	
`Triangle[left]`	thin	**thick**	

`Triangle[right]`	thin	**thick**	
`Triangle[red]`	thin	**thick**	

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	*0.4pt*	*0.8pt*	*1.6pt*
`Turned Square[]`	thin	**thick**	
`Turned Square[sep] Turned Square[]`	thin	**thick**	
`Turned Square[sep] . Turned Square[]`	thin	**thick**	
`Turned Square[open]`	thin	**thick**	
`Turned Square[length=4pt]`	thin	**thick**	
`Turned Square[round]`	thin	**thick**	
`Turned Square[slant=.3]`	thin	**thick**	
`Turned Square[left]`	thin	**thick**	
`Turned Square[right]`	thin	**thick**	
`Turned Square[red]`	thin	**thick**	

16.5.4 Caps

Recall that a *cap* is a way of ending a line. The graphic languages underlying Ti*k*Z (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.

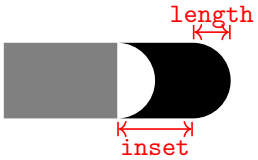
Arrow Tip Kind `Butt Cap`

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:


```
\usetikzlibrary {arrows.meta}
\tikz \draw [line width=1ex, line cap=round, -Butt Cap] (0,0) -- (1,0);
```

Arrow Tip Kind `Fast Round`

This arrow tip is not really a cap, you use it in conjunction with (typically) the `Round Cap`. The idea is that you end your line using the round cap and then add several `Fast Round`s. As for `Round Cap`, the `length` parameter dictates the length is the length of the "main part", the inset sets the length of a line that comes before this tip.




```
\usetikzlibrary {arrows.meta}
\tikz \draw [line width=1ex,
             -{Round Cap [] . Fast Round[] Fast Round[]}]
  (0,0) -- (1,0);
```

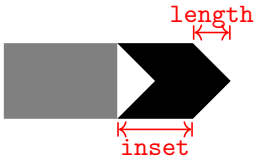
Note that in conjunction with the `bend` option, this works even quite well for curves:

```
\usetikzlibrary {arrows.meta,bending}
\tikz [f/.tip = Fast Round] % shorthand
  \draw [line width=1ex, -{[bend] Round Cap[] . f f f}]
  (0,0) to [bend left] (1,0);
```

Appearance of the below at line width	*1ex*	*1em*
`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 `Fast Round`, only for triangular caps.




```
\usetikzlibrary {arrows.meta}
\tikz \draw [line width=1ex,
             -{Triangle Cap []. Fast Triangle[] Fast Triangle[]}]
  (0,0) -- (1,0);
```

Again, this tip works well for curves:

```
\usetikzlibrary {arrows.meta,bending}
\tikz [f/.tip = Fast Triangle] % shorthand
  \draw [line width=1ex, -{[bend] Triangle Cap[] . f f f}]
  (0,0) to [bend left] (1,0);
```

Appearance of the below at line width	*1ex*	*1em*
`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	*1ex*	*1em*
`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:

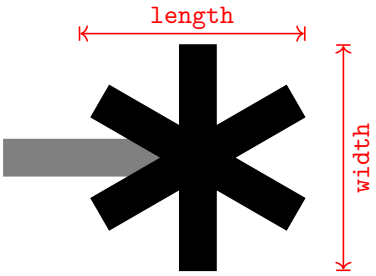
`/pgf/arrow keys/cap angle=`⟨*angle*⟩ (no default)

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	*1ex*	*1em*
`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	*0.4pt*	*0.8pt*	*1.6pt*
`Rays[]`	thin	**thick**	
`Rays[sep] Rays[]`	thin	**thick**	
`Rays[sep] . Rays[]`	thin	**thick**	
`Rays[width'=0pt 2]`	thin	**thick**	
`Rays[round]`	thin	**thick**	
`Rays[n=2]`	thin	**thick**	
`Rays[n=3]`	thin	**thick**	
`Rays[n=4]`	thin	**thick**	
`Rays[n=5]`	thin	**thick**	
`Rays[n=6]`	thin	**thick**	
`Rays[n=7]`	thin	**thick**	
`Rays[n=8]`	thin	**thick**	
`Rays[n=9]`	thin	**thick**	
`Rays[slant=.3]`	thin	**thick**	
`Rays[left]`	thin	**thick**	
`Rays[right]`	thin	**thick**	
`Rays[left,n=5]`	thin	**thick**	
`Rays[right,n=5]`	thin	**thick**	
`Rays[red]`	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 Ti*k*Z 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.

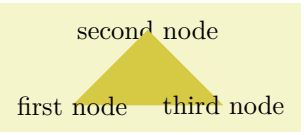
In Section 17.2 the basic syntax of the node operation is explained, followed in Section 17.3 by the syntax for multi-part nodes, which are nodes that contain several different text parts. After this, the different options for the text in nodes are explained. In Section 17.5 the concept of *anchors* 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 Ti*k*Z 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:


```
\tikz {
  \path (0,0) node [red]                    {A}
        (1,0) node [blue]                   {B}
        (2,0) node [green, node contents=C]
        (3,0) node [node contents=D]           ;
}
```

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).

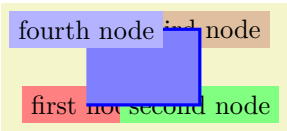
`/tikz/at=`⟨*coordinate*⟩ (no default)

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, Ti*k*Z 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).




```
\tikz \fill [fill=blue!50, draw=blue, very thick]
      (0,0)   node [behind path, fill=red!50]   {first node}
   -- (1.5,0) node [behind path, fill=green!50] {second node}
   -- (1.5,1) node [behind path, fill=brown!50] {third node}
   -- (0,1)   node [             fill=blue!30]  {fourth node};
```

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 behavior, 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 behavior.

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)

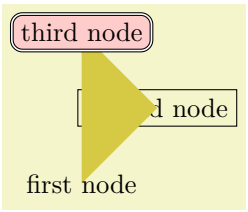
Assigns a name to the node for later reference. Since this is a "high-level" name (drivers never know of it), you can use spaces, number, letters, or whatever you like when naming a node. Thus, you can name a node just `1` or perhaps `start of chart` or even `y_1`. Your node name should *not* contain 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.

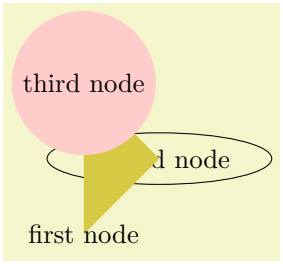
The options of a node. The ⟨*options*⟩ is an optional list of options that *apply only to the node* and have no effect outside. The other way round, most "outside" options also apply to the node, but not all. For example, the "outside" rotation does not apply to nodes (unless some special options are 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 Ti*k*Z 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 71 for details.

To select the shape of a node, the following option is used:

/tikz/shape=⟨*shape name*⟩ (no default, initially `rectangle`)

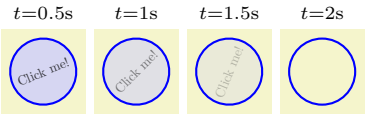
Select the shape either of the current node or, when this option is not given inside a node but somewhere outside, the shape of all nodes in the current scope.

Since this option is used often, you can leave out the `shape=`. When Ti*k*Z 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}
\tikz
  \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 backslash 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:

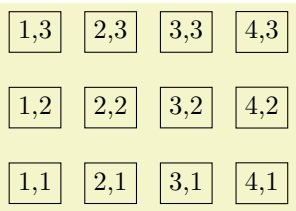
1 2 3

```
\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":



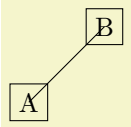

```
\tikz \node foreach \x in {1,...,4} foreach \y in {1,2,3}
            [draw] at (\x,\y) {\x,\y};
```

As the example shows, a ⟨*list*⟩ can contain ellipses (three dots) to indicate 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 88.

Styles for nodes. The following styles influence how nodes are rendered:

`/tikz/every node` (style, initially empty)

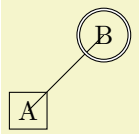
This style is installed at the beginning of every node.




```
\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

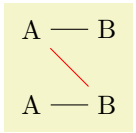
```
\begin{tikzpicture}
  [execute at begin node={A},
   execute at end node={D}]
  \node[execute at begin node={B}] {C};
\end{tikzpicture}
```

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:




```
\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);
}
```

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`.

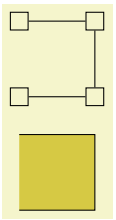
17.2.2 Predefined Shapes

PGF and Ti*k*Z define three shapes, by default:

- `rectangle`,
- `circle`, and
- `coordinate`.

By loading library packages, you can define more shapes like ellipses or diamonds; see Section 71 for the complete list of shapes.

The `coordinate` shape is handled in a special way by Ti*k*Z. 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

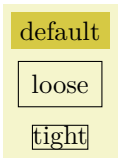
The exact behavior of shapes differs, shapes defined for more special purposes (like a, say, transistor shape) will have even more custom behaviors. However, there are some options that apply to most shapes:

`/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.

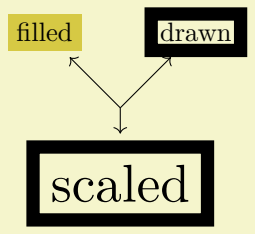
`/pgf/outer sep`=⟨*dimension or "auto"*⟩ (no default)

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).

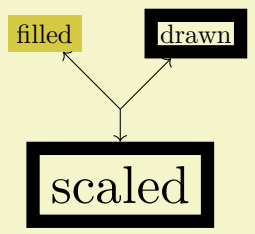



```
\begin{tikzpicture}
  \draw[line width=5pt]
    (0,0)  node[fill=yellow!80!black] (f) {filled}
    (2,0)  node[draw]                 (d) {drawn}
    (1,-2) node[draw,scale=2]         (s) {scaled};

  \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 108.2.6 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*⟩ (no default, initially `.5\pgflinewidth`)

alias `/tikz/outer xsep`

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*⟩ (no default, initially `.5\pgflinewidth`)

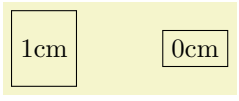
alias `/tikz/outer ysep`

Specifies the outer separation in the y-direction, only.

`/pgf/minimum height=`⟨*dimension*⟩ (no default, initially `1pt`)

alias `/tikz/minimum height`

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.

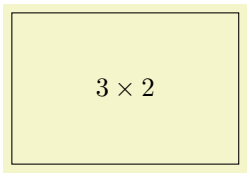



```
\begin{tikzpicture}
  \draw (0,0) node[minimum height=1cm,draw] {1cm}
        (2,0) node[minimum height=0cm,draw] {0cm};
\end{tikzpicture}
```

`/pgf/minimum width=`⟨*dimension*⟩ (no default, initially `1pt`)

alias `/tikz/minimum width`

Same as `minimum height`, only for the width.

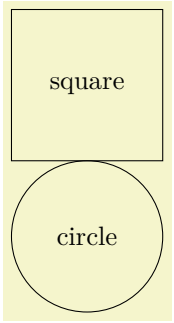



```
\begin{tikzpicture}
  \draw (0,0) node[minimum height=2cm,minimum width=3cm,draw] {$3 \times 2$};
\end{tikzpicture}
```

/pgf/minimum size=⟨*dimension*⟩ (no default)

alias /tikz/minimum size

Sets both the minimum height and width at the same time.

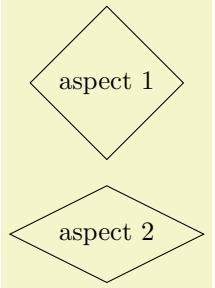



```
\begin{tikzpicture}
  \draw (0,0)  node[minimum size=2cm,draw] {square};
  \draw (0,-2) node[minimum size=2cm,draw,circle] {circle};
\end{tikzpicture}
```

/pgf/shape aspect=⟨*aspect ratio*⟩ (no default)

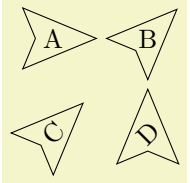
alias /tikz/shape aspect

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}
```

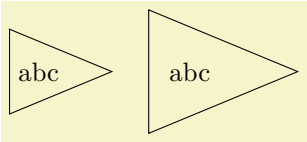
Some shapes (but not all), support a special kind of rotation. This rotation affects only the border of a shape and is independent of the node contents, but *in addition* to any other transformations.




```
\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`

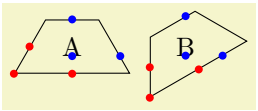
Determines if the border of a shape is constructed using the incircle. If no value is given ⟨*boolean*⟩ will take the default value `true`.

`/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 Ti*k*Z support such multipart nodes. On the lower level, PGF provides a system for specifying that a shape consists of several parts. On the Ti*k*Z 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 LaTeX. 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 `\nodepart` 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} $00$};
```

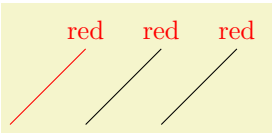
17.4 The Node Text

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

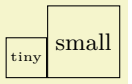
Next, you may wish to adjust the font used for the text. Naturally, you can just use a font command like `\small` or `\rm` at the beginning of a node. However, the following two options make it easier to set the font used in nodes on a general basis. Let us start with:

`/tikz/node font=`⟨*font commands*⟩ (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 ⟨*font commands*⟩ 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 ⟨*font commands*⟩:




```
\tikz \node [node font=\tiny,  minimum height=3em, draw] {tiny};
\tikz \node [node font=\small, minimum height=3em, draw] {small};
```

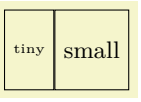
The other font command is:

/tikz/font=⟨*font commands*⟩ (no default)

Sets the font used for the text inside nodes. However, this font will *not* (yet) 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).

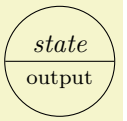
italic

```
\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

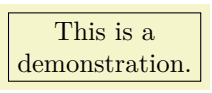
```
\tikz \node [draw] {
  \begin{tabular}{cc}
    upper left & upper right\\
    lower left & lower right
  \end{tabular}
};
```

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 `\\` inside your node to mark the end of lines and then request Ti*k*Z to arrange these lines in some manner. This will only be done, however, if the `align` option has been given.

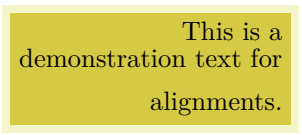
This is a
demonstration.

```
\tikz[align=left] \node[draw] {This is a\\demonstration.};
```




```
\tikz[align=center] \node[draw] {This is a\\demonstration.};
```

The \\ command takes an optional extra space as an argument in square brackets.




```
\tikz \node[fill=yellow!80!black,align=right]
  {This is a\\[-2pt] demonstration text for\\[1ex] alignments.};
```

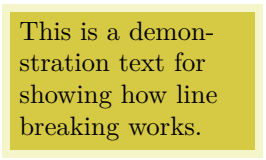
3. You can request that Ti*k*Z does an automatic line-breaking for you inside the node by specifying a fixed `text width` for the node. In this case, you can still use \\ to enforce a line-break. Note that when you specify a text width, the node will have this width, independently of whether the text 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 line `\centering`.




```
\tikz \draw (0,0) node[fill=yellow!80!black,text width=3cm]
  {This is a demonstration text for showing how line breaking works.};
```

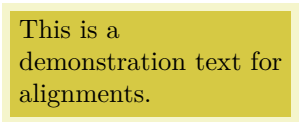
Setting ⟨*dimension*⟩ to an empty string causes the automatic line breaking to be disabled.

`/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.




```
\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 LATEX-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, Ti*k*Z 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 all 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)

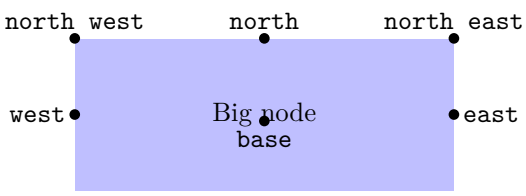
This option works like `text height`, only for the depth of the text box. This option is mostly useful when you need to ensure a uniform depth of text boxes that need to be aligned.

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.

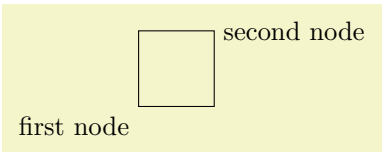
17.5.1 Positioning Nodes Using Anchors

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 to 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};
```

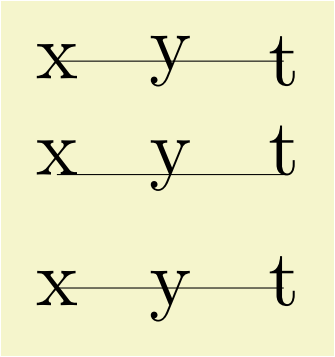
Since the default anchor is `center`, the default behavior is to shift the node in such a way that it is centered on the current position.

`/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}
```

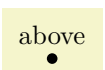
17.5.2 Basic Placement Options

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*⟩.


```
\tikz \fill (0,0) circle (2pt) node[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] % ConTeXt
```

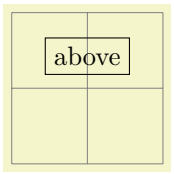
The library defines additional options for placing nodes conveniently. It also redefines the standard options like above so that they give you better control of node placement.

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:

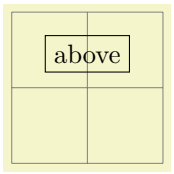
1. It can simply be a ⟨*dimension*⟩ (or a mathematical expression that evaluates to a dimension) like 2cm or 3cm/2+4cm. In this case, the following happens: the node's anchor is set to south and the node is vertically shifted upwards by ⟨*dimension*⟩.




```
\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.

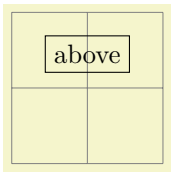
2. It can be a ⟨*number*⟩ (that is, any mathematical expression that does not include a unit like `pt` or `cm`). Examples are `2` or `3+sin(60)`. In this case, the anchor is also set to `south` and the node is vertically shifted by the vertical component of the coordinate `(0,`⟨*number*⟩`)`.




```
\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`.

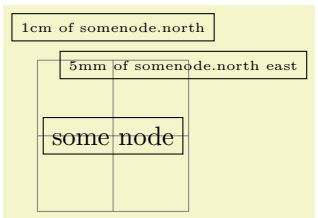



```
\usetikzlibrary {positioning}
\begin{tikzpicture}
  \draw[help lines] (0,0) grid (2,2);
  \node at (1,1) [above=.2 and 3mm,draw] {above};
  % south border of the node is also 2mm above (1,1)
\end{tikzpicture}
```

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=5mm of somenode.north east] {\tiny 5mm of somenode.north east};
  \node [above=1cm of somenode.north]      {\tiny 1cm of somenode.north};
\end{tikzpicture}
```

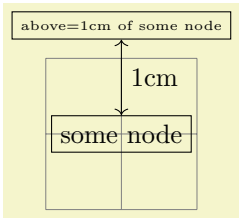
   As can be seen the `above=5mm of somenode.north east` option does, indeed, place the node 5mm above the north east anchor of `somenode`. The same effect could have been achieved writing `above=5mm` followed by `at=(somenode.north east)`.

   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.



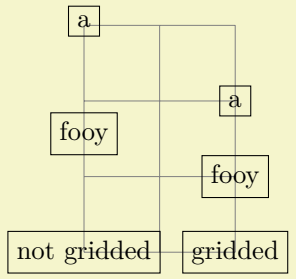

```
\usetikzlibrary {positioning}
\begin{tikzpicture}[every node/.style=draw]
  \draw[help lines] (0,0) grid (2,2);
  \node (some node) at (1,1) {some node};

  \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 behavior 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`.

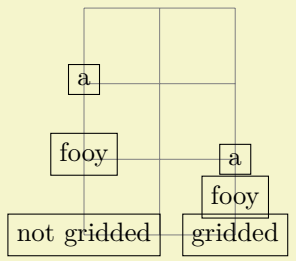
This has the following effect: When you say `above=1cm of somenode` with `on grid` set to true, the new node will be placed in such a way that its center is 1cm above the center of `somenode`. Repeatedly placing nodes in this way will result in nodes that are centered on "grid coordinate", hence the 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)

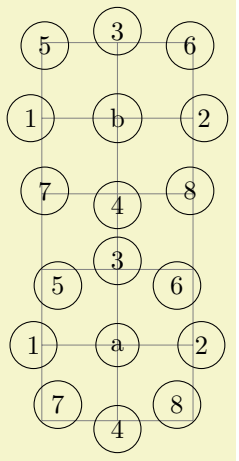
This key is redefined in the same manner as `left`.

`/tikz/above left=`⟨*specification*⟩ (no default)

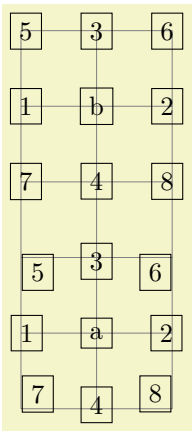
This key is also redefined in a manner similar to the above, but behavior of the ⟨*shifting part*⟩ is more complicated:

1. 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 be 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.)
2. 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.

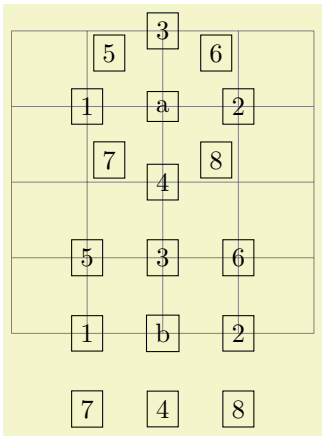
The following example should help to illustrate the difference:




```
\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)

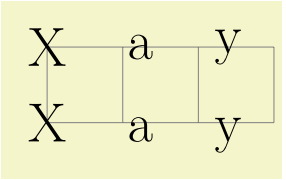
Works similar to `above left`.

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`.

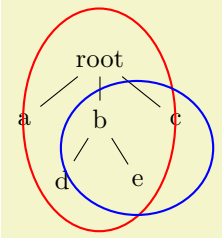
17.5.4 Advanced Arrangements of Nodes

The simple `above` and `right` options may not always suffice for arranging a large number of nodes. For such situations Ti*k*Z offers libraries that make positioning easier: The `matrix` library and the `graphdrawing` library. These libraries for positioning nodes are described in two separate Sections 20 and 27.

17.6 Fitting Nodes to a Set of Coordinates

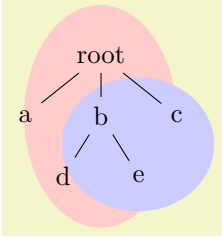
It is sometimes desirable that the size and position of a node is not given using anchors and size parameters, rather one would sometimes have a box be placed and be sized such that it "is just large enough to contain this, that, and that point". This situation typically arises when a picture has been 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 54 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:




```
\usetikzlibrary {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} };

  \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.

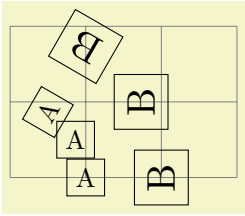
However, sometimes you *do* wish to transform a node, for example, it certainly sometimes makes sense to rotate a node by 90 degrees. There are two ways to achieve this:

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.

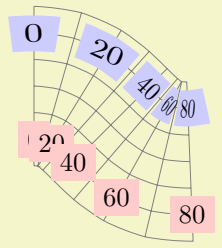



```
\usepgfmodule {nonlineartransformations}\usetikzlibrary {curvilinear}
\begin{tikzpicture}[every node/.style={draw}]
  \draw[help lines](0,0) grid (3,2);
  \draw            (1,0) node{A}
                   (2,0) node[rotate=90,scale=1.5] {B};
  \draw[rotate=30] (1,0) node{A}
                   (2,0) 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 Ti*k*Z currently does not allow you to configure so-called *nonlinear transformations,* see Section 108.4, 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, Ti*k*Z 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 108.4.




```
\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

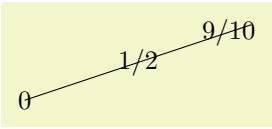
Until now, we always placed nodes on a coordinate that is mentioned in the path. Often, however, we wish to place nodes on "the middle" of a line and we do not wish to compute these coordinates "by hand". To facilitate such placements, Ti*k*Z allows you to specify that a certain node should be somewhere "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.

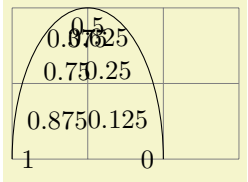
Now, what is "the previous line"? This depends on the previous path construction operation.

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};
```

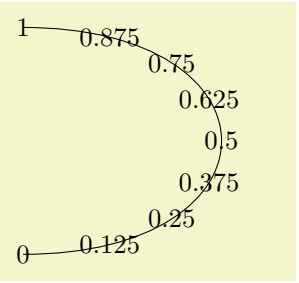
For the `arc` operation, the position is simply the corresponding position on the arc:




```
\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};
}
```

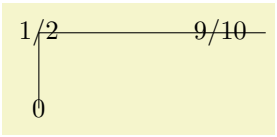
The next case is the curve-to operation (the `..` operation). In this case, the "middle" of the curve, that is, the position `0.5` is not necessarily the point at the exact half distance on the line. Rather, it is some

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.

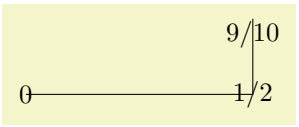



```
\tikz \draw (0,0) .. controls +(right:3.5cm) and +(right:3.5cm) .. (0,3)
  node foreach \p in {0,0.125,...,1} [pos=\p]{\p};
```

Another interesting case are the horizontal/vertical line-to operations `|-` and `-|`. For them, the position (or time) `0.5` is exactly the corner point.




```
\tikz \draw (0,0) |- (3,1)
  node[pos=0]{0} node[pos=0.5]{1/2} node[pos=0.9]{9/10};
```



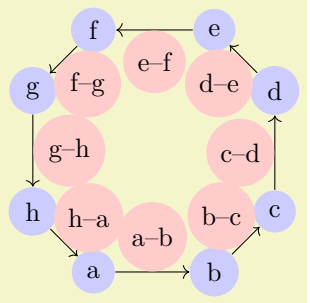

```
\tikz \draw (0,0) -| (3,1)
  node[pos=0]{0} node[pos=0.5]{1/2} node[pos=0.9]{9/10};
```

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 position to be calculated automatically according to the following rule. Consider a line between two 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.

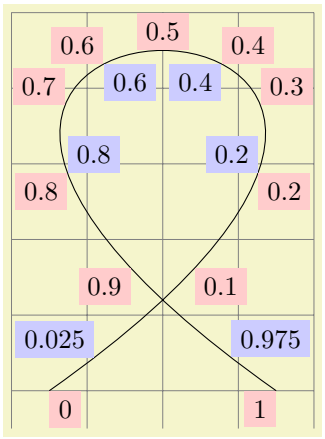
This option only has an effect for nodes that are placed on lines or curves.



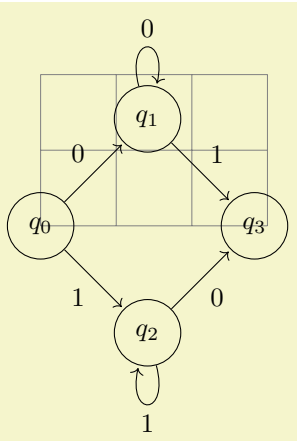

```
\begin{tikzpicture}
  [scale=.8,auto=left,every node/.style={circle,fill=blue!20}]
  \node (a) at (-1,-2) {a};
  \node (b) at ( 1,-2) {b};
  \node (c) at ( 2,-1) {c};
  \node (d) at ( 2, 1) {d};
  \node (e) at ( 1, 2) {e};
  \node (f) at (-1, 2) {f};
  \node (g) at (-2, 1) {g};
  \node (h) at (-2,-1) {h};

  \foreach \from/\to in {a/b,b/c,c/d,d/e,e/f,f/g,g/h,h/a}
    \draw [->] (\from) -- (\to)
               node[midway,fill=red!20] {\from--\to};
\end{tikzpicture}
```

`/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}
```



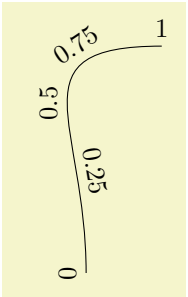

```
\usetikzlibrary {automata}
\begin{tikzpicture}[shorten >=1pt,node distance=2cm,auto]
  \draw[help lines] (0,0) grid (3,2);

  \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 [swap] {1} (q_2)
            (q_1) edge              node        {1} (q_3)
                  edge [loop above] node        {0} ()
            (q_2) edge              node [swap] {0} (q_3)
                  edge [loop below] node        {1} ();
\end{tikzpicture}
```

/tikz/' (no value)

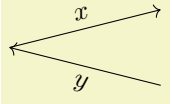
This is a very short alias for `swap`.

/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.



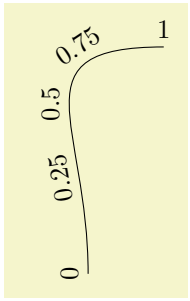

```
\tikz \draw (0,0) .. controls +(up:2cm) and +(left:2cm) .. (1,3)
    node foreach \p in {0,0.25,...,1} [sloped,above,pos=\p]{\p};
```



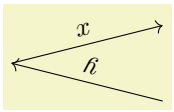

```
\begin{tikzpicture}[->]
  \draw (0,0)   -- (2,0.5) node[midway,sloped,above] {$x$};
  \draw (2,-.5) -- (0,0)   node[midway,sloped,below] {$y$};
\end{tikzpicture}
```

/tikz/allow upside down=⟨*boolean*⟩ (default `true`, `initially false`)

If set to `true`, Ti*k*Z will not "righten" upside down text.




```
\tikz [allow upside down]
  \draw (0,0) .. controls +(up:2cm) and +(left:2cm) .. (1,3)
    node foreach \p in {0,0.25,...,1} [sloped,above,pos=\p]{\p};
```

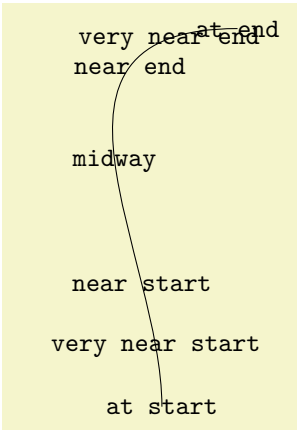



```
\begin{tikzpicture}[->,allow upside down]
  \draw (0,0)   -- (2,0.5) node[midway,sloped,above] {$x$};
  \draw (2,-.5) -- (0,0)   node[midway,sloped,below] {$y$};
\end{tikzpicture}
```

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) .. (1,5)
      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`.

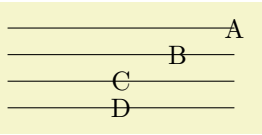
17.9 Placing Nodes on a Line or Curve Implicitly

When you wish to place a node on the line `(0,0) -- (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) -- 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 `-- node`⟨*node specification*⟩⟨*coordinate*⟩. (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 ⟨*coordinate*⟩ 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.

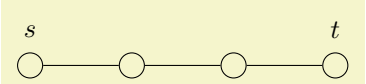
17.10 The Label and Pin Options

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.

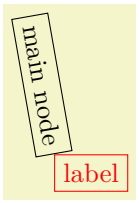
   The ⟨*angle*⟩ determines the position on the border of the shape in two different ways. Normally, the border position is given by `main node.`⟨*angle*⟩. This means that the ⟨*angle*⟩ can either be a number like `0` or `-340`, but it can also be an anchor like `north`. Additionally, the special angles `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:

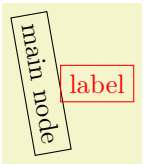
   /tikz/absolute=⟨*true or false*⟩ (default `true`)

   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:



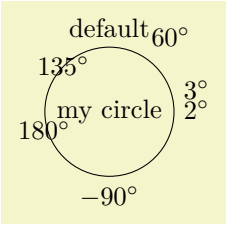

```
\tikz [rotate=-80,every label/.style={draw,red}]
  \node [transform shape,rectangle,draw,label=right:label] {main node};
```




```
\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`.

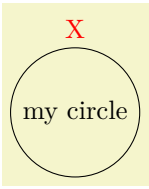



```
\tikz
  \node [circle, draw,
         label=default,
         label=60:$60^\circ$,
         label=below:$-90^\circ$,
         label=3:$3^\circ$,
         label=2:$2^\circ$,
         label={[below]180:$180^\circ$},
         label={[centered]135:$135^\circ$}] {my circle};
```

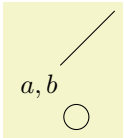
3. One ⟨*angle*⟩ is special: If you set the ⟨*angle*⟩ to `center`, then the label will be placed on the center of the main node. This is mainly useful for adding a label text to an existing node, especially if it has been rotated.


```
\tikz \node [transform shape,rotate=90,
             rectangle,draw,label={[red]center:R}] {main node};
```

You can pass ⟨*options*⟩ to the node `label node`. For this, you provide the options in square brackets before the ⟨*angle*⟩. 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};
```



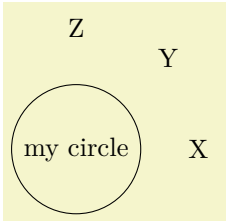

```
\begin{tikzpicture}
  \node [circle,draw,label={[name=label node]above left:$a,b$}] {};
  \draw (label node) -- +(1,1);
\end{tikzpicture}
```

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 ⟨*distance*⟩ 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)

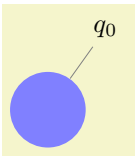
This style is used in every node created by the `label` option. The default is `draw=none,fill=none`.

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:

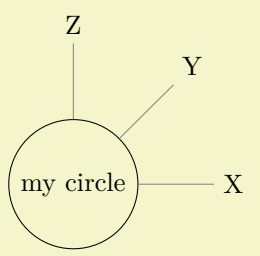



```
\tikz \node [circle,fill=blue!50,minimum size=1cm,pin=60:$q_0$] {};
```

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`)

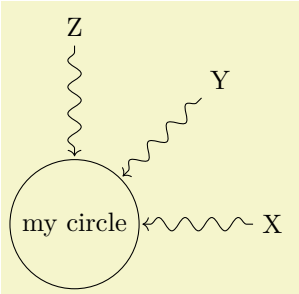
This style is used in every node created by the `pin` option.

`/tikz/pin position=`⟨*angle*⟩ (no default, initially `above`)

The default pin position. Works like `label position`.

`/tikz/every pin edge` (style, initially `help lines`)

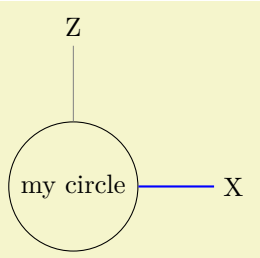
This style is used in every edge created by the `pin` options.



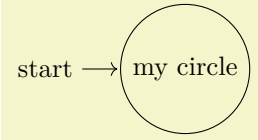

```
\usetikzlibrary {decorations.pathmorphing}
\tikz [pin distance=15mm,
       every pin edge/.style={<-,shorten <=1pt,decorate,
                              decoration={snake,pre length=4pt}}]
  \node [circle,draw,pin=right:X,
                     pin=above right:Y,
                     pin=above:Z]       {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 ⟨*key*⟩=⟨*value*⟩ pairs, you may also provide strings of the following form (the actual syntax is slightly more general, see the detailed descriptions later on):

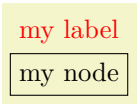
"⟨*text*⟩"⟨*options*⟩

The ⟨*options*⟩ must be surrounded in curly braces when they contain a comma, otherwise the curly braces are optional. The ⟨*options*⟩ may be preceded by an optional space.

When a ⟨*string*⟩ of the above form is encountered inside the options of a `node`, then it is internally transformed to

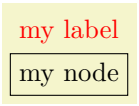
`label={[`⟨*options*⟩`]`⟨*text*⟩`}`

Let us have a look at an example:



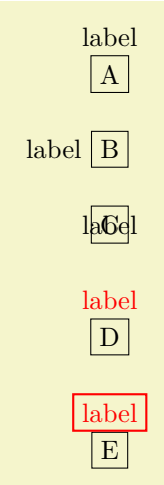

```
\usetikzlibrary {quotes}
\tikz \node ["my label" red, draw] {my node};
```

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 ⟨*options*⟩ 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 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 Ti*k*Z do not start with quotation marks). When this happens, the ⟨*string*⟩ should not be a key–value pair, but, rather, must have the form:

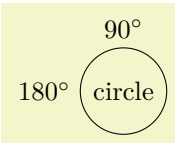
"⟨*text*⟩"'⟨*options*⟩

(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

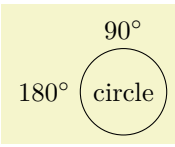
`label={[`⟨*options*⟩`]`⟨*text*⟩`}`

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 ["$90^\circ$" above, "$180^\circ$" left, circle, draw] {circle};
```

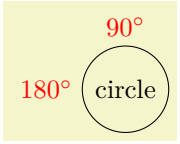
Alternatively, you can also use ⟨*direction*⟩`:`⟨*actual text*⟩ as your ⟨*text*⟩. This works since the `label` command allows you to specify a direction at the beginning when it is separated by a colon:




```
\usetikzlibrary {quotes}
\tikz
  \node ["90:$90^\circ$", "left:$180^\circ$", circle, draw] {circle};
```

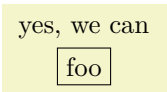
Arguably, placing `above` or `left` behind the ⟨*text*⟩ seems more natural than having it inside the ⟨*text*⟩. In addition to the above, before the ⟨*options*⟩ are executed, the following style is also executed:

`/tikz/every label quotes` (style, no value)



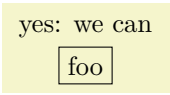

```
\usetikzlibrary {quotes}
\tikz [every label quotes/.style=red]
  \node ["90:$90^\circ$", "left:$180^\circ$", circle, draw] {circle};
```

Handling commas and colons inside the text. The ⟨*text*⟩ 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 ⟨*text*⟩:




```
\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:



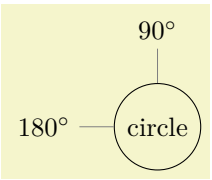

```
\usetikzlibrary {quotes}
\tikz \node ["yes{:} we can", draw] {foo};
```

The optional apostrophe. Following the closing quotation marks in a ⟨*string*⟩ 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 ⟨*options*⟩ as another option (and, indeed, a single apostrophe is a legal option in Ti*k*Z, 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 ["$90^\circ$" above, "$180^\circ$" 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)

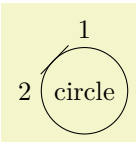
This key allows you to define your own handler for quotes options. Inside the options of a `node`, whenever a key–value pair with the syntax

"⟨*text*⟩"'⟨*options*⟩

is encountered, the following happens: The above string gets replaced by ⟨*replacement*⟩ where inside the ⟨*replacement*⟩ the parameter `#1` is ⟨*text*⟩ and `#2` is ⟨*options*⟩. If the apostrophe is present (see also the discussion of `quotes mean label`), the ⟨*options*⟩ start with `',`.

The ⟨*replacement*⟩ 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}
\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`.

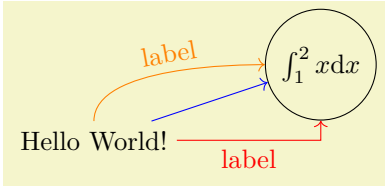
1. Once the node `x` has been defined, you can use `(x.`⟨*anchor*⟩`)` wherever you would normally use a normal coordinate. This will yield the position at which the given ⟨*anchor*⟩ is in the picture. Note that transformations do not apply to this coordinate, that is, `(x.north)` will be the northern 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)`.

   If the specified coordinate is almost identical to the node center, for example `(x)--(0,0)`, no line will be drawn and a warning message will be printed.

   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) {$\int_1^2 x \mathrm d x$};

  \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*⟩`)` … `;`

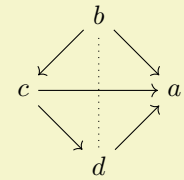
The effect of the `edge` operation is that after the main path the following path is added to the picture:

`\path[every edge,`⟨*options*⟩`] (\tikztostart)` ⟨*path*⟩`;`

Here, ⟨*path*⟩ is the `to path`. Note that, unlike the path added by the `to` operation, the `(\tikztostart)` is added before the ⟨*path*⟩ (which is unnecessary for the `to` operation, since this coordinate is already part of the main path).

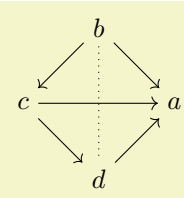
The `\tikztostart` is the last coordinate on the path just before the `edge` operation, just as for the `node` or `to` operations. However, there is one exception to this rule: If the `edge` operation is directly preceded by a `node` operation, then this just-declared node is the start coordinate (and 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:



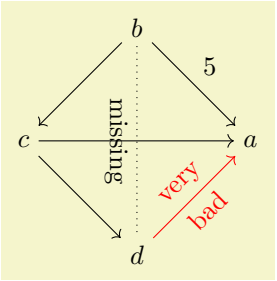

```
\begin{tikzpicture}
  \node (a) at   (0:1) {$a$};
  \node (b) at  (90:1) {$b$} edge [->]     (a);
  \node (c) at (180:1) {$c$} edge [->]     (a)
                             edge [<-]     (b);
  \node (d) at (270:1) {$d$} edge [->]     (a)
                             edge [dotted] (b)
                             edge [<-]     (c);
\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.

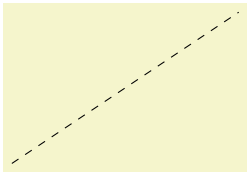



```
\begin{tikzpicture}
  \node foreach \name/\angle in {a/0,b/90,c/180,d/270}
        (\name) at (\angle:1.5) {$\name$};

  \path[->] (b) edge            node[above right]  {$5$}      (a)
                edge                                          (c)
                edge [-,dotted] node[below,sloped] {missing} (d)
            (c) edge                                          (a)
                edge                                          (d)
            (d) edge [red]      node[above,sloped] {very}
                                node[below,sloped] {bad}      (a);
\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,`⟨*options*⟩`]{`⟨*text*⟩`}`

As described in Section 17.10.4, the apostrophe becomes part of the ⟨*options*⟩, when present.

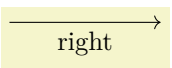
The following style is important for the placement of the labels:

`/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.

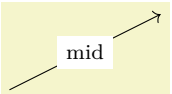

```
\usetikzlibrary {quotes}
\tikz \draw (0,0) edge ["left", ->] (2,0);
```

In order to place all labels to the right by default, change this style to `auto=right`:



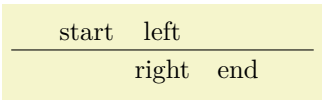

```
\usetikzlibrary {quotes}
\tikz [every edge quotes/.style={auto=right}]
  \draw (0,0) edge ["right", ->] (2,0);
```

To place all nodes "on" the edge, just make this style empty (and, possibly, make your labels opaque):



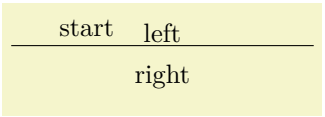

```
\usetikzlibrary {quotes}
\tikz [every edge quotes/.style={fill=white,font=\footnotesize}]
  \draw (0,0) edge ["mid", ->] (2,1);
```

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 Ti*k*Z 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 is 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:




```
\usetikzlibrary {quotes}
\tikz [tight/.style={inner sep=1pt}, loose/.style={inner sep=.7em}]
  \draw (0,0) edge ["left"   tight,
                    "right"' loose,
                    "start"  near start] (4,0);
```

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 TEX twice.

(For more details on what is going on behind the scenes, see Section 106.3.2.)

Let us have a look at the effect of these options.

`/tikz/remember picture`=⟨*boolean*⟩ (no default, initially `false`)

This option tells Ti*k*Z 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 TEX.

Provided that remembering works, you may consider saying

```
\tikzset{every picture/.append style={remember picture}}
```

to make Ti*k*Z 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, Ti*k*Z will attempt to make the current picture large enough to encompass *the node in the other picture*. However, on a second run of TEX 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) {};
```

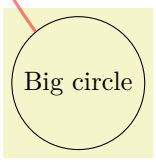
which yields , and

```
\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

This is an absolutely positioned text in the lower left corner. No shipout-hackery is used.

"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:

`\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 a 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:

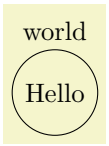
`\tikzlastnode`

Expands to the last node on the path.

Instead of the `node also` syntax, you can also use 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

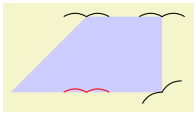
18.1 Overview

A "pic" is a "short picture" (hence the short name…) that can be inserted anywhere in Ti*k*Z 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 Ti*k*Z options for the current scope (which is the whole document); you could put `seagull/.pic=...` anywhere else where Ti*k*Z 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 see, 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 Ti*k*Z 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 Ti*k*Z 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.

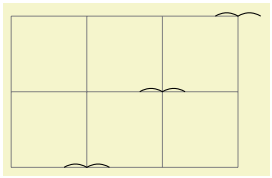
Order of the parts of the specification. Just like for nodes, everything between "pic" and the opening brace of the ⟨*pic type*⟩ is optional and can be given in any order. If there are ⟨*foreach statements*⟩, they must come first, directly following "pic". As for nodes, the "end" of the pic specification 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};
}
```

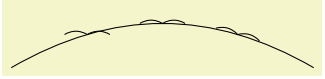
As for nodes, except for the described shifting, the coordinate system of a pic is reset prior to executing the pic type's code. This can be changed using the `transform shape` option, which has the same effect as for nodes: The "outer" transformation gets applied to the node:


```
\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:

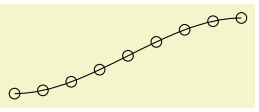
`/tikz/pics/code=`⟨*code*⟩ (no default)

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}`, Ti*k*Z 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, 0.125, ..., 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:

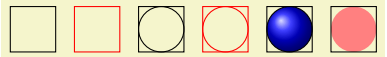
`/tikz/pic actions` (no value)

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);
  }
}
```

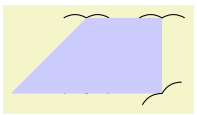
In the code, whether or not the circle gets drawn/filled/shaded depends on which options where given to the `pic` command when it is used. In contrast, the rectangle will always (just) be drawn.



```
\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};
```

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:

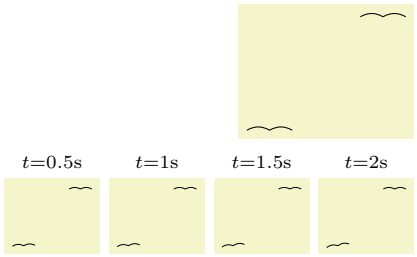
```
\tikz {
  \pic (Emma)              {seagull};
  \pic (Alexandra) at (0,1) {seagull};

  \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)

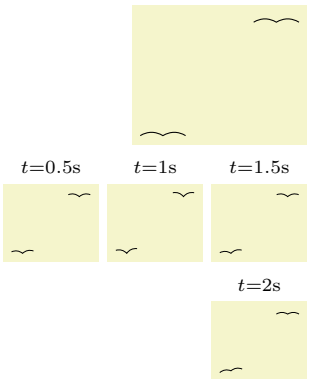
This key is available only inside the code of a pic. There, it (locally) changes the name prefix to the value it had outside the pic. This allows you to access nodes outside the current pic.

Animations for pics. Just as for nodes, you can use the attribute–colon syntax to add an animation to a pic:




```
\usetikzlibrary {animations}
\tikz {
  \pic :rotate={0s="0", 20s="90"} {seagull};
  \pic at (1.5,1.5)               {seagull};
}
```

Naturally, you can also use animations in the code of a picture:




```
\usetikzlibrary {animations}
\begin{tikzpicture} [flapping seagull/.pic={
    \draw (0,0) :path={
      0s= {"{(180:3mm) to [bend left] (0,0) to [bend left] (0:3mm)}"=base},
      1s=  "{(160:3mm) to [bend left] (0,0) to [bend left] (20:3mm)}",
      2s=  "{(180:3mm) to [bend left] (0,0) to [bend left] (0:3mm)}",
      repeats };
  }]
  \pic :rotate={0s="0", 20s="90"} {flapping seagull};
  \pic at (1.5,1.5)               {flapping seagull};
\end{tikzpicture}
```

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.

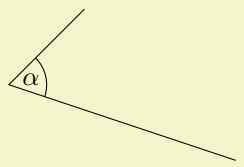
In detail, when the `quotes` library is loaded, each time a key–value pair in a list of options passed to an `pic` starts with `"`, the key–value pair must actually be a string of the following form:

"⟨*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}
\tikz \draw (3,0) coordinate (A)
         -- (0,1) coordinate (B)
         -- (1,2) coordinate (C)
            pic [draw, "$\alpha$"] {angle};
```

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 (...) ... ;
  }
}
```

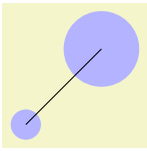
Key handler ⟨*key*⟩`/.pic=`⟨*some code*⟩

This handler can only be used with a key with the prefix `/tikz/`, so you 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

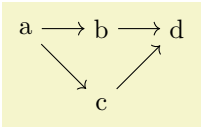
Ti*k*Z offers a powerful path command for specifying how the nodes in a graph are connected by edges and arcs: The `graph` path command, which becomes available when you load the `graphs` library.

Ti*k*Z Library graphs

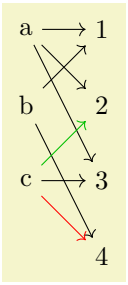
```
\usetikzlibrary{graphs} % LaTeX and plain TeX
\usetikzlibrary[graphs] % ConTeXt
```

The package must be loaded to use the `graph` path command.

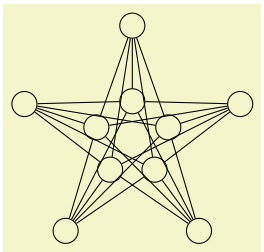
In this section, by *graph* we refer to a set of nodes together with some edges (sometimes also called arcs, in case they are directed) such as the following:



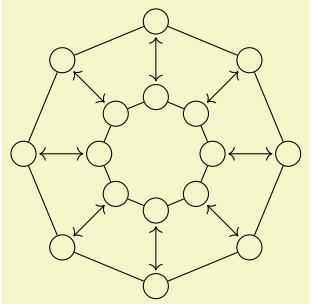

```
\usetikzlibrary {graphs}
\tikz \graph { a -> {b, c} -> d };
```



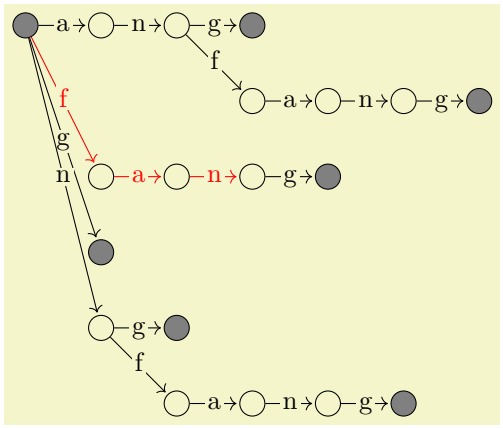

```
\usetikzlibrary {graphs.standard}
\tikz \graph {
  subgraph I_nm [V={a, b, c}, W={1,...,4}];

  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 I_n [name=outer]
  };
```

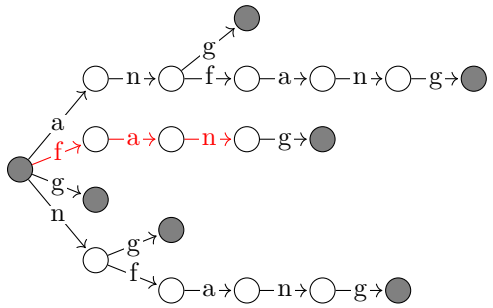


```
\usetikzlibrary {graphs}
\tikz
  \graph [nodes={draw, circle}, clockwise, radius=.75cm, empty nodes, n=8] {
    subgraph C_n [name=inner] <->[shorten <=1pt, shorten >=1pt]
    subgraph C_n [name=outer]
  };
```




```
\usetikzlibrary {graphs}
\tikz [>={To[sep]}, rotate=90, xscale=-1,
       mark/.style={fill=black!50}, mark/.default=]
  \graph [trie, simple,
          nodes={circle,draw},
          edges={nodes={
              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 Ti*k*Z 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. As an example, consider the above drawing of a trie, which is drawn without using the graph drawing libraries. Its layout can be somewhat improved by loading the `layered` graph drawing library, saying `\tikz[layered layout,...`, and then using LuaTEX, resulting in the following drawing of the same graph:



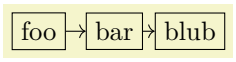
The `graphs` library uses a syntax that is quite different from the normal Ti*k*Z syntax for specifying nodes. The reason for this is that for many medium-sized graphs it can become quite cumbersome to specify all the nodes using `\node` repeatedly and then using a great number of `edge` command; possibly 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

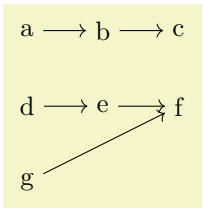
The basic way of specifying a graph is to write down a *node chain* as in the following example:




```
\usetikzlibrary {graphs}
\tikz [every node/.style = draw]
  \graph { foo -> bar -> blub };
```

As can be seen, the text `foo -> bar -> blub` 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.

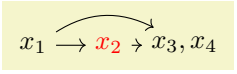
Inside a graph there can be more than one chain:




```
\usetikzlibrary {graphs}
\tikz \graph {
  a -> b -> c;
  d -> e -> f;
  g -> f;
};
```

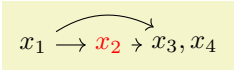
Multiple chains are separated by a semicolon or a comma (both have exactly the same effect). As the example shows, when a node text is seen for the second time, instead of creating a new node, a connection is created to the already existing node.

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}
\tikz \graph {
  x1/$x_1$ -> x2 [as=$x_2$, red] -> x34/{$x_3,x_4$};
  x1 -> [bend left] x34;
};
```

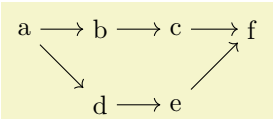
When you wish to use a node name that contains special symbols like commas or dashes, you must surround the node name by quotes. This allows you to use quite arbitrary text as a "node name":




```
\usetikzlibrary {graphs}
\tikz \graph {
  "$x_1$" -> "$x_2$"[red] -> "$x_3,x_4$";
  "$x_1$" ->[bend left] "$x_3,x_4$";
};
```

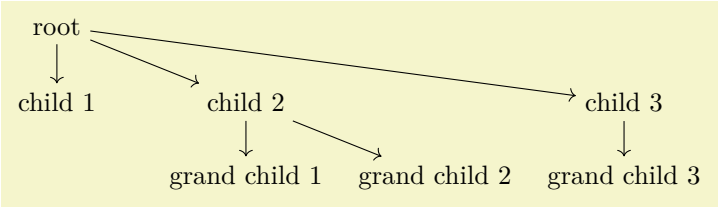
19.2.2 Concept: Chain Groups

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:



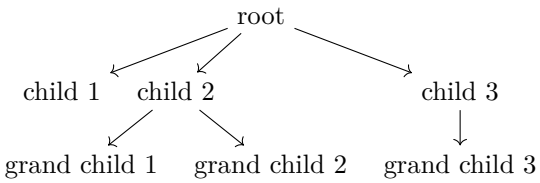

```
\usetikzlibrary {graphs}
\tikz \graph {
  a -> {
    b -> c,
    d -> e
  } -> f
};
```

Chain groups make it easy to create tree structures:



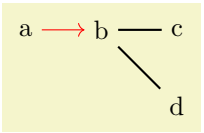

```
\usetikzlibrary {graphs}
\tikz
  \graph [grow down,
          branch right=2.5cm] {
  root -> {
    child 1,
    child 2 -> {
      grand child 1,
      grand child 2
    },
    child 3 -> {
      grand child 3
    }
  }
};
```

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 (and `\usetikzlibrary{graphdrawing}` as well as `\usegdlibrary{trees}` to the preamble) results in the following somewhat more pleasing rendering:



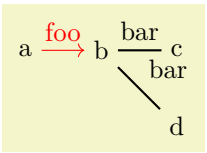
19.2.3 Concept: Edge Labels and Styles

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:



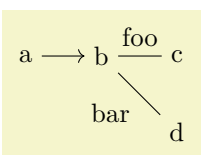

```
\usetikzlibrary {graphs}
\tikz \graph {
  a ->[red] b --[thick] {c, d};
};
```

Using the quotes syntax, see Section 17.10.4, you can even add labels to the edges easily by putting the labels in quotes:



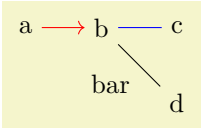

```
\usetikzlibrary {graphs,quotes}
\tikz \graph {
  a ->[red, "foo"] b --[thick, "bar"] {c, d};
};
```

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 behavior, 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 nodes. This is achieved by preceding the options with a greater-than sign:



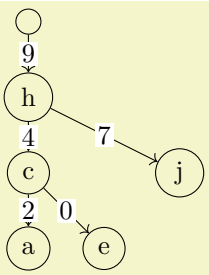

```
\usetikzlibrary {graphs,quotes}
\tikz \graph {
  a -> b -- {c [> "foo"], d [> "bar"']};
};
```

Symmetrically, preceding the options by `<` causes the options and labels to apply to the "outgoing" edges of the node:




```
\usetikzlibrary {graphs,quotes}
\tikz \graph {
  a [< red] -> b -- {c [> blue], d [> "bar"']};
};
```

This syntax allows you to easily create trees with special edge labels as in the following example of a treap:

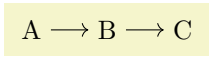



```
\usetikzlibrary {graphs,quotes}
\tikz
  \graph [edge quotes={fill=white,inner sep=1pt},
          grow down, branch right, nodes={circle,draw}] {
    "" -> h [>"9"] -> {
      c [>"4"] -> {
        a [>"2"],
        e [>"0"]
      },
      j [>"7"]
    }
  };
```

19.2.4 Concept: Node Sets

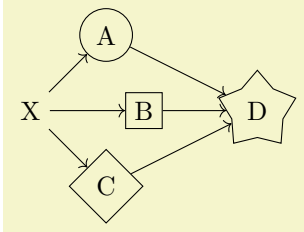
When you write down some node text inside a `graph` command, a new node is created by default unless this node has already been created inside the same `graph` command. In particular, if a node has already been declared outside of the current `graph` command, a new node of the same name gets created.

This is not always the desired behavior. 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}
\tikz {
  \node (a) at (0,0) {A};
  \node (b) at (1,0) {B};
  \node (c) at (2,0) {C};

  \graph { (a) -> (b) -> (c) };
}
```

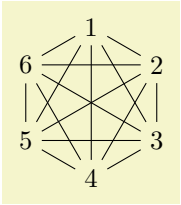
You can even go a step further: A whole collection of nodes can all be flagged to belong to a *node set* by adding the option `set=`⟨*node set name*⟩. Then, inside a `graph` command, you can collectively refer to these nodes by surrounding the node set name in parentheses:



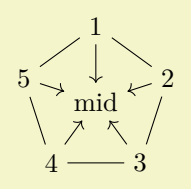

```
\usetikzlibrary {graphs,shapes.geometric}
\tikz [new set=my nodes] {
  \node [set=my nodes, circle,    draw] at (1,1)   {A};
  \node [set=my nodes, rectangle, draw] at (1.5,0) {B};
  \node [set=my nodes, diamond,   draw] at (1,-1)  {C};
  \node (d)            [star,     draw] at (3,0)   {D};

  \graph { X -> (my nodes) -> (d) };
}
```

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}
\tikz \graph { subgraph K_n [n=6, clockwise] };
```



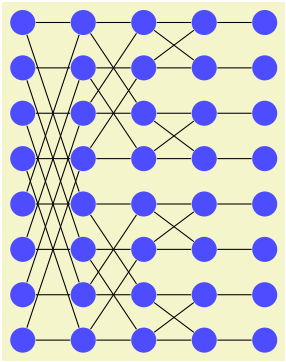

```
\usetikzlibrary {graphs.standard}
\tikz \graph { subgraph C_n [n=5, clockwise] -> mid };
```

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 in explained in detail in Section 19.3.2. Basically, a group specification is a list of chain specifications, separated by commas or semicolons.

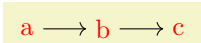
Depending on the content of the ⟨*group specification*⟩, two things will happen:

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*⟩.

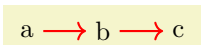



```
\usetikzlibrary {graphs}
\tikz \graph [nodes=red] { a -> b -> c };
```

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}
\tikz \graph [edges={red,thick}] { a -> b -> c };
```

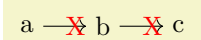
Again, multiple uses of this key accumulate.

`/tikz/graphs/edge=`⟨*options*⟩ (no default)

This is an alias for `edges`.

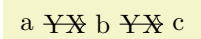
`/tikz/graphs/edge node=`⟨*node specification*⟩ (no default)

This key specifies that the ⟨*node specification*⟩ should be added to each newly created edge as an implicitly placed node.




```
\usetikzlibrary {graphs}
\tikz \graph [edge node={node [red, near end] {X}}] { a -> b -> c };
```

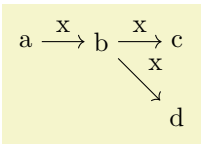
Again, multiple uses of this key accumulate.




```
\usetikzlibrary {graphs}
\tikz \graph [edge node={node [near end] {X}},
              edge node={node [near start] {Y}}] { a -> b -> c };
```

/tikz/graphs/edge label=⟨*text*⟩ (no default)

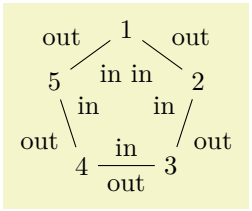
This key is an abbreviation for `edge node=node[auto]{`⟨*text*⟩`}`. The net effect is that the `text` is placed next to the newly created edges.




```
\usetikzlibrary {graphs}
\tikz \graph [edge label=x] { a -> b -> {c,d} };
```

/tikz/graphs/edge label'=⟨*text*⟩ (no default)

This key is an abbreviation for `edge node=node[auto,swap]{`⟨*text*⟩`}`.




```
\usetikzlibrary {graphs.standard}
\tikz \graph [edge label=out, edge label'=in]
  { subgraph C_n [clockwise, n=5] };
```

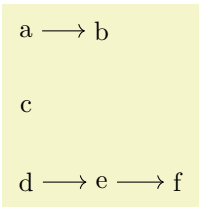
19.3.2 Syntax of Group Specifications

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.

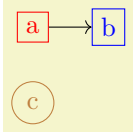
In the following example, the group specification consists of three chain specifications, namely of `a -> b`, then `c` alone, and finally `d -> e -> f`:




```
\usetikzlibrary {graphs}
\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}`.

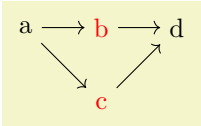
Commas are used to detect where chain specifications end. However, you will often wish to use a comma also inside the options of a single node like in the following example:




```
\usetikzlibrary {graphs}
\tikz \graph {
  a [red, draw] -> b [blue, draw],
  c [brown, draw, circle]
};
```

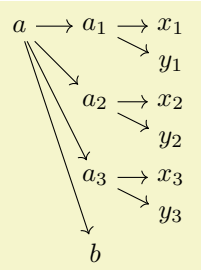
Note that the above example works as expected: The first comma inside the option list of `a` is *not* interpreted as the end of the chain specification "`a [red`". Rather, commas inside square brackets are "protected" against being interpreted as separators of group specifications.

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}
\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.

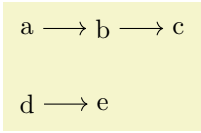



```
\usetikzlibrary {graphs}
\tikz \graph [math nodes, branch down=5mm] {
  a -> {
    \foreach \i in {1,2,3} {
      a_\i -> { x_\i, y_\i }
    },
    b
  }
};
```

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.

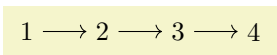
`/tikz/graph/parse=`⟨*text*⟩ (no default)

This key can only be used inside the ⟨*options*⟩ of a ⟨*group specification*⟩. Its effect is that the ⟨*text*⟩ 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 ⟨*text*⟩ 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 ⟨*chain specification*⟩ has the following syntax: It consists of a sequence of ⟨*node specifications*⟩, where subsequent node specifications are separated by ⟨*edge specifications*⟩. 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 ⟨*edge specification*⟩ 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 ⟨*left nodes*⟩ are to the left of the ⟨*edge specification*⟩ and ⟨*right nodes*⟩ are to the right and suppose we have written `->` between them. Then the following happens:

1. The ⟨*options*⟩ 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 ⟨*options*⟩ 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 ⟨*left nodes*⟩ an edge should lead to which of the ⟨*right nodes*⟩. Suppose that (l_1, r_1), …, (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 `->`, 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 `->` 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 Ti*k*Z'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);
   ```

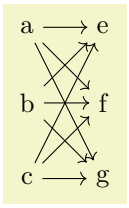
   You are welcome to change the code underlying the key.

   `/tikz/every new ->` (style, no value)

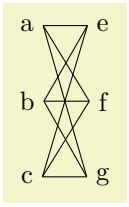
   This key gets executed by default for a `new ->`.

`/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 `->` 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 `->` is replaced by `<-`

`/tikz/every new <->` (style, no value)

`/tikz/graphs/new <-=`{⟨*left node*⟩}{⟨*right node*⟩}{⟨*edge options*⟩}{⟨*edge nodes*⟩} (no default)

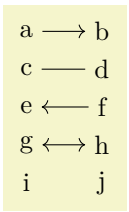
Called for `<-` with the same parameters as above.[4]

`/tikz/every new <-` (style, no value)

`/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}
\tikz \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:

[4] You might wonder why this key is needed: It seems more logical at first sight to just call `new edge 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.

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

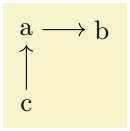
⟨*group specification*⟩

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}
\tikz \graph {
  a -> b; % both are fresh
  c -> a; % only c is fresh, a is referenced
};
```

This behavior of deciding whether a node is fresh or referenced can, however, be modified by using the following keys:

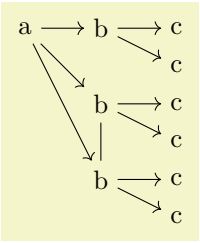
`/tikz/graphs/use existing nodes`=⟨*true or false*⟩ (default `true`)

When this key is set to `true`, all nodes will be considered to the 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 is follows: An apostrophe (`'`) is appended repeatedly until a node name is found that has not yet been used:

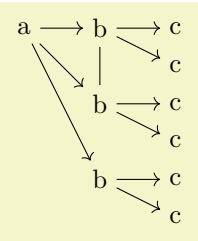



```
\usetikzlibrary {graphs}
\tikz \graph [branch down=5mm] {
  { [fresh nodes]
    a -> {
      b -> {c, c},
      b -> {c, c},
      b -> {c, c},
    }
  },
  b' -- b''
};
```

/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:

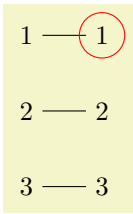
- The ⟨*full node name*⟩ is normally the ⟨*node name*⟩ that has been determined as described before. However, there are two exceptions:

  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":



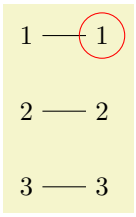

```
\usetikzlibrary {graphs}
\begin{tikzpicture}
  \graph {
    { [name=first]  1, 2, 3} --
    { [name=second] 1, 2, 3}
  };
  \draw [red] (second 1) circle [radius=3mm];
\end{tikzpicture}
```

  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.

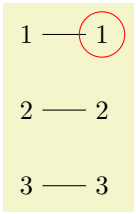
  This key can be used repeatedly, leading to ever longer node names.

  /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

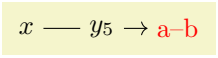
  1. The options that have accumulated in calls to `nodes` from the surrounding scopes.
  2. The local ⟨*options*⟩.

  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.


  ```
  \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*⟩ of 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:

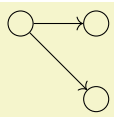
$a_{1,\dots,n}$ ↓ $b_{2,\dots,m}$ ↓ $c_{4,\dots,nm}$

```
\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)

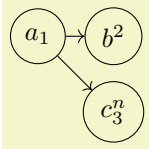
Just sets `typeset` to nothing, which causes all nodes to have an empty text (unless, of course, the `as` option is used):



```
\usetikzlibrary {graphs}
\tikz \graph [empty nodes, nodes={circle, draw}] { a -> {b, c} };
```

`/tikz/graphs/math nodes` (no value)

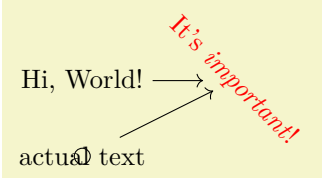
Sets `typeset` to `$\tikzgraphnodetext$`, which causes all nodes names to be typeset in math mode:




```
\usetikzlibrary {graphs}
\tikz \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 gather, 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)

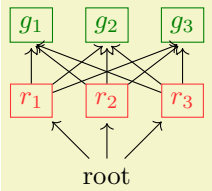
This will setup a node set named ⟨*set name*⟩ within the current scope. Inside the scope, you can add nodes to the node set using the `set` key. If a node set of the same name already exists in the current scope, it will be reset and made empty for the current scope.

Note that this command has the path `/tikz` and is normally used *outside* the `graph` command.

`/tikz/set=`⟨*set name*⟩ (no default)

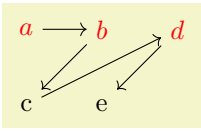
This key can be used as an option with a `node` command. The ⟨*set name*⟩ must be the name of a node set that has previously been created inside some enclosing scope via the `new set` key. The effect is that the current node is added to the node set.

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.



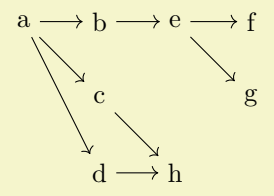

```
\usetikzlibrary {graphs}
\begin{tikzpicture}[new set=red, new set=green, shorten >=2pt]
  \foreach \i in {1,2,3} {
    \node [draw, red!80,          set=red]   (r\i) at (\i,1) {$r_\i$};
    \node [draw, green!50!black, set=green] (g\i) at (\i,2) {$g_\i$};
  }
  \graph {
    root [xshift=2cm] ->
    (red)             -> [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}
\tikz \graph { a -> {b,c,d} -> {e -> {f,g}, 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.

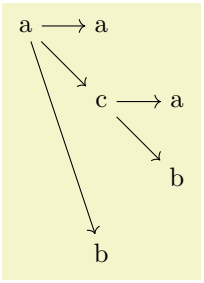
The `trie` key is useful for drawing tries, but it can also be used in other situations. What it does, essentially, is to prepend the node names of all nodes *before* the current node of the current chain to the node's name. This will often make it easier or more natural to specify graphs in which several 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

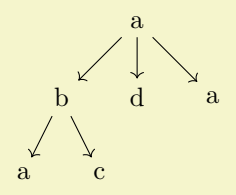
1. two nodes of the same name but in different parts of a chain will be different,
2. while if another chain starts with the same nodes, no new nodes get created.

In total, this is exactly the behavior you would expect of a trie:



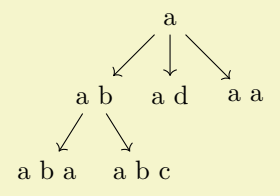

```
\usetikzlibrary {graphs}
\tikz \graph [trie] {
  a -> {
    a,
    c -> {a, b},
    b
  }
};
```

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.



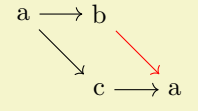

```
\usetikzlibrary {graphs,graphdrawing}\usegdlibrary {layered}
\tikz \graph [trie, simple, layered layout] {
  a -> b -> a,
  a -> b -> c,
  a -> {d,a}
};
```

In the following example, we setup the `typeset` key so that it shows the complete names of the nodes:




```
\usetikzlibrary {graphs,graphdrawing}\usegdlibrary {layered}
\tikz \graph [trie, simple, layered layout,
              typeset=\tikzgraphnodefullname] {
  a -> b -> a,
  a -> b -> c,
  a -> {d,a}
};
```

You can also use the `trie` key locally and later reference nodes using their full name:




```
\usetikzlibrary {graphs}
\tikz \graph {
  { [trie, simple]
    a -> {
      b,
      c -> a
    }
  },
  a b ->[red] a 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, the 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

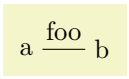
   ⟨*node spec*⟩ ⟨*connector*⟩ ⟨*node spec*⟩ ⟨*connector*⟩ …⟨*connector*⟩ ⟨*node spec*⟩`;`

   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

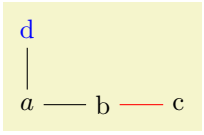
   `{ [`⟨*options*⟩`]` ⟨*chains and groups*⟩ `};`

   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:




```
\usetikzlibrary {graphs,quotes}
\tikz \graph [quick] { "a" --["foo"] "b"[x=1] };
```




```
\usetikzlibrary {graphs}
\tikz \graph [quick] {
  "a"/"$a$" -- "b"[x=1] --[red] "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:

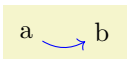

```
\usetikzlibrary {graphs}
\tikz \graph [multi] { % "multi" is not really necessary here
  a ->[bend left,  red]  b;
  a ->[bend right, blue] b;
};
```

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)

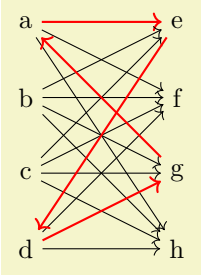
In contrast a multi-graph, in a simple graph, at most one edge gets created for every pair of vertices:




```
\usetikzlibrary {graphs}
\tikz \graph [simple]{
  a ->[bend left,  red]  b;
  a ->[bend right, blue] b;
};
```

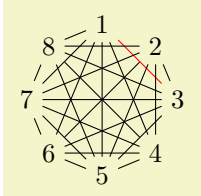
As can be seen, the second edge "wins" over the first edge. The general rule is as follows: In a simple graph, whenever an edge between two vertices is specified multiple times, only the very last specification and its options will actually be executed.

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}
\tikz \graph [simple, grow right=2cm] {
  {a,b,c,d} ->[complete bipartite] {e,f,g,h};

  { [edges={red,thick}] a -> e -> d -> g -> a };
};
```

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}
\tikz \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 -> b` and later also `a <- b`, then only the edge `a <- b` will be created. Similarly, when you say `a -> b` and later `b -> a`, then only the edge `b -> a` will be created.

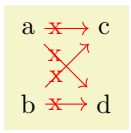
The power of the `simple` command comes at a certain cost: As the graph is being constructed, a (sparse) array is created that keeps track for each edge of the last edge being specified. Then, at the end of the scope containing the `simple` command, for every pair of vertices the edge is created. This 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 Ti*k*Z) 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 Ti*k*Z's point of view, almost the same as styling edges, since they are also specified using options.

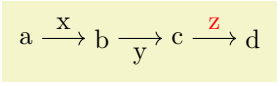
19.6.1 Options For All Edges Between Two Groups

When you write `... ->[options] ...` somewhere inside your graph specification, this typically cause one or more edges to be created between the nodes in the chain group before the `->` 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}
\tikz
  \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:

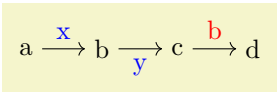



```
\usetikzlibrary {graphs,quotes}
\tikz \graph {
  a ->["x"] b ->["y"'] c ->["z" red] d;
};
```

The following options make it easy to setup the styling of nodes created in this way:

`/tikz/graphs/edge quotes=`⟨*options*⟩ (no default)

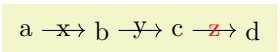
A shorthand for setting the style `every edge quotes` to ⟨*options*⟩.




```
\usetikzlibrary {graphs,quotes}
  \tikz \graph [edge quotes={blue,auto}] {
  a ->["x"] b ->["y"'] c ->["b" red] d;
};
```

`/tikz/graphs/edge quotes center` (no value)

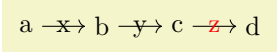
A shorthand for `edge quotes` to `anchor=center`.




```
\usetikzlibrary {graphs,quotes}
\tikz \graph [edge quotes center] {
  a ->["x"] b ->["y"] c ->["z" red] d;
};
```

`/tikz/graphs/edge quotes mid` (no value)

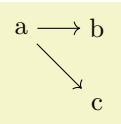
A shorthand for `edge quotes` to `anchor=mid`.




```
\usetikzlibrary {graphs,quotes}
\tikz \graph [edge quotes mid] {
  a ->["x"] b ->["y"] c ->["z" red] d;
};
```

19.6.2 Changing Options For Certain Edges

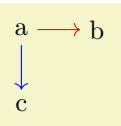
Consider the following tree-like graph:




```
\usetikzlibrary {graphs}
\tikz \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.

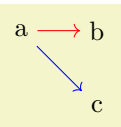
There are several ways to solve this problem. First, we can simply split up the specification and specify the two edges separately:




```
\usetikzlibrary {graphs}
\tikz \graph {
  a -> [red]  b;
  a -> [blue] c;
};
```

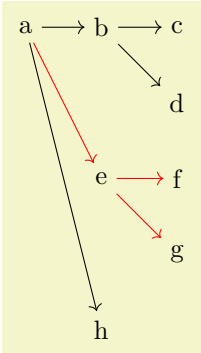
While this works quite well, we can no longer use the nice chain group syntax of the `graphs` library. For the rather simple graph `a->{b,c}` this is not a big problem, but if you specify a tree with, say, 30 nodes it is really worthwhile being able to specify the tree "in its natural form in the TEX 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}
\tikz \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}
\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}
  }
};
```

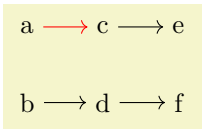
19.6.3 Options For Incoming and Outgoing Edges

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)

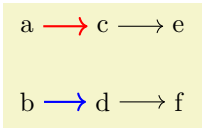
This key can (only) be used with a *node* inside a graph specification. When used, the ⟨*options*⟩ will be added to every edge that is created by a connector like `->` in which the node is a *target*. Consider the following example:




```
\usetikzlibrary {graphs}
\tikz \graph {
  { a, b } ->
  { c [target edge style=red], d } ->
  { e, f }
};
```

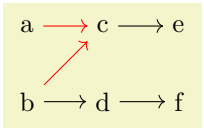
In the example, only when the edge from `a` to `c` is created, `c` is the "target" of the edge. Thus, only this edge becomes red.

When an edge already has options set directly, the ⟨*options*⟩ are executed after these direct options, thus, they "overrule" them:




```
\usetikzlibrary {graphs}
\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:

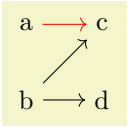



```
\usetikzlibrary {graphs}
\tikz \graph {
  { a, b } ->
  { c [target edge style=red], d } ->
  { e, f },
  b -> c
};
```

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)

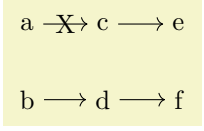
Clears all ⟨*options*⟩ for edges with the node as a target and also edge labels (see below) for this node.




```
\usetikzlibrary {graphs}
\tikz \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}
\tikz \graph {
  { a, b } ->
  { c [target edge node=node{X}], d } ->
  { e, f }
};
```