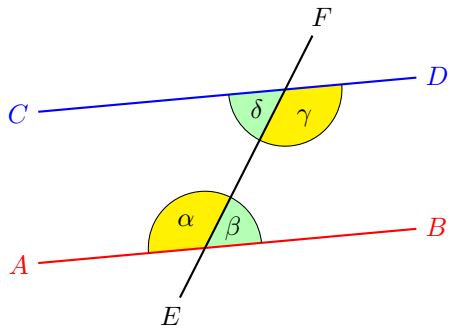
# Part I
# 绘何物为

[1]
*by Till Tantau*




When we assume that $AB$ and $CD$ are parallel, i. e., $AB \parallel CD$, then $\alpha = \delta$ and $\beta = \gamma$.

```
\begin{tikzpicture}[angle radius=.75cm]

  \node (A) at (-2,0)    [red,left]   {$A$};
  \node (B) at ( 3,.5)    [red,right]  {$B$};
  \node (C) at (-2,2)  [blue,left]  {$C$};
  \node (D) at ( 3,2.5)  [blue,right] {$D$};
  \node (E) at (60:-5mm) [below]      {$E$};
  \node (F) at (60:3.5cm)  [above]      {$F$};

  \coordinate (X) at (intersection cs:first line={(A)--(B)}, second line={(E)--(F)});
  \coordinate (Y) at (intersection cs:first line={(C)--(D)}, second line={(E)--(F)});

  \path
    (A) edge [red, thick]  (B)
    (C) edge [blue, thick] (D)
    (E) edge [thick]       (F)
      pic ["$\alpha$",  draw, fill=yellow]   {angle = F--X--A}
      pic ["$\beta$",   draw, fill=green!30] {angle = B--X--F}
      pic ["$\gamma$",  draw, fill=yellow]   {angle = E--Y--D}
      pic ["$\delta$",  draw, fill=green!30] {angle = C--Y--E};

  \node at ($ (D)!.5!(B) $) [right=1cm,text width=6cm,rounded corners,fill=red!20,inner sep=1ex]
    {
      When we assume that $\color{red}AB$ and $\color{blue}CD$ are
      parallel, i.\,e., ${\color{red}AB} \mathbin{\|} \color{blue}CD$,
      then $\alpha = \delta$ and $\beta = \gamma$.
    };
\end{tikzpicture}
```

[1] 该部分标题原文为德文：TikZ ist *kein* Zeichenprogramm, 翻译成英文就是“TikZ is not a drawing program”，中文意思是“TikZ 不是一个绘图程序”。然而德文采用的是“GNU's Not Unix!”式的递归缩写，这里如果直接采用原文也并非不可以，但是中文博大精深，一定有对应的贴切的翻译。

我这里译成“绘何物为”，用了拼音的递归：Huì hé wù wéi。意即“‘绘’是什么呢”，当然也可以将“绘”直接作为动词，理解成“绘制什么呢”。这样中文含义就和原文含义形成一问一答，无论是形式上还是内容上，都有了合理的对应。当然，这里着实夹杂了译者的私货，正文中依旧使用 TikZ 来指代这一绘图系统。

# 1 设计原则

这一节论述 TikZ 前端背后的设计原则，TikZ 意思是“TikZ 不是一个绘图程序”。要使用 TikZ，LaTeX 用户得在序言部分加上 `usepackage {tikz}`，而 plainTeX 用户则是 `input tikz.tex`。TikZ 提供了一种描述图形的语法，易学易用，让你的生活更简单。

TikZ 的命令和语法受几个来源影响。基本的命令名称和路径操作的概念来自 METAFONT，选项机制源于 PSTRICKS，样式的概念联想自 SVG，图的语法取材于 GRAPHVIZ。为了让它们一起工作，一些折中是必要的。我还加了一些自己的想法，比如坐标变换。

TikZ 遵循下面的基本设计原则：

1.
2. 指定的
3. 图形参数用键值对

## 1.1 Special Syntax For Specifying Points

TikZ provides a special syntax for specifying points and coordinates. In the simplest case, you provide two TeX dimensions, separated by commas, in round brackets as in `(1cm,2pt)`.

You can also specify a point in polar coordinates by using a colon instead of a comma as in `(30:1cm)`, which means “1cm in a 30 degrees direction.”

If you do not provide a unit, as in `(2,1)`, you specify a point in PGF's $xy$-coordinate system. By default, the unit $x$-vector goes 1cm to the right and the unit $y$-vector goes 1cm upward.

By specifying three numbers as in `(1,1,1)` you specify a point in PGF's $xyz$-coordinate system.

It is also possible to use an anchor of a previously defined shape as in `(first node.south)`.

You can add two plus signs before a coordinate as in `++(1cm,0pt)`. This means “1cm to the right of the last point used.” This allows you to easily specify relative movements. For example, `(1,0) ++(1,0) ++(0,1)` specifies the three coordinates `(1,0)`, then `(2,0)`, and `(2,1)`.

Finally, instead of two plus signs, you can also add a single one. This also specifies a point in a relative manner, but it does not “change” the current point used in subsequent relative commands. For example, `(1,0) +(1,0) +(0,1)` specifies the three coordinates `(1,0)`, then `(2,0)`, and `(1,1)`.

## 1.2 Special Syntax For Path Specifications

When creating a picture using TikZ, your main job is the specification of *paths*. A path is a series of straight or curved lines, which need not be connected. TikZ makes it easy to specify paths, partly using the syntax of METAPOST. For example, to specify a triangular path you use

```
(5pt,0pt) -- (0pt,0pt) -- (0pt,5pt) -- cycle
```

and you get ◺ when you draw this path.

## 1.3 Actions on Paths

A path is just a series of straight and curved lines, but it is not yet specified what should happen with it. One can *draw* a path, *fill* a path, *shade* it, *clip* it, or do any combination of these. Drawing (also known as *stroking*) can be thought of as taking a pen of a certain thickness and moving it along the path, thereby drawing on the canvas. Filling means that the interior of the path is filled with a uniform color. Obviously, filling makes sense only for *closed* paths and a path is automatically closed prior to filling, if necessary.

Given a path as in `\path (0,0) rectangle (2ex,1ex);`, you can draw it by adding the `draw` option as in `\path[draw] (0,0) rectangle (2ex,1ex);`, which yields ▭. The `\draw` command is just an abbreviation for `\path[draw]`. To fill a path, use the `fill` option or the `\fill` command, which is an abbreviation for `\path[fill]`. The `\filldraw` command is an abbreviation for `\path[fill,draw]`. Shading is caused by the `shade` option (there are `\shade` and `\shadedraw` abbreviations) and clipping by the `clip` option. There is also a `\clip` command, which does the same as `\path[clip]`, but not commands like `\drawclip`. Use, say, `\draw[clip]` or `\path[draw,clip]` instead.

All of these commands can only be used inside `{tikzpicture}` environments.

TikZ allows you to use different colors for filling and stroking.

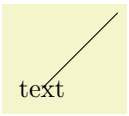
## 1.4 Key-Value Syntax for Graphic Parameters

Whenever Ti*k*Z draws or fills a path, a large number of graphic parameters influences the rendering. Examples include the colors used, the dashing pattern, the clipping area, the line width, and many others. In Ti*k*Z, all these options are specified as lists of so called key-value pairs, as in `color=red`, that are passed as optional parameters to the path drawing and filling commands. This usage is similar to PSTRICKS. For example, the following will draw a thick, red triangle;

```
\tikz \draw[line width=2pt,color=red] (1,0) -- (0,0) -- (0,1) -- cycle;
```

## 1.5 Special Syntax for Specifying Nodes

Ti*k*Z introduces a special syntax for adding text or, more generally, nodes to a graphic. When you specify a path, add nodes as in the following example:




```
\tikz \draw (1,1) node {text} -- (2,2);
```

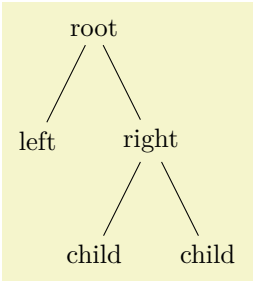
Nodes are inserted at the current position of the path, but either *after* (the default) or *before* the complete path is rendered. When special options are given, as in `\draw (1,1) node[circle,draw] {text};`, the text is not just put at the current position. Rather, it is surrounded by a circle and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

You can add a name to a node for later reference either by using the option `name=`⟨*node name*⟩ or by stating the node name in parentheses outside the text as in `node[circle](name){text}`.

Predefined shapes include `rectangle`, `circle`, and `ellipse`, but it is possible (though a bit challenging) to define new shapes.

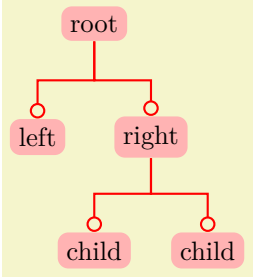
## 1.6 Special Syntax for Specifying Trees

The "node syntax" can also be used to draw tress: A `node` can be followed by any number of children, each introduced by the keyword `child`. The children are nodes themselves, each of which may have children in turn.

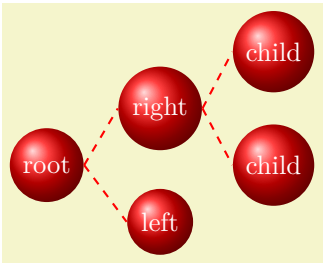



```
\begin{tikzpicture}
  \node {root}
    child {node {left}}
    child {node {right}
      child {node {child}}
      child {node {child}}
    };
\end{tikzpicture}
```

Since trees are made up from nodes, it is possible to use options to modify the way trees are drawn. Here are two examples of the above tree, redrawn with different options:



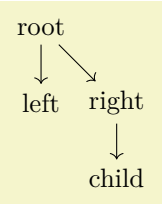

```
\begin{tikzpicture}
  [edge from parent fork down, sibling distance=15mm, level distance=15mm,
   every node/.style={fill=red!30,rounded corners},
   edge from parent/.style={red,-o,thick,draw}]
  \node {root}
      child {node {left}}
      child {node {right}
        child {node {child}}
        child {node {child}}
      };
\end{tikzpicture}
```




```
\begin{tikzpicture}
  [parent anchor=east,child anchor=west,grow=east,
   sibling distance=15mm, level distance=15mm,
   every node/.style={ball color=red,circle,text=white},
   edge from parent/.style={draw,dashed,thick,red}]
  \node {root}
      child {node {left}}
      child {node {right}
        child {node {child}}
        child {node {child}}
      };
\end{tikzpicture}
```

## 1.7 Special Syntax for Graphs

The `\node` command gives you fine control over where nodes should be placed, what text they should use, and what they should look like. However, when you draw a graph, you typically need to create numerous fairly similar nodes that only differ with respect to the name they show. In these cases, the `graph` syntax can be used, which is another syntax layer build "on top" of the node syntax.




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

The syntax of the `graph` command extends the so-called DOT-notation used in the popular GRAPHVIZ program.

Depending on the version of TEX you use (it must allow you to call Lua code, which is the case for LuaTEX), you can also ask Ti*k*Z to do automatically compute good positions for the nodes of a graph using one of several integrated *graph drawing algorithms*.

## 1.8 Grouping of Graphic Parameters

Graphic parameters should often apply to several path drawing or filling commands. For example, we may wish to draw numerous lines all with the same line width of 1pt. For this, we put these commands in a `{scope}` environment that takes the desired graphic options as an optional parameter. Naturally, the specified graphic parameters apply only to the drawing and filling commands inside the environment. Furthermore, nested `{scope}` environments or individual drawing commands can override the graphic parameters of outer `{scope}` environments. In the following example, three red lines, two green lines, and one blue line are drawn:

```
\begin{tikzpicture}
  \begin{scope}[color=red]
    \draw (0mm,10mm) -- (10mm,10mm);
    \draw (0mm, 8mm) -- (10mm, 8mm);
    \draw (0mm, 6mm) -- (10mm, 6mm);
  \end{scope}
  \begin{scope}[color=green]
    \draw              (0mm, 4mm) -- (10mm, 4mm);
    \draw              (0mm, 2mm) -- (10mm, 2mm);
    \draw[color=blue] (0mm, 0mm) -- (10mm, 0mm);
  \end{scope}
\end{tikzpicture}
```

The `{tikzpicture}` environment itself also behaves like a `{scope}` environment, that is, you can specify graphic parameters using an optional argument. These optional apply to all commands in the picture.

## 1.9 Coordinate Transformation System

Ti*k*Z supports both PGF's *coordinate* transformation system to perform transformations as well as *canvas* transformations, a more low-level transformation system. (For details on the difference between coordinate transformations and canvas transformations see Section **??**.)

The syntax is set up in such a way that it is harder to use canvas transformations than coordinate transformations. There are two reasons for this: First, the canvas transformation must be used with great care and often results in "bad" graphics with changing line width and text in wrong sizes. Second, PGF loses track of where nodes and shapes are positioned when canvas transformations are used. So, in almost all circumstances, you should use coordinate transformations rather than canvas transformations.