A Picture for Karl's New Students

This tutorial is intended for new users of CeTZ. It does not give an exhaustive account of all the features of CeTZ, just of those you are likely to use right away. This tutorial also acts as a parallel to TikZ's tutorial A Picture for Karl's Students.

Karl is a math and chemistry high-school teacher. He used to create the graphics in his worksheets and exams using the TikZ package with LaTeX. While the results were acceptable, Karl, for his own reasons, has started using Typst instead. He looks through the provided packages in Typst: Universe and finds CeTZ, which is supposed to stand for "CeTZ, ein Typst Zeichenpaket" and appears appropriate.

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. He already has the graphic drawn with TikZ and would like to keep it as close to it as possible:

```
Either the example gets rendered using a block or we pre-render it I'm not sure
yet.
```

Setting up the Environment

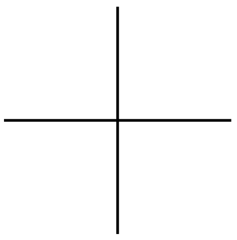
In CeTZ, to draw a picture, two imports and a function call is all you need. Karl sets up his file as follows:

```
#set page(width: auto, height: auto)
#import "@preview/cetz:0.3.3"

We are working on
#cetz.canvas({
  import cetz.draw: *
  line((-1.5, 0), (1.5, 0))
  line((0, -1.5), (0, 1.5))
})
```

When compiled via the Typst web app or the Typst command line interface, the resulting output will contain something that looks like this:

We are working on



```
We are working on
#cetz.canvas({
  import cetz.draw: *
  line((-1.5, 0), (1.5, 0))
  line((0, -1.5), (0, 1.5))
})
```

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 `cetz` is imported. The `canvas` function in the `cetz` module is then called and a pair of curly braces are placed as the function's first (and only) positional argument. The braces create a scope or body, in which more functions can be called, but first must be imported from the `draw` module.

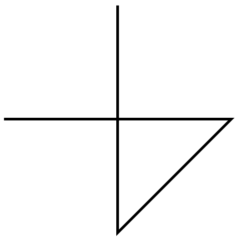
Inside the body there are two `line` functions. They are `draw` functions that `draw` straight lines between given positions. The first `line` function is given the parameters `(-1.5, 0), (1.5, 0)`, which refer to a point at position (−1.5,0) and (1.5,0). Here, the positions are specified within a special 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.

Line Construction

The basic building block of all pictures in CeTZ are draw functions. A draw function is a function that can be called inside the canvas body in order to create the graphic, such as `line`, `bezier`, `rect` and many more (not all draw functions actually draw something, like `set-style`, but still effect the outcome of the picture).

In order to draw a path of straight lines, the `line` draw function can be used. You specify the coordinates of the start position by passing an array with two numbers (a coordinate type) to the first parameter of `line`. The second `coordinate` must be given as the second parameter of the function otherwise it will panic. Subsequent coordinates can be passed to the function to draw additional lines between the previous and next coordinates.

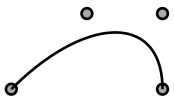


```
#cetz.canvas({
  import cetz.draw: *
  line((-1.5, 0), (1.5, 0), (0, -1.5), (0, 1.5))
})
```

Curve Construction

The next think 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, CeTZ provides several other draw functions, the most useful here would be the `bezier` function. As the name suggests, it can draw a bezier curve when a start and end coordinate is given, as well as one or two control points.

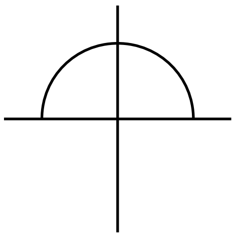
Here is an example (the control points have been added for clarity):



```
#cetz.canvas({
  import cetz.draw: *
  // start and end
  circle((0, 0), radius: 2pt, fill: gray)
  circle((2, 0), radius: 2pt, fill: gray)
  // control points
  circle((1, 1), radius: 2pt, fill: gray)
  circle((2, 1), radius: 2pt, fill: gray)

  bezier((0, 0), (2, 0), (1, 1), (2, 1))
})
```

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



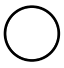
```
#cetz.canvas({
  import cetz.draw: *
  line((-1.5, 0), (1.5, 0))
  line((0, -1.5), (0, 1.5))

  bezier((-1, 0), (0, 1), (-1, 0.555), (-0.555, 1))
  bezier((0, 1), (1, 0), (0.555, 1), (1, 0.555))
})
```

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

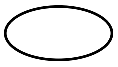
Circle Construction

In order to draw a circle, the `circle` draw function can be used:



```
#cetz.canvas({
  import cetz.draw: *
  circle((0, 0), radius: 10pt)
})
```

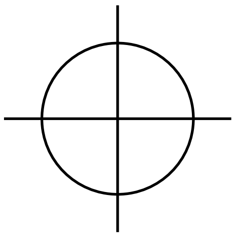
You can also draw an ellipse with this draw function by passing an array of two numbers to the `radius` argument:



```
#cetz.canvas({
  import cetz.draw: *
  circle((0, 0), radius: (20pt, 10pt))
})
```

To draw an ellipse whose axes are not horizontal and vertical, but point in an arbitrary direction you can use transformations, which are explained later.

So, returning to Karl's problem, he can write `circle((0, 0))` to draw the circle as, by default, the `radius` argument is 1:



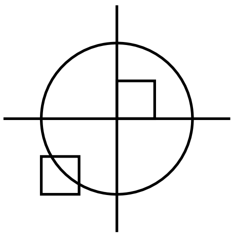
```
#cetz.canvas({
  import cetz.draw: *
  line((-1.5, 0), (1.5, 0))
  line((0, -1.5), (0, 1.5))

  circle((0, 0))
})
```

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 CeTZ has transformation draw functions 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.

Rectangle 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. To do so, the `rect` draw function can be used. Two coordinates should be passed as arguments, they specify the corners of the rectangle:



```
#cetz.canvas({
  import cetz.draw: *
  line((-1.5, 0), (1.5, 0))
  line((0, -1.5), (0, 1.5))
  circle((0, 0))

  rect((0, 0), (0.5, 0.5))
  rect((-0.5, -0.5), (-1, -1))
})
```

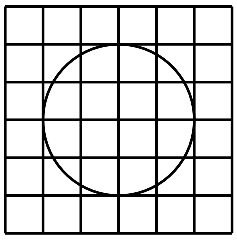
While this may be nice in other situations, this is not really leading anywhere with Karl's problem: First, we would need an awful lot of these rectangles and then there is the border that is not "closed".

So, Karl is about to resort to simply drawing four vertical and four horizontal lines using the nice `line` draw function, when he learns that there is a `grid` draw function.

Grid Construction

The `grid` draw function adds a grid to the picture. It will add lines making up a grid that fills the rectangle specified by two coordinates passed to it.

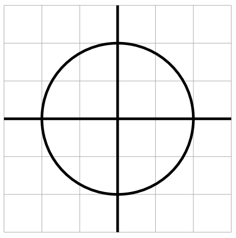
For Karl, the following code could be used:



```
#cetz.canvas({
  import cetz.draw: *
  line((-1.5, 0), (1.5, 0))
  line((0, -1.5), (0, 1.5))
  circle((0, 0))

  grid((-1.5, -1.5), (1.5, 1.5), step: 0.5)
})
```

Having another look at his desired picture, Karl notices that it would be nice for the `grid` to be more subdued. To subdue the `grid`, Karl adds more named arguments to the grid draw function. First, he uses the color gray for the grid lines. Second, he reduces the line width to `0.2pt` (TikZ's `very thin`). Finally, he swaps the ordering of the commands so that the grid is drawn first and everything else on top.

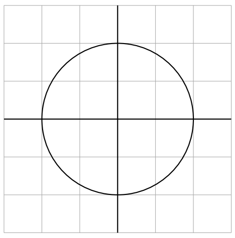


```
#cetz.canvas({
  import cetz.draw: *
  grid((-1.5, -1.5), (1.5, 1.5), step: 0.5, stroke: gray +
0.2pt)
  line((-1.5, 0), (1.5, 0))
  line((0, -1.5), (0, 1.5))
  circle((0, 0))
})
```

Adding a Touch of Style

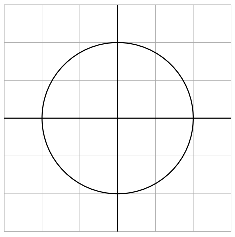
Karl notices that the thickness of the circle and axes paths are much greater than the grid's thickness. He learns that CeTZ's default stroke thickness is actually `1pt` and not TikZ's `0.4pt`. Karl decides that he would like to use the thinner lines to keep this new picture as close to the original as possible.

We can use the `set-style` draw function to apply styling to all subsequent draw functions, similar to how Typst's `set` and `show` rules work. To set the stroke's thickness he uses the named argument `stroke: 0.4pt`:



```
#cetz.canvas({
  import cetz.draw: *
  set-style(stroke: 0.4pt)
  grid((-1.5, -1.5), (1.5, 1.5), step: 0.5, stroke: gray +
0.2pt)
  line((-1.5, 0), (1.5, 0))
  line((0, -1.5), (0, 1.5))
  circle((0, 0))
})
```

Karl can also move the grid's styling into the same `set-style` function by passing it as a dictionary to the `grid` named argument:

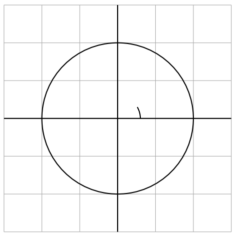


```
#cetz.canvas({
  import cetz.draw: *
  set-style(
    stroke: 0.4pt,
    grid: (
      stroke: gray + 0.2pt,
      step: 0.5
    )
  )
  grid((-1.5, -1.5), (1.5, 1.5))
  line((-1.5, 0), (1.5, 0))
  line((0, -1.5), (0, 1.5))
  circle((0, 0))
})
```

Arc Construction

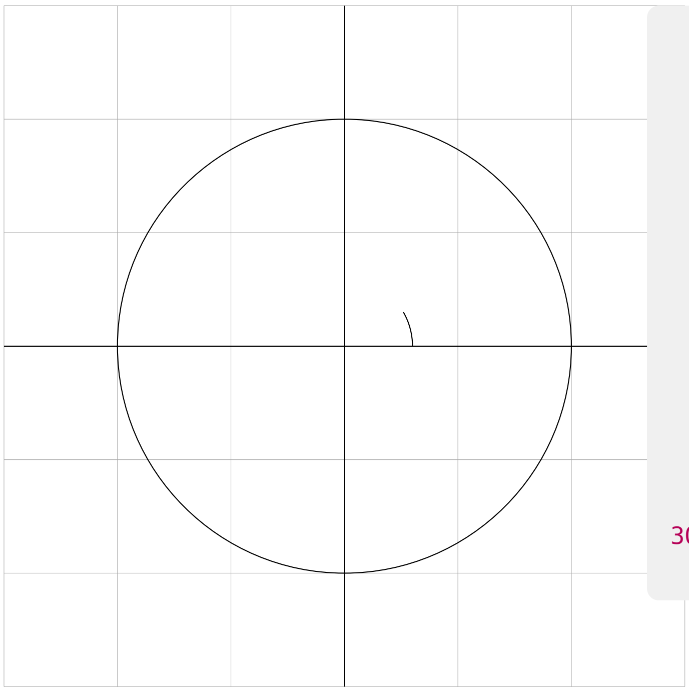
Our next obstacle is to draw the `arc` for the angle. For this, the arc draw function can be used, which draws part of a circle or ellipse. This function requires arguments to specify the arc. An example would be `arc((0, 0), start: 10deg, stop: 80deg, radius: 10pt)`, which creates an arc starting at (0,0)(0,0) at an angle of 10°10° to 80°80° with a radius of `10pt`. Karl obviously needs an arc from 0°0° to 30°30°. The radius should be something relatively small, perhaps around one third of the circle's radius.

```
#cetz.canvas({
  import cetz.draw: *
  set-style(
    stroke: 0.4pt,
    grid: (
```



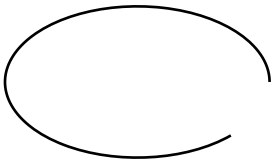
```
      stroke: gray + 0.2pt,
      step: 0.5
    )
  )
  grid((-1.5, -1.5), (1.5, 1.5))
  line((-1.5, 0), (1.5, 0))
  line((0, -1.5), (0, 1.5))
  circle((0, 0))
  arc((3mm, 0), start: 0deg, stop: 30deg, radius: 3mm)
})
```

Karl thinks this is really a bit small and he cannot continue unless he learns how to do scaling. For this, he can use the `scale` draw function at the start of the canvas body



```
#cetz.canvas({
  import cetz.draw: *
  set-style(
    stroke: 0.4pt,
    grid: (
      stroke: gray + 0.2pt,
      step: 0.5
    )
  )
  scale(3)
  grid((-1.5, -1.5), (1.5, 1.5))
  line((-1.5, 0), (1.5, 0))
  line((0, -1.5), (0, 1.5))
  circle((0, 0))
  arc((3mm, 0), start: 0deg, stop:
30deg, radius: 3mm)
})
```

As for circles, you can specify the radius as an array of two numbers to get an elliptical arc.

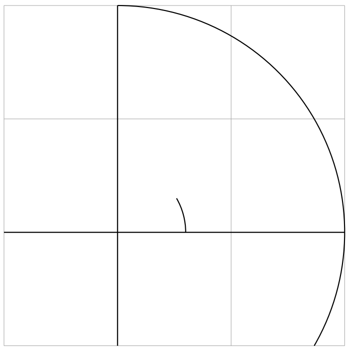


```
#cetz.canvas({
  import cetz.draw: *
  arc((0, 0), start: 0deg, stop: 315deg, radius: (1.75,
1))
})
```

Not Really Path Clipping

In order to save space in this manual, it would be nice to clip Karl's graphics a bit so we can focus on the "interesting" parts. Unfortunately clipping is not currently possible in CeTZ. So instead we can replace the circle with an arc and modify the grid.

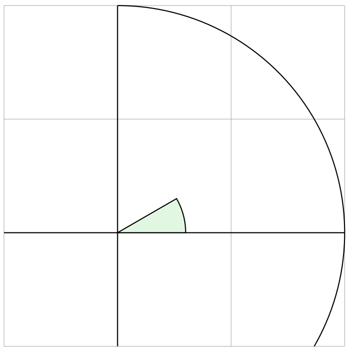
```
#cetz.canvas({
  import cetz.draw: *
  set-style(
    stroke: 0.4pt,
```



```
      grid: (
        stroke: gray + 0.2pt,
        step: 0.5
      )
    )
    scale(3)
    grid((-0.5, -0.5), (1, 1))
    line((-0.5, 0), (1, 0))
    line((0, -0.5), (0, 1))
    arc((0, 1), start: 90deg, delta: -120deg)
    arc((3mm, 0), start: 0deg, stop: 30deg, radius:
3mm)
  })
```

Filling and Drawing

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



```
  #cetz.canvas({
    import cetz.draw: *
    set-style(
      stroke: 0.4pt,
      grid: (
        stroke: gray + 0.2pt,
        step: 0.5
      )
    )
    scale(3)
    grid((-0.5, -0.5), (1, 1))
    line((-0.5, 0), (1, 0))
    line((0, -0.5), (0, 1))
    arc((0, 1), start: 90deg, delta: -120deg)
    arc(
      (3mm, 0),
      start: 0deg,
      stop: 30deg,
      radius: 3mm,
      mode: "PIE",
      fill: color.mix((green, 20%), white),
    )
  })
```

The color `color.mix((green, 20%), white)` means 20% green and 80% white mixed together. In fact `fill` can take any value of type `color`, you can even do shading!

By default arcs are in “OPEN” mode where only the curve is drawn and only the chord of the arc will be filled. By using `mode: "PIE"` the arc will be closed around its origin leaving you with a shape akin to a slice of pie, or in this case an angle which can be properly filled.

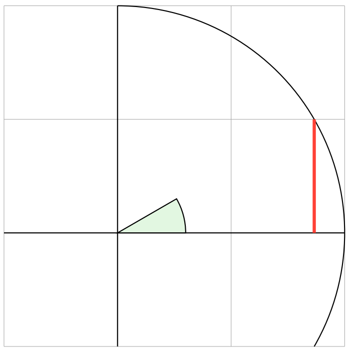
Specifying Coordinates

Karl now wants to add the sine and cosine lines. He knows already that he can use the `stroke:` styling parameter 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 unit length in the x-direction and twice the unit length in the y-direction". The unit length defaults to 1cm in both directions.

In order to specify points in polar coordinates, use an `array` of the form `(30deg, 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 wrap a coordinate in a `dictionary` with the key `rel` as in `(rel: (0cm, 1cm))`. Such coordinates are interpreted differently: They mean "1cm upwards from the previous specified position, making this the new specified position". You can include the key and value `update: false` in the coordinate `(rel: (2cm, 0cm), update: false)` which means "2cm to the right of the previous specified position and do not change the previous position". For example, we an draw the sine line as follows:



```
#cetz.canvas({
  import cetz.draw: *
  set-style(
    stroke: 0.4pt,
    grid: (
      stroke: gray + 0.2pt,
      step: 0.5
    )
  )
  scale(3)
  grid((-0.5, -0.5), (1, 1))
  line((-0.5, 0), (1, 0))
  line((0, -0.5), (0, 1))
  arc((0, 1), start: 90deg, delta: -120deg)
  arc(
    (3mm, 0),
    start: 0deg,
    stop: 30deg,
    radius: 3mm,
    mode: "PIE",
    fill: color.mix((green, 20%), white),
  )
  // draw the sin 30° line
  line((30deg, 1cm), (rel: (0, -0.5)), stroke: red
+ 1.2pt)
})
```

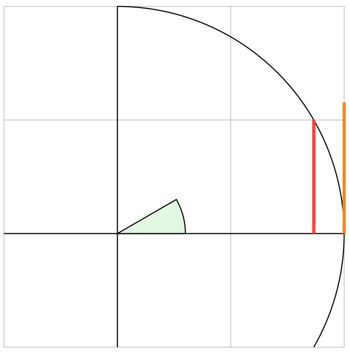
Note that an empty array `()` is given as the line's starting coordinate. This value means "the previous specified coordinate", which in this case is the end of the cosine line.

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, calc.tan(30deg))`.

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 `(30deg, 1cm)`. Such computations are made available by the `intersections` function.

What Karl must do is to create two “invisible” paths that intersect at the position of interest. Creating lines that are not otherwise seen can be done by either setting their stroke to `none` or by wrapping them in the `hide` function. Then, Karl can add the `name` parameter to the lines for later reference. Once the lines have been constructed, Karl can use the `intersections` function to get the coordinate for later reference.



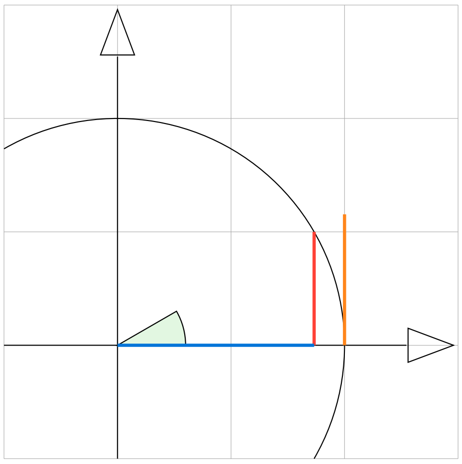
```
  #cetz.canvas({
    import cetz.draw: *
    set-style(
      stroke: 0.4pt,
      grid: (
        stroke: gray + 0.2pt,
        step: 0.5
      )
    )
    scale(3)
    grid((-0.5, -0.5), (1, 1))
    line((-0.5, 0), (1, 0))
    line((0, -0.5), (0, 1))
    arc((0, 1), start: 90deg, delta: -120deg)
    arc(
      (3mm, 0),
      start: 0deg,
      stop: 30deg,
      radius: 3mm,
      mode: "PIE",
      fill: color.mix((green, 20%), white),
    )
    line((30deg, 1cm), (rel: (0, -0.5)), stroke: red + 1.2pt)
    hide({
      line((1, 0), (1, 1), name: "upward line")
      line((0, 0), (30deg, 1.5cm), name: "sloped line") // a bit longer, so that
there is an intersection
    })

    intersections("x", "upward line", "sloped line")
    line((1, 0), "x.0", stroke: orange + 1.2pt)
  })
```

Adding Marks

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 `mark` styling parameter with the value `(end: ">")` to the line functions for the axes:



```
  #cetz.canvas({
    import cetz.draw: *
    set-style(
      stroke: 0.4pt,
      grid: (
        stroke: gray + 0.2pt,
        step: 0.5
      )
    )
    scale(3)
    grid((-0.5, -0.5), (1.5, 1.5))
    line((-0.5, 0), (1.5, 0), mark: (end:
">"))
    line((0, -0.5), (0, 1.5), mark: (end:
">"))
    arc((0, 0), start: 120deg, stop: -30deg,
anchor: "origin")
    arc(
      (3mm, 0),
      start: 0deg,
      stop: 30deg,
      radius: 3mm,
      mode: "PIE",
      fill: color.mix((green, 20%), white),
    )
    line((30deg, 1cm), (rel: (0, -0.5)),
stroke: red + 1.2pt)
    line((), (0, 0), stroke: blue + 1.2pt)

    hide({
      line((1, 0), (1, 1), name: "upward
line")
      line((0, 0), (30deg, 1.5cm), name:
"sloped line")
    })
    intersections("x", "upward line",
"sloped line")
    line((1, 0), "x.0", stroke: orange +
1.2pt)
  })
```

If Karl had used the value `(start: ">")` instead of `(end: ">")`, arrow tips would have been put at the beginning of the path. The value `(start: ">", end: ">")` or `(symbol: ">")` puts the marks at both ends of the path.

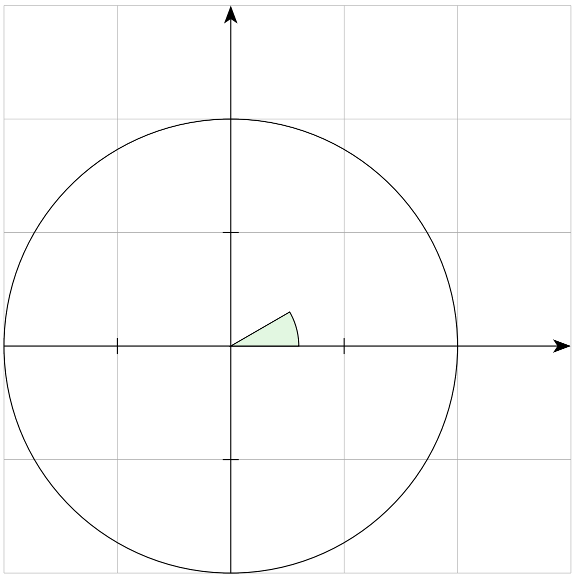
Some elements do not support marks. As a rule of thumb, you can add marks only to elements that do not have a closed path.

Karl notices that the marks are unnaturally large, this is because the shape of marks are transformed by default. So the marks are three times as big. This can be resolved by setting the `transform-shape: false` styling parameter. He also wants the mark to be filled and to have a different shape. As done earlier, the `fill` styling parameter can be used to fill marks and a different shape can be obtained by referring to the table of currently supported marks.

Repeating Things: For-Loops

Karl’s next aim is to add little ticks on the axes at positions -1, $-\frac{1}{2}$, $\frac{1}{2}$, 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. Thankfully Typst has built in loops.

Karl can then use the following code:



```
  #cetz.canvas({
    import cetz.draw: *
    set-style(
      stroke: 0.4pt,
      grid: (
        stroke: gray + 0.2pt,
        step: 0.5
      ),
      mark: (
        transform-shape: false,
        fill: black
      )
    )
    scale(3)
    grid((-1, -1), (1.5, 1.5))
    line((-1, 0), (1.5, 0), mark:
(end: "stealth"))
    line((0, -1), (0, 1.5), mark:
(end: "stealth"))
    circle((0, 0))
    arc(
      (3mm, 0),
      start: 0deg,
      stop: 30deg,
      radius: 3mm,
      mode: "PIE",
      fill: color.mix((green, 20%),
white),
    )

    for x in (-1, -0.5, 0.5, 1) {
      line((x, -1pt), (x, 1pt))
    }
    for y in (-1, -0.5, 0.5, 1) {
      line((-1pt, y), (1pt, y))
    }
  })
```

Adding Text

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

CeTZ has the `content` draw function which allows you to place any content onto the canvas at a specified position. Refer to here for more details.