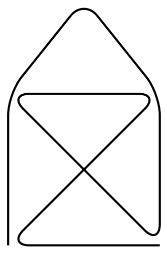
Part I
Tutorials and Guidelines

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

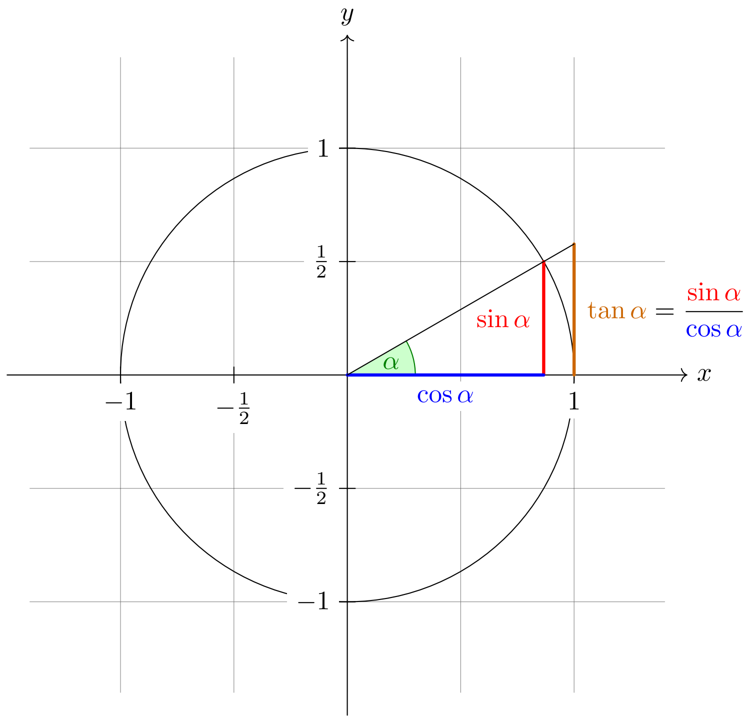
This tutorial is intended for new users of Ti*k*Z. It does not give an exhaustive account of all the features of Ti*k*Z, just of those that you are likely to use right away.

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

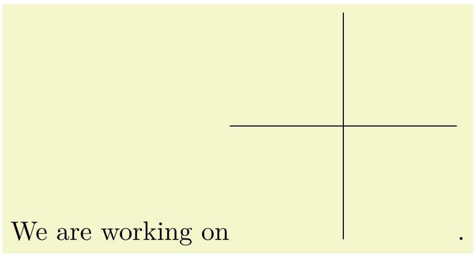
In Ti*k*Z, to draw a picture, at the start of the picture you need to tell TeX or LaTeX that you want to start a picture. In LaTeX this is done using the environment `{tikzpicture}`, in plain TeX you just use `\tikzpicture` to start the picture and `\endtikzpicture` to end it.

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`.

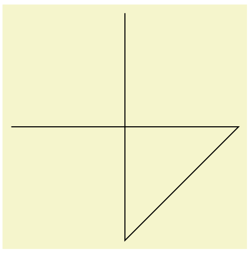
His version of the example looks like this:

```
%% 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:



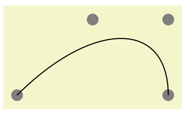
```
\tikz \draw (-1.5,0) -- (1.5,0) -- (0,-1.5) -- (0,1.5);
```

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.

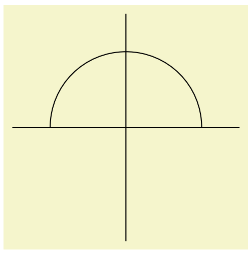
Here is an example (the control points have been added for clarity):



```
\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:

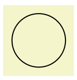


```
\begin{tikzpicture}
  \draw (-1.5,0) -- (1.5,0);
  \draw (0,-1.5) -- (0,1.5);
  \draw (-1,0) .. controls (-1,0.555) and (-0.555,1) .. (0,1)
               .. controls (0.555,1) and (1,0.555) .. (1,0);
\end{tikzpicture}
```

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

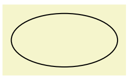
2.5 Circle Path Construction

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];
```

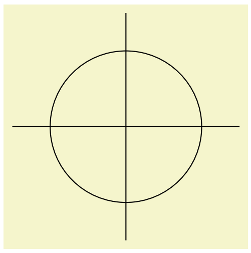
You can also append an ellipse to the path using the `ellipse` operation. Instead of a single radius you can specify two of them:



```
\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:

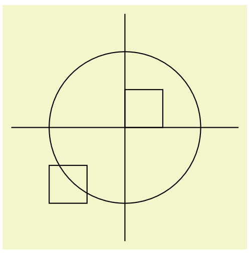


```
\begin{tikzpicture}
  \draw (-1.5,0) -- (1.5,0);
  \draw (0,-1.5) -- (0,1.5);
  \draw (0,0) circle [radius=1cm];
\end{tikzpicture}
```

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}
```

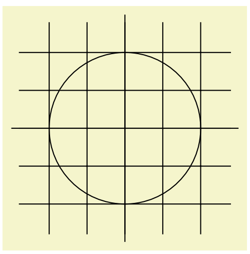
While this may be nice in other situations, this is not really leading anywhere with Karl's problem: First, we would need an awful lot of these rectangles and then there is the border that is not "closed".

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

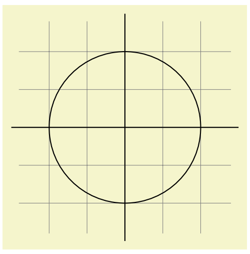
The `grid` path operation adds a grid to the current path. It will add lines making up a grid that fills the rectangle whose one corner is the current point and whose other corner is the point following the `grid` operation. For example, the code `\tikz \draw[step=2pt] (0,0) grid (10pt,10pt);` produces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

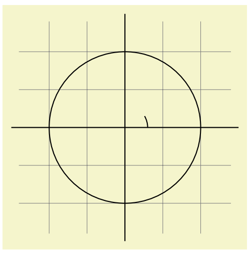
Karl wonders what other options there are that influence how a path is drawn. He saw already that the `color=`⟨*color*⟩ option can be used to set the line's color. The option `draw=`⟨*color*⟩ does nearly the same, only it sets the color for the lines only and a different color can be used for filling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

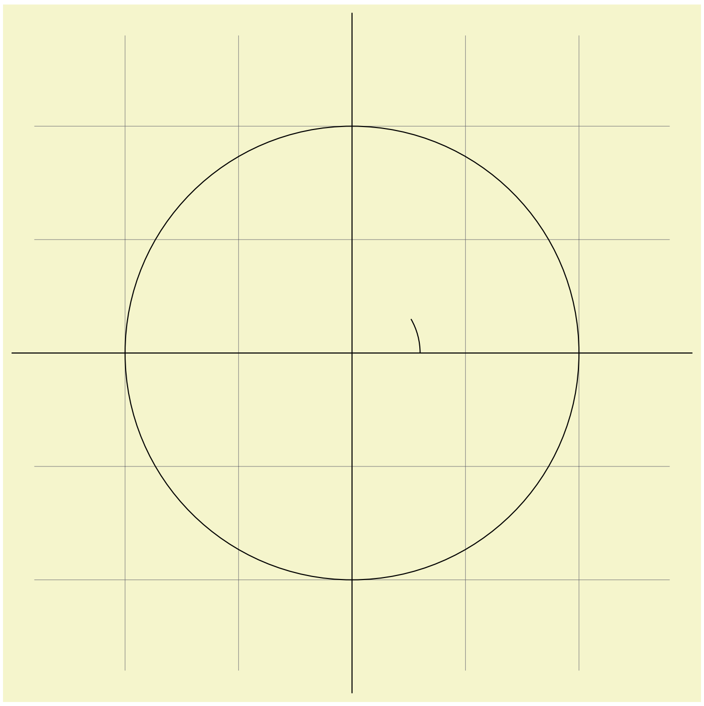
2.10 Arc Path Construction

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".



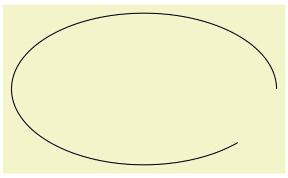
```
\begin{tikzpicture}
  \draw[step=.5cm,gray,very thin] (-1.4,-1.4) grid (1.4,1.4);
  \draw (-1.5,0) -- (1.5,0);
  \draw (0,-1.5) -- (0,1.5);
  \draw (0,0) circle [radius=1cm];
  \draw (3mm,0mm) arc [start angle=0, end angle=30, radius=3mm];
\end{tikzpicture}
```

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}
```

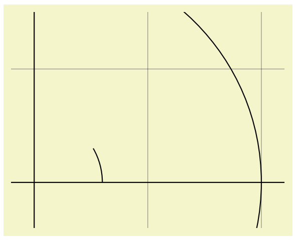
As for circles, you can specify “two” radii in order to get an elliptical arc.



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

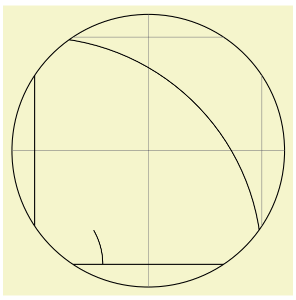
2.11 Clipping a Path

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}
```

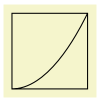
You can also do both at the same time: Draw *and* clip a path. For this, use the `\draw` command and add the `clip` option. (This is not the whole picture: You can also use the `\clip` command and add the `draw` option. Well, that is also not the whole picture: In reality, `\draw` is just a shorthand 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}
```

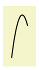
2.12 Parabola and Sine Path Construction

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:

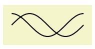


```
\tikz \draw[x=1pt,y=1pt] (0,0) parabola bend (4,16) (6,12);
```

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.

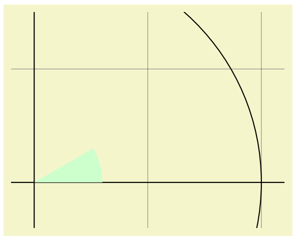
```
A sine \tikz \draw[x=1ex,y=1ex] (0,0) sin (1.57,1); curve.
```



```
\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:



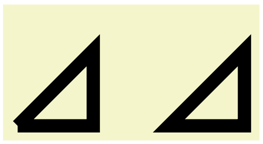
```
\begin{tikzpicture}[scale=3]
  \clip (-0.1,-0.2) rectangle (1.1,0.75);
  \draw[step=.5cm,gray,very thin] (-1.4,-1.4) grid (1.4,1.4);
  \draw (-1.5,0) -- (1.5,0);
  \draw (0,-1.5) -- (0,1.5);
  \draw (0,0) circle [radius=1cm];
  \fill[green!20!white] (0,0) -- (3mm,0mm)
    arc [start angle=0, end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

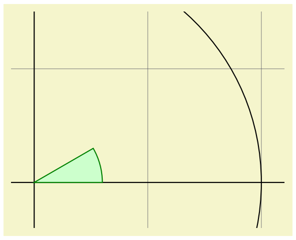
```
  \fill[green!20!white] (0,0) -- (3mm,0mm)
    arc [start angle=0, end angle=30, radius=3mm] -- cycle;
```

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}
```

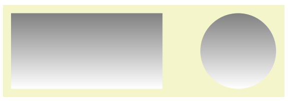
You can also fill and draw a path at the same time using the \filldraw command. This will first draw the path, then fill it. This may not seem too useful, but you can specify different colors to be used for filling and for stroking. These are specified as optional arguments like this:



```
\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}
```

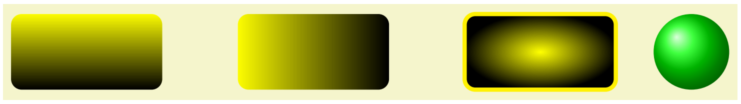
2.14 Shading

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:



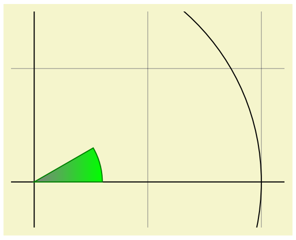
```
  \tikz \shade (0,0) rectangle (2,1)  (3,0.5) circle (.5cm);
```

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.

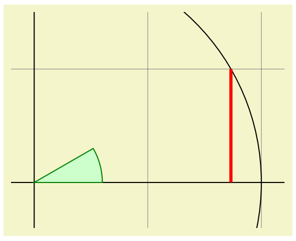
2.15 Specifying Coordinates

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".

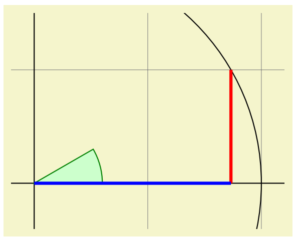
You can add a single `+` sign in front of a coordinate or two of them as in `+(0cm,1cm)` or `++(2cm,0cm)`. Such coordinates are interpreted differently: The first form means "1cm upwards from the previous specified position" and the second means "2cm to the right of the previous specified position, making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

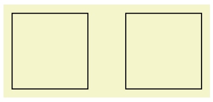
Next, let us draw the cosine line. One way would be to say `(30:1cm |- 0,0) -- (0,0)`. Another way is the following: we "continue" from where the sine ends:



```
\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}
```

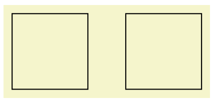
Note that there is no `--` between `(30:1cm)` and `++(0,-0.5)`. In detail, this path is interpreted as follows: "First, the `(30:1cm)` tells me to move my pen to $(\cos 30°, 1/2)$. Next, there comes another coordinate specification, so I move my pen there without drawing anything. This new point is half 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:



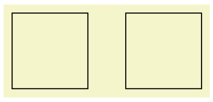
```
\begin{tikzpicture}
  \def\rectanglepath{-- ++(1cm,0cm)  -- ++(0cm,1cm)  -- ++(-1cm,0cm) -- cycle}
  \draw (0,0) \rectanglepath;
  \draw (1.5,0) \rectanglepath;
\end{tikzpicture}
```

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.

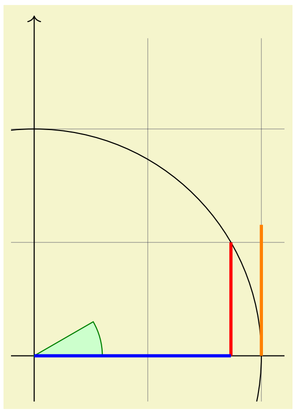
What Karl must do is to create two "invisible" paths that intersect at the position of interest. Creating paths that are not otherwise seen can be done using the `\path` command without any options like `draw` or `fill`. Then, Karl can add the `name path` option to the path for later reference. Once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

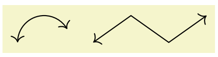
It turns out that adding arrow tips is pretty easy: Karl adds the option `->` to the drawing commands for the axes:



```
\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.

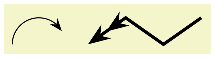
There are certain restrictions to the kind of paths to which arrow tips can be added. As a rule of thumb, you can add arrow tips only to a single open "line". For example, you cannot add tips to, say, a rectangle or a circle. However, you can add arrow tips to curved paths and to paths that have several 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 Ti*k*Z puts at the end. It looks like this when he zooms it: $\rightarrow$. The shape seems vaguely familiar and, indeed, this is exactly the end of TeX's standard arrow used in something like $f\colon A \to B$.

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 Ti*k*Z 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.

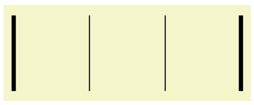
In addition to `Stealth` there are several other predefined kinds of arrow tips Karl can choose from, see Section 105. Furthermore, he can define arrows types himself, if he needs new ones.

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.

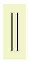
Karl has also already seen that giving options to commands like `\draw` apply only to that command. It turns out that the situation is slightly more complex. First, options to a command like `\draw` are not really options to the command, but they are "path options" and can be given anywhere on the path. 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.

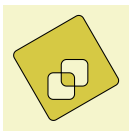
Ti*k*Z provides numerous options that allow you to transform coordinates in Ti*k*Z's private coordinate system. For example, the `xshift` option allows you to shift all subsequent points by a certain amount:



```
\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.

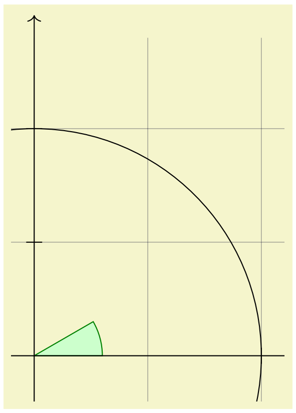
In its basic form, the `\foreach` command is easy to use:

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

```
\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:

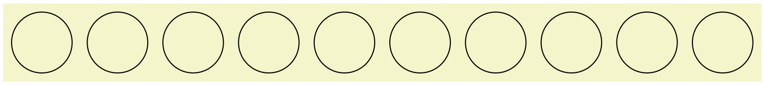


```
\begin{tikzpicture}[scale=3]
  \clip (-0.1,-0.2) rectangle (1.1,1.51);
  \draw[step=.5cm,gray,very thin] (-1.4,-1.4) grid (1.4,1.4);
  \filldraw[fill=green!20,draw=green!50!black] (0,0) -- (3mm,0mm)
      arc [start angle=0, end angle=30, radius=3mm] -- cycle;
  \draw[->] (-1.5,0) -- (1.5,0);
  \draw[->]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

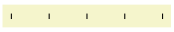
```
\foreach \x in {-1,-0.5,1}
  \draw[xshift=\x cm] (0pt,-1pt) -- (0pt,1pt);
```

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:



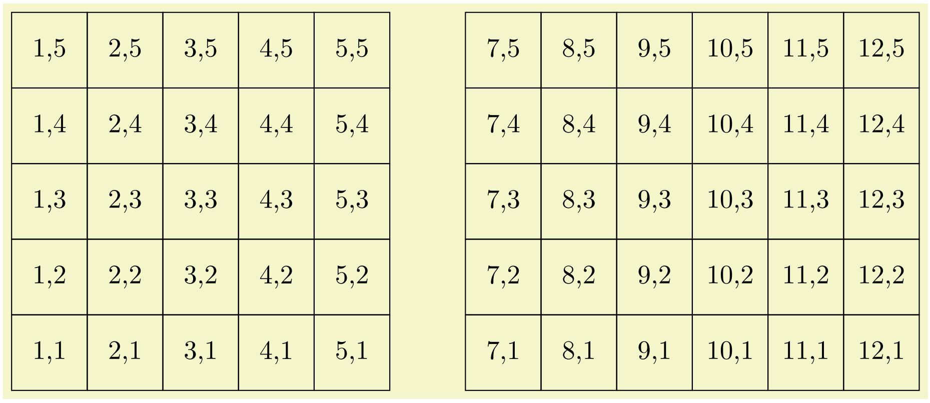
```
\tikz \foreach \x in {1,...,10}
        \draw (\x,0) circle (0.4cm);
```

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

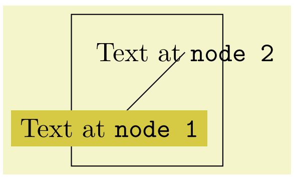
```
\begin{tikzpicture}
  \foreach \x in {1,2,...,5,7,8,...,12}
    \foreach \y in {1,...,5}
    {
      \draw (\x,\y) +(-.5,-.5) rectangle ++(.5,.5);
      \draw (\x,\y) node{\x,\y};
    }
\end{tikzpicture}
```

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

2.21 Adding Text

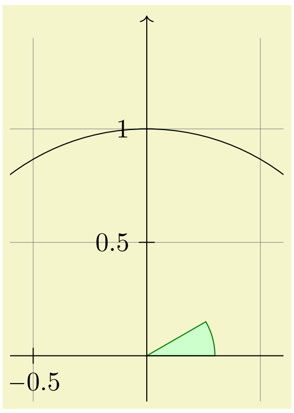
Karl is, by now, quite satisfied with the picture. However, the most important parts, namely the labels, are still missing!

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:

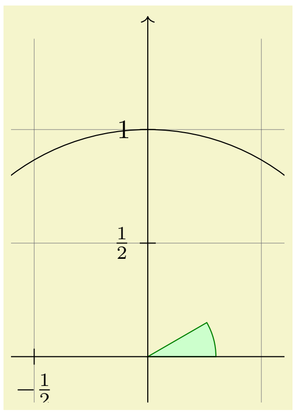


```
\begin{tikzpicture}[scale=3]
  \clip (-0.6,-0.2) rectangle (0.6,1.51);
  \draw[step=.5cm,help lines] (-1.4,-1.4) grid (1.4,1.4);
  \filldraw[fill=green!20,draw=green!50!black] (0,0) -- (3mm,0mm)
    arc [start angle=0, end angle=30, radius=3mm] -- cycle;
  \draw[->] (-1.5,0) -- (1.5,0);    \draw[->]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

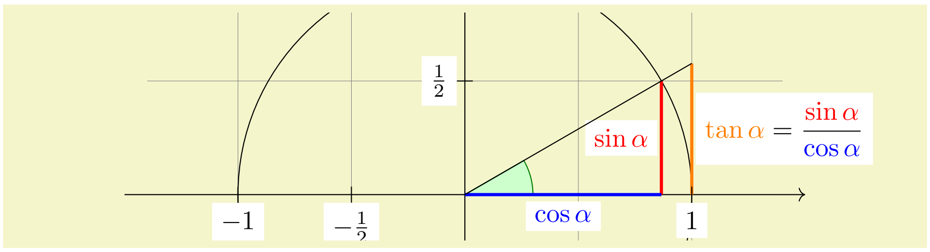
Karl now faces a problem: For the `\foreach` statement, the position `\x` should still be given as `0.5` since Ti*k*Z will not know where `\frac{1}{2}` is supposed to be. On the other hand, the typeset text should really be `\frac{1}{2}`. To solve this problem, `\foreach` offers a special syntax: Instead 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}
```

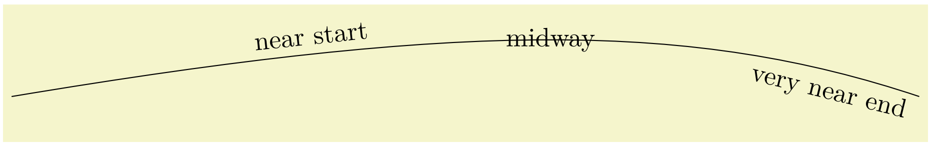
Karl is quite pleased with the result, but his son points out that this is still not perfectly satisfactory: The grid and the circle interfere with the numbers and decrease their legibility. Karl is not very concerned by this (his students do not even notice), but his son insists that there is an easy 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}
```

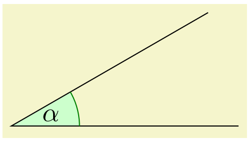
2.22 Pics: The Angle Revisited

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.

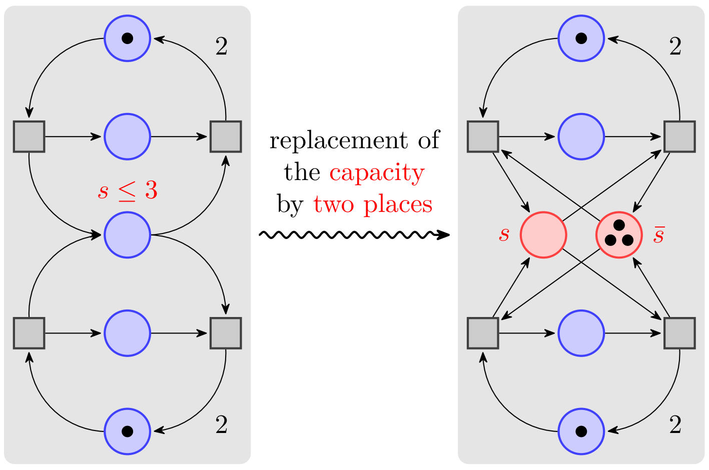
3 Tutorial: A Petri-Net for Hagen

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

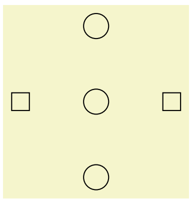
In principle, we already know how to create the graphics that Hagen desires (except perhaps for the snaked line, we will come to that): We start with big light gray rectangle and then add lots of circles and small rectangle, plus some arrows.

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.

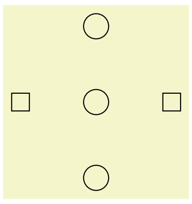
In the previous tutorial we learned that a `node` will add a piece of text at the last coordinate. Thus, each of the five nodes is added at a different position. In the above code, this text is empty (because of the empty {}). So, why do we see anything at all? The answer is the draw option for the node 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

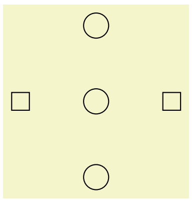
Hagen now understands how the node operation adds nodes to the path, but it seems a bit silly to create a path using the \path operation, consisting of numerous superfluous move-to operations, only to place nodes. He is pleased to learn that there are ways to add nodes in a more sensible manner.

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:

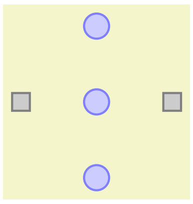


```
\begin{tikzpicture}
  \node at ( 0,2) [circle,draw] {};
  \node at ( 0,1) [circle,draw] {};
  \node at ( 0,0) [circle,draw] {};
  \node at ( 1,1) [rectangle,draw] {};
  \node at (-1,1) [rectangle,draw] {};
\end{tikzpicture}
```

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

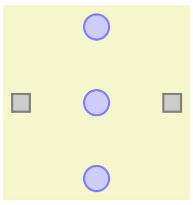
Feeling adventurous, Hagen tries to make the nodes look nicer. In the final picture, the circles and rectangle should be filled with different colors, resulting in the following code:



```
\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.

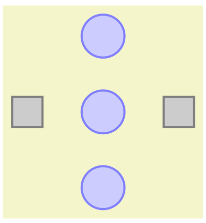
To solve this problem, Hagen uses styles. He defines a style for places and another style for transitions:



```
\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

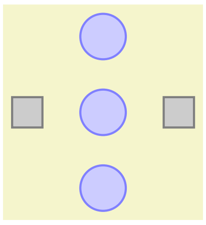
Before Hagen starts naming and connecting the nodes, let us first make sure that the nodes get their final appearance. They are still too small. Indeed, Hagen wonders why they have any size at all, after all, the text is empty. The reason is that Ti*k*Z automatically adds some space around the text.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



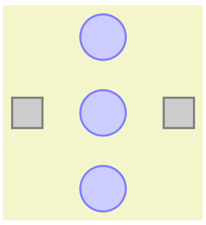
```
\begin{tikzpicture}
  [place/.style={circle,draw=blue!50,fill=blue!20,thick,
                 inner sep=0pt,minimum size=6mm},
   transition/.style={rectangle,draw=black!50,fill=black!20,thick,
                      inner sep=0pt,minimum size=4mm}]
  \node at ( 0,2) [place] {};
  \node at ( 0,1)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

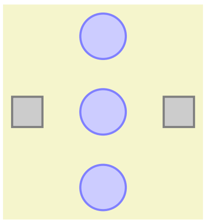
There are two ways to name a node. The first is to use the `name=` option. The second method is to write the desired name in parentheses after the `node` operation. Hagen thinks that this second method seems strange, but he will soon change his opinion.



```
% ... 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.

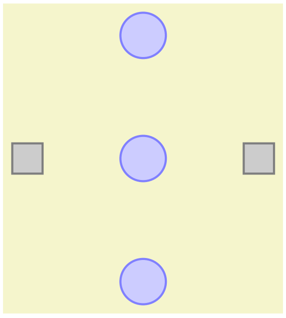
The syntax for the `node` operation is quite liberal with respect to the order in which node names, the `at` specifier, and the options must come. Indeed, you can even have multiple option blocks between the `node` and the text in curly braces, they accumulate. You can rearrange them arbitrarily and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

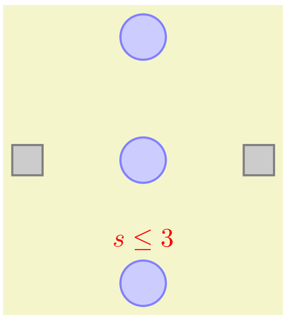
With the `positioning` library loaded, when an option like `below` is followed by `of`, then the position of the node is shifted in such a manner that it is placed at the distance `node distance` in the specified direction of the given direction. The `node distance` is either the distance between the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

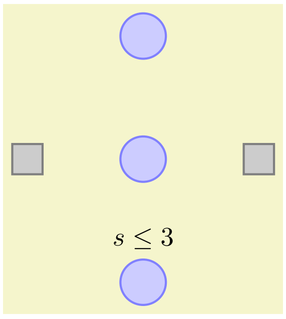
1. Hagen can just add a new node above the `north` anchor of the `semaphore` node.



```
\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}
```

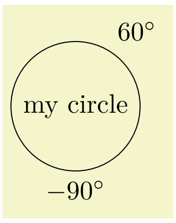
This is a general approach that will "always work".

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`.



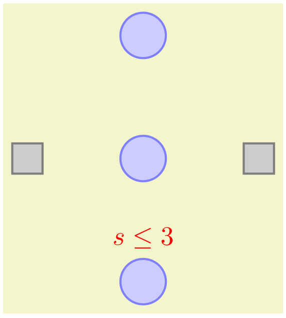
```
\usetikzlibrary {positioning}
\begin{tikzpicture}
  \node[place]      (waiting)                            {};
  \node[place]      (critical)       [below=of waiting]  {};
  \node[place]      (semaphore)      [below=of critical,
                                      label=above:$s\le3$] {};
  \node[transition] (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};
```

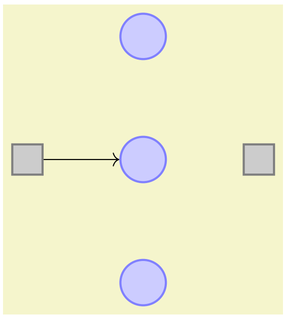
Hagen is not fully satisfied with the `label` option since the label is not red. To achieve this, he has two options: First, he can redefine the `every label` style. Second, he can add options to the label's node. These options are given following the `label=`, so he would write `label=[red]above:$s\le3$`. 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}
```

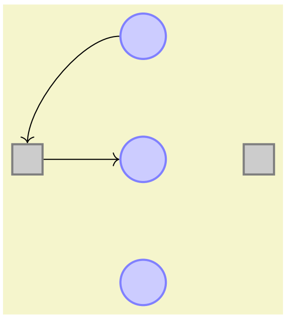
3.10 Connecting Nodes

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}
```

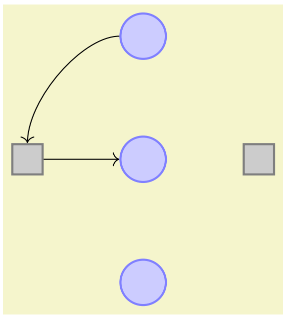
Next, let us tackle the curve from `waiting` to `enter critical`. This can be specified using curves and controls:



```
\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.

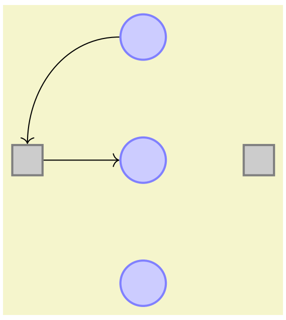
So, let us start improving the code for the edges. First, Hagen can leave out the anchors:



```
\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}
```

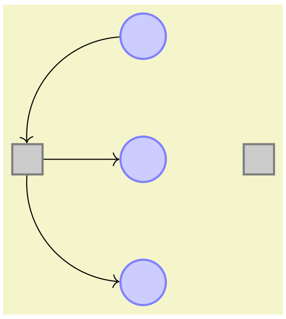
Hagen is a bit surprised that this works. After all, how did Ti*k*Z know that the line from `enter critical` to `critical` should actually start on the borders? Whenever Ti*k*Z encounters a whole node name as a "coordinate", it tries to "be smart" about the anchor that it should choose for this node.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



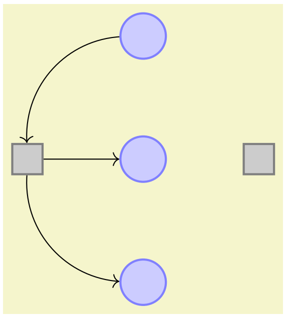
```
\usetikzlibrary {positioning}
\begin{tikzpicture}
  \node[place]      (waiting)                            {};
  \node[place]      (critical)       [below=of waiting]  {};
  \node[place]      (semaphore)      [below=of critical] {};
  \node[transition] (leave critical) [right=of critical] {};
  \node[transition]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

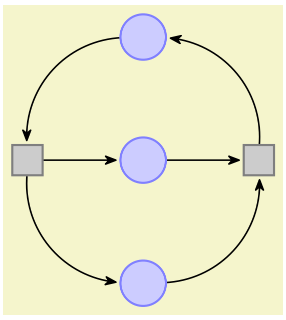
It is now time for Hagen to learn about yet another way of specifying edges: Using the `edge` path operation. This operation is very similar to the `to` operation, but there is one important difference: Like a node the edge generated by the `edge` operation is not part of the main path, but is added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

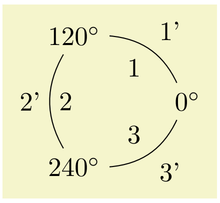
The finishing touch is to introduce two styles `pre` and `post` and to use the `bend angle=45` option to set the bend angle once and for all:



```
\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

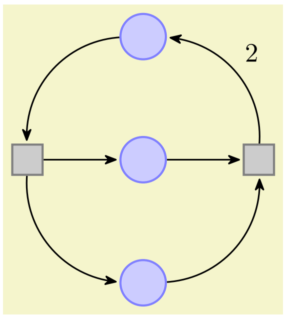
The next thing that Hagen needs to add is the "2" at the arcs. For this Hagen can use Ti*k*Z's automatic node placement: By adding the option `auto`, Ti*k*Z will position nodes on curves and lines in such a way that they are not on the curve but next to it. Adding `swap` will mirror the label with respect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

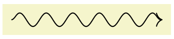


```
\usetikzlibrary {arrows.meta,positioning}
% Styles as before
\begin{tikzpicture}[bend angle=45]
  \node[place]      (waiting)                            {};
  \node[place]      (critical)       [below=of waiting]  {};
  \node[place]      (semaphore)      [below=of critical] {};

  \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

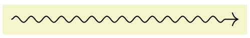
With the node mechanism Hagen can now easily create the two Petri nets. What he is unsure of is how he can create the snaked line between the nets.

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.



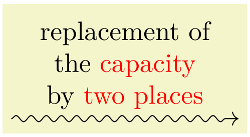
```
\usetikzlibrary {decorations.pathmorphing}
\begin{tikzpicture}
  \draw [->,decorate,decoration=snake] (0,0) -- (2,0);
\end{tikzpicture}
```

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.



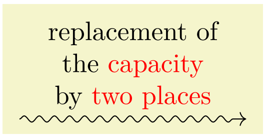
```
\usetikzlibrary {decorations.pathmorphing}
\begin{tikzpicture}
  \draw [->,decorate,
     decoration={snake,amplitude=.4mm,segment length=2mm,post length=1mm}]
    (0,0) -- (3,0);
\end{tikzpicture}
```

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:



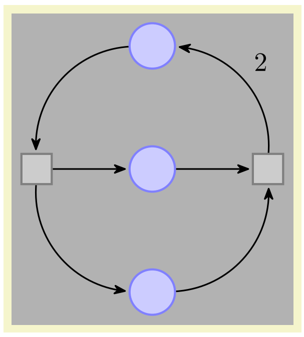
```
\usetikzlibrary {decorations.pathmorphing}
\begin{tikzpicture}
  \draw [->,decorate,
      decoration={snake,amplitude=.4mm,segment length=2mm,post length=1mm}]
    (0,0) -- (3,0)
    node [above,text width=3cm,align=center,midway]
    {
      replacement of the \textcolor{red}{capacity} by
      \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 TikZ compute a rectangle into which all the nodes "fit". For this, the `fit` library can be used. It defines the `fit` options, which, when given to a node, causes the node to be resized and shifted such that it exactly covers all the nodes and coordinates given as parameters to the `fit` option.



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

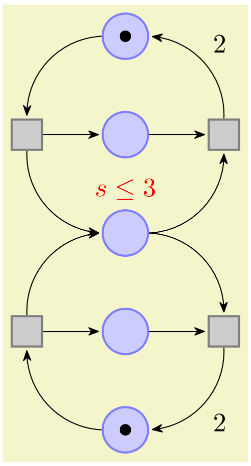
3.14 The Complete Code

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.

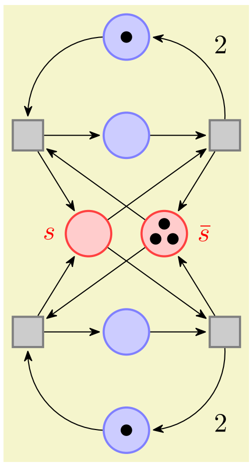
First, Hagen needs less style definitions, but he still needs to specify the colors of places and transitions.

```
\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*

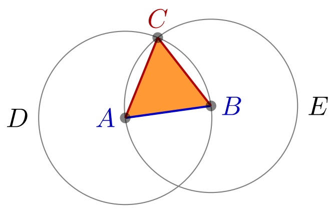
In this third tutorial we have a look at how Ti*k*Z can be used to draw geometric constructions.

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}
```

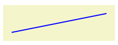
```
% For ConTeXt:
\usemodule[tikz]
\usetikzlibrary[calc,intersections,through,backgrounds]
```

[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

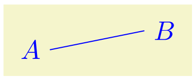
The first part of the picture that Euclid wishes to draw is the line AB. That is easy enough, something like `\draw (0,0) -- (2,1);` might do. However, Euclid does not wish to reference the two points A and B as $(0,0)$ and $(2,1)$ subsequently. Rather, he wishes to just write `A` and `B`. Indeed, the 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

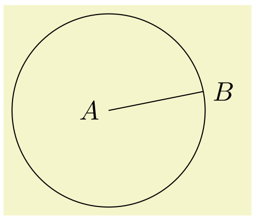
The first tricky construction is the circle around A. We will see later how to do this in a very simple manner, but first let us do it the "hard" way.

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}
```

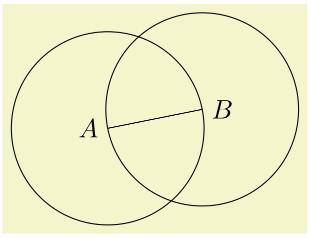
Each assignment in a let operation starts with `\p`, usually followed by a ⟨*digit*⟩. Then comes an equal sign and a coordinate. The coordinate is evaluated and the result is stored internally. From then on you can use the following expressions:

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.

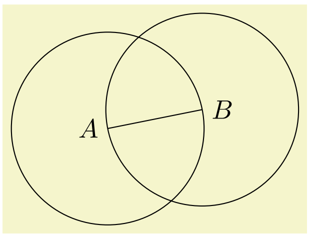
Note that `\p1` is not a coordinate in the usual sense. Rather, it just expands to a string like `10pt,20pt`. So, you cannot write, for instance, `(\p1.center)` since this would just expand to `(10pt,20pt.center)`, which makes no sense.

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}
```

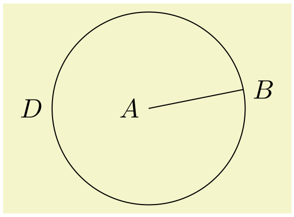
In the above example, you may wonder, what `\n1` would yield? The answer is that it would be undefined – the `\p`, `\x`, and `\y` macros refer to the same logical point, while the `\n` macro has "its own namespace". We could even have replaced `\n2` in the example by `\n1` and it would still work. Indeed, 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.

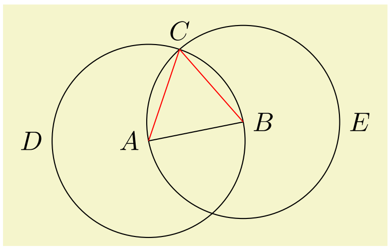


```
\usetikzlibrary {through}
\begin{tikzpicture}
  \coordinate [label=left:$A$]  (A) at (0,0);
  \coordinate [label=right:$B$] (B) at (1.25,0.25);
  \draw (A) -- (B);

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

4.1.4 The Intersection of the Circles

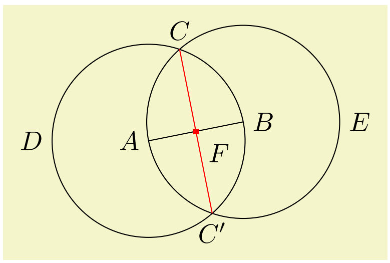
Euclid can now draw the line and the circles. The final problem is to compute the intersection of the two circles. This computation is a bit involved if you want to do it "by hand". Fortunately, the `intersections` library allows us to compute the intersection of arbitrary paths.

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}
```

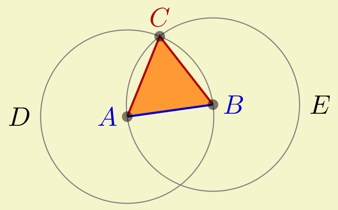
It turns out that this can be further shortened: The `name intersections` takes an optional argument `by`, which lets you specify names for the coordinates and options for them. This creates more compact code. Although Euclid does not need it for the current picture, it is just a small step to computing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



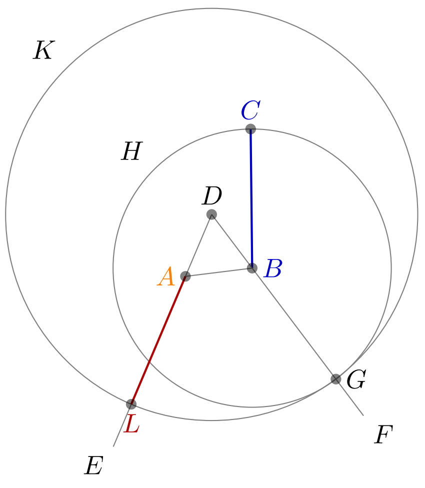
Proposition I
To construct an equilateral triangle on a given finite straight line.

Let AB be the given finite straight line. …

```
\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.

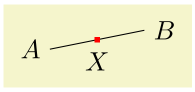
Let $A$ be the given point, and BC the given straight line. It is required to place a straight line equal to the given straight line BC with one end at the point $A$.
Join the straight line AB from the point $A$ to the point $B$, and construct the equilateral triangle DAB on it.
Produce the straight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

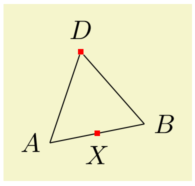
For this reason, Ti*k*Z supports some simplifications. First, there is a simple syntax for computing a point that is "partway" on a line from p to q: You place these two points in a coordinate calculation – remember, they start with `($` and end with `$)` – and then combine them using `!`⟨*part*⟩`!`.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



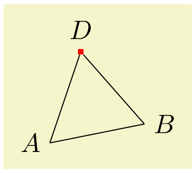
```
\usetikzlibrary {calc}
\begin{tikzpicture}
  \coordinate [label=left:$A$]  (A) at (0,0);
  \coordinate [label=right:$B$] (B) at (1.25,0.25);
  \draw (A) -- (B);
  \node [fill=red,inner sep=1pt,label=below:$X$] (X) at ($ (A)!.5!(B) $) {};
\end{tikzpicture}
```

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}
```

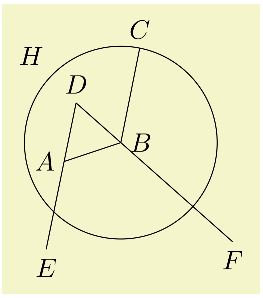
Finally, it is not necessary to explicitly name the point X. Rather, again like in the `xcolor` package, it is possible to chain partway modifiers:



```
\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

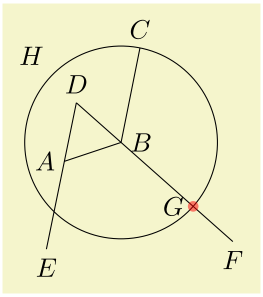
The next step in the construction is to draw a circle around $B$ through $C$, which is easy enough to do using the `circle through` option. Extending the lines DA and DB can be done using partway calculations, but this time with a part value outside the range $[0, 1]$:



```
\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}
```

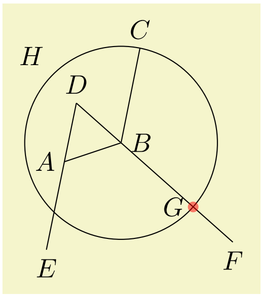
We now face the problem of finding the point $G$, which is the intersection of the line BF and the circle $H$. One way is to use yet another variant of the partway computation: Normally, a partway computation has the form $\langle p\rangle!\langle factor\rangle!\langle q\rangle$, resulting in the point $(1-\langle factor\rangle)\langle p\rangle+\langle factor\rangle\langle q\rangle$. 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 $H$:



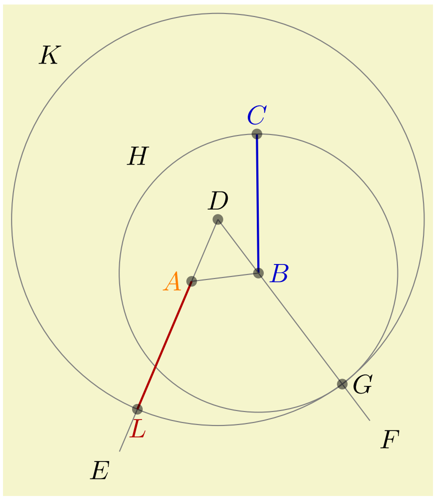
```
\usetikzlibrary {calc,through}
  \node (H) [label=135:$H$,draw,circle through=(C)] at (B) {};
  \path let \p1 = ($ (B) - (C) $) in
    coordinate [label=left:$G$] (G) at ($ (B) ! veclen(\x1,\y1) ! (F) $);
  \fill[red,opacity=.5] (G) circle (2pt);
```

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}
```