1 Function manipulations

Thinking of functions as objects themselves that can be manipulated - rather than just blackboxes for evaluation - is a major abstraction of calculus. The main operations to come: the limit *of a function*, the derivative *of a function*, and the integral *of a function* all operate on functions. Hence the idea of an operator. Here we discuss manipulations of functions from pre-calculus that have proven to be useful abstractions.

1.1 The algebra of functions

We can talk about the algebra of functions. For example, the sum of functions f and g would be a function whose value at x was just $f(x) + g(x)$. More formally, we would have:

$$(f + g)(x) = f(x) + g(x),$$

We have given meaning to a new function $f + g$ by defining what is does to x with the rule on the right hand side. Similarly, we can define operations for subtraction, multiplication, addition, and powers. These aren't defined in base `Julia`, though they could be if desired, by a commands such as:

```
import Base: +
f::Function + g::Function = x -> f(x) + g(x)
```

```
+ (generic function with 519 methods)
```

This adds a method to the generic `+` function for functions. The type annotations `::Function` ensure this applies only to functions. To see that it would work, we could do odd-looking things like:

```
(sin + sqrt)(4)
```

1 . 2 4 3 1 9 7 5 0 4 6 9 2 0 7 1 8

Doing this works, as Julia treats functions as first class objects, lending itself to higher order programming. However, this definition in general is kind of limiting, as functions in mathematics and Julia can be much more varied than just the univariate functions we have defined addition for. We won't pursue this further.

1.2 Composition of functions

As seen, just like with numbers, it can make sense mathematically to define addition, subtraction, multiplication and division of functions. Unlike numbers though, we can also define a new operation on functions called **composition** that involves chaining the output of one function to the input of another. Composition is a common practice in life, where the result of some act is fed into another process. For example, making a pie from scratch involves first making a crust, then composing this with a filling. A better abstraction might be how

we "surf" the web. The output of one search leads us to another search whose output then is a composition.

Mathematically, a composition of univariate functions f and g is written $f \circ g$ and defined by what it does to a value in the domain of g by:

$$(f \circ g)(x) = f(g(x)).$$

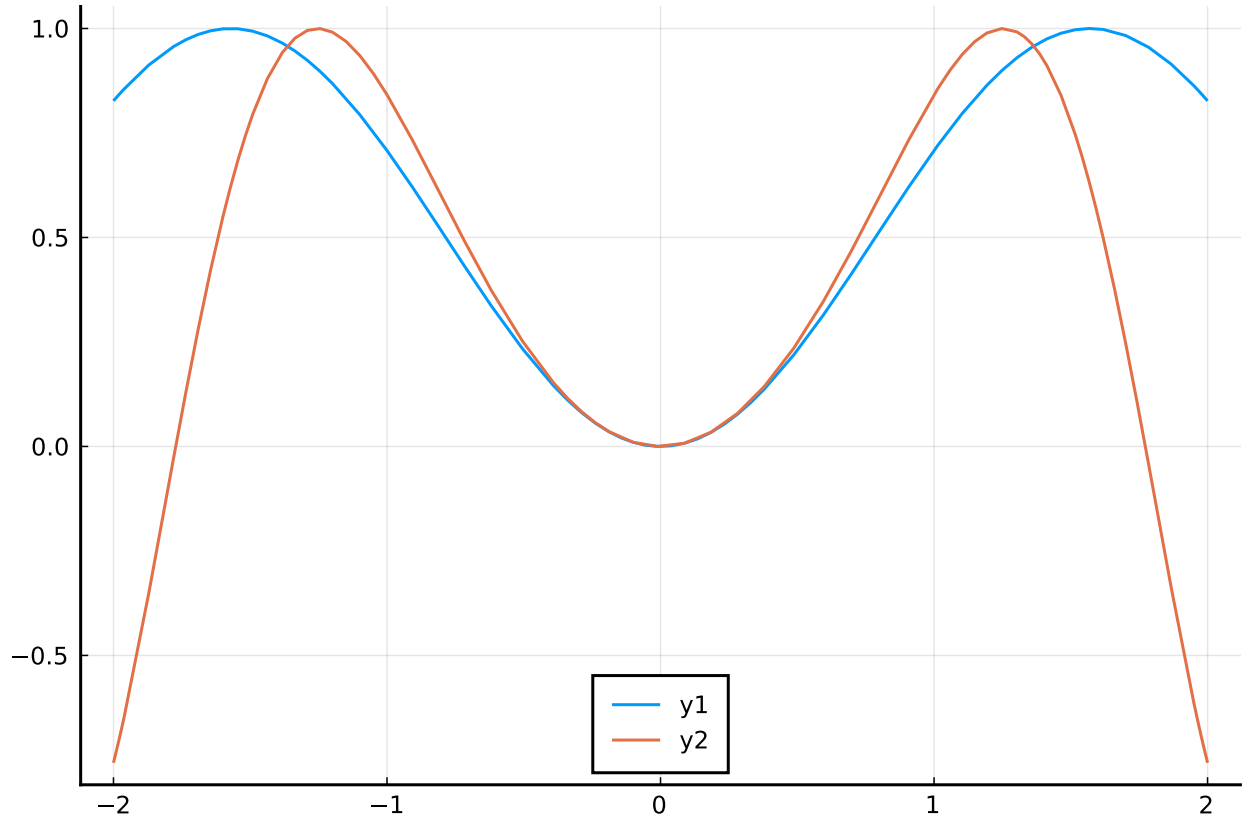
The output of g becomes the input of f.

Composition depends on the order of things. There is no guarantee that $f \circ g$ should be the same as $g \circ f$. (Putting on socks then shoes is quite different from putting on shoes then socks.) Mathematically, we can see this quite clearly with the functions $f(x) = x^2$ and $g(x) = \sin(x)$. Algebraically we have:

$$(f \circ g)(x) = \sin(x)^2, \quad (g \circ f)(x) = \sin(x^2).$$

Though they may be *typographically* similar don't be fooled, the following graph shows that the two functions aren't even close except for x near 0 (for example, one composition is always non-negative, whereas the other is not):

```
using CalculusWithJulia
using Plots
f(x) = x^2
g(x) = sin(x)
fg = f ∘ g       # typed as f \circ[tab] g
gf = g ∘ f       # typed as g \circ[tab] f
plot(fg, -2, 2)
plot!(gf, -2, 2)
```



Unlike how the basic arithmetic operations are treated, `Julia` defines the infix Unicode operator `\circ[tab]` to represent composition of functions, mirroring mathematical notation. It's use with generic functions is a bit cumbersome, as the operation returns an anonymous function. However, it can be useful and will mirror standard mathematical usage up to issues with precedence rules.)

Starting with two functions and composing them requires nothing more than a solid grasp of knowing the rules of function evaluation. If $f(x)$ is defined by some rule involving x, then $f(g(x))$ just replaces each x in the rule with a $g(x)$.

So if $f(x) = x^2 + 2x - 1$ and $g(x) = e^x - x$ then $f \circ g$ would be (before any simplification)

$$(f \circ g)(x) = (e^x - x)^2 + 2(e^x - x) - 1.$$

Here we look at a few compositions:

- The function $h(x) = \sqrt{1 - x^2}$ can be seen as $f \circ g$ with $f(x) = \sqrt{x}$ and $g(x) = 1 - x^2$.
- The function $h(x) = \sin(x/3 + x^2)$ can be viewed as $f \circ g$ with $f(x) = \sin(x)$ and $g(x) = x/3 + x^2$.
- The function $h(x) = e^{-1/2 \cdot x^2}$ can be viewed as $f \circ g$ with $f(x) = e^{-x}$ and $g(x) = (1/2) \cdot x^2$.

Decomposing a function into a composition of functions is not unique, other compositions could have been given above. For example, the last function is also $f(x) = e^{-x/2}$ composed with $g(x) = x^2$.

The real value of composition is to break down more complicated things into a sequence of easier steps. This is good mathematics, but also good practice more generally. For example, when we approach a problem with the computer, we generally use a smallish set of functions and piece them together (that is, compose them) to find a solution.

1.3 Shifting and scaling graphs

It is very useful to mentally categorize functions within families. The difference between $f(x) = \cos(x)$ and $g(x) = 12\cos(2(x - \pi/4))$ is not that much - both are cosine functions, one is just a simple enough transformation of the other. As such, we expect bounded, oscillatory behaviour with the details of how large and how fast the oscillations are to depend on the specifics of function. Similarly, both these functions $f(x) = 2^x$ and $g(x) = e^x$ behave like exponential growth, the difference being only in the rate of growth. There are families of functions that are qualitatively similar, but quantitatively different, linked together by a few basic transformations.

There is a set of operations of functions, which does not really change the type of function. Rather, it basically moves and stretches how the functions are graphed. We discuss these four main transformations of f:

The functions h are derived from f in a predictable way. To implement these transformations within `Julia`, we define operators (functions which transform one function into another). As

Transformation	Description
vertical shifts	The function $h(x) = k + f(x)$ will have the same graph as f shifted up by k units.
horizontal shifts	The function $h(x) = f(x - k)$ will have the same graph as f shifted right by k units.
stretching	The function $h(x) = kf(x)$ will have the same graph as f stretched by a factor of k in the y direction.
scaling	The function $h(x) = f(kx)$ will have the same graph as f compressed horizontally by a factor of 1 over k.

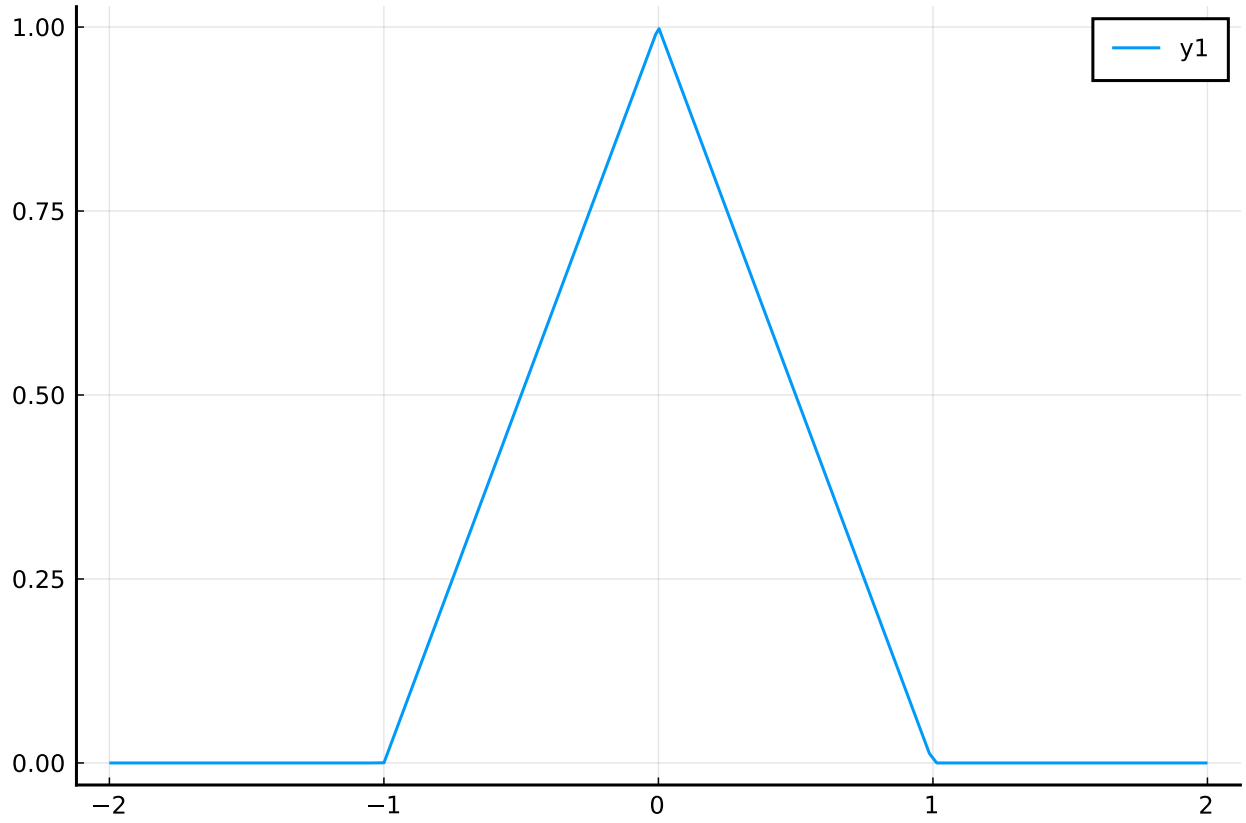
these return functions, the function bodies are anonymous functions. The basic definitions are similar, save for the `x -> ...` part that signals the creation of an anonymous function to return:

```
up(f, k)       = x -> f(x) + k
over(f, k)     = x -> f(x - k)
stretch(f, k)  = x -> k * f(x)
scale(f, k)    = x -> f(k * x)
```

```
scale (generic function with 1 method)
```

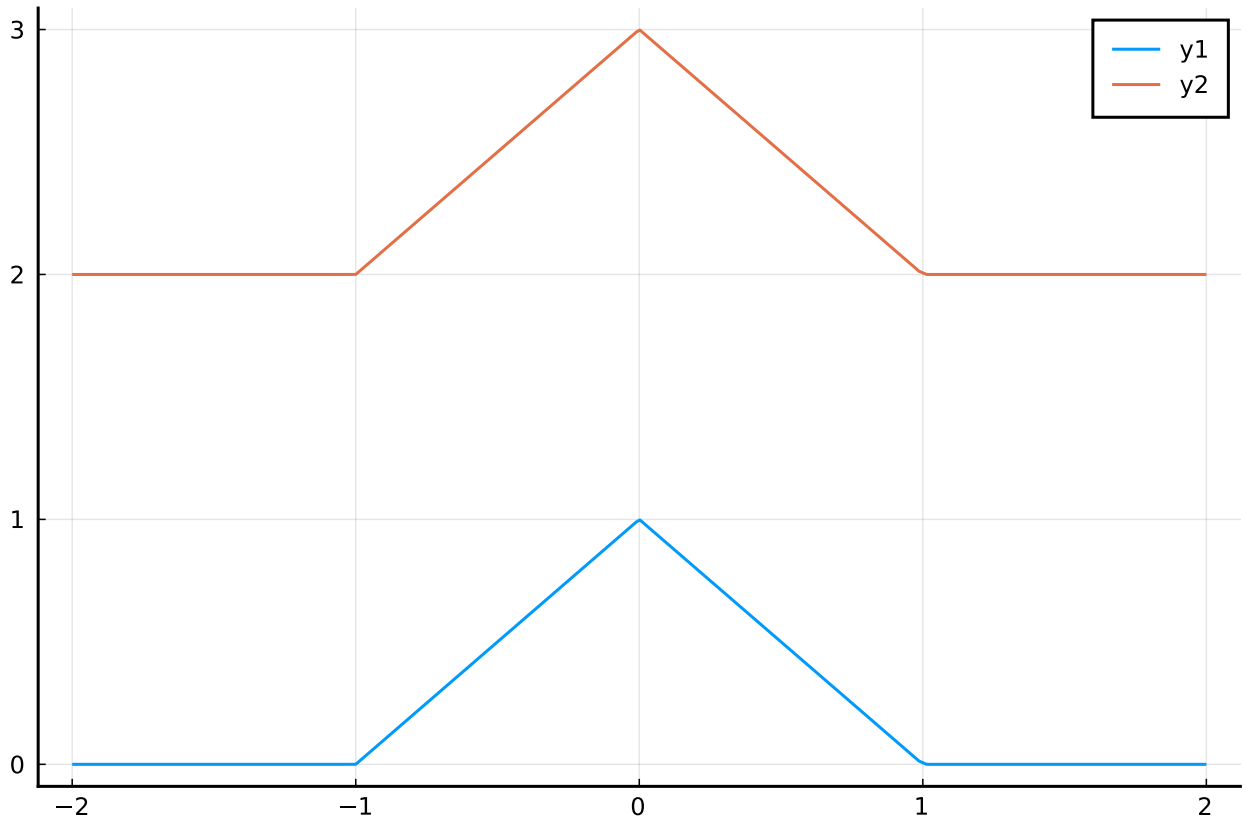
To illustrate, let's define a hat-shaped function as follows:

```
f(x) = max(0, 1 - abs(x))
plot(f, -2,2)
```



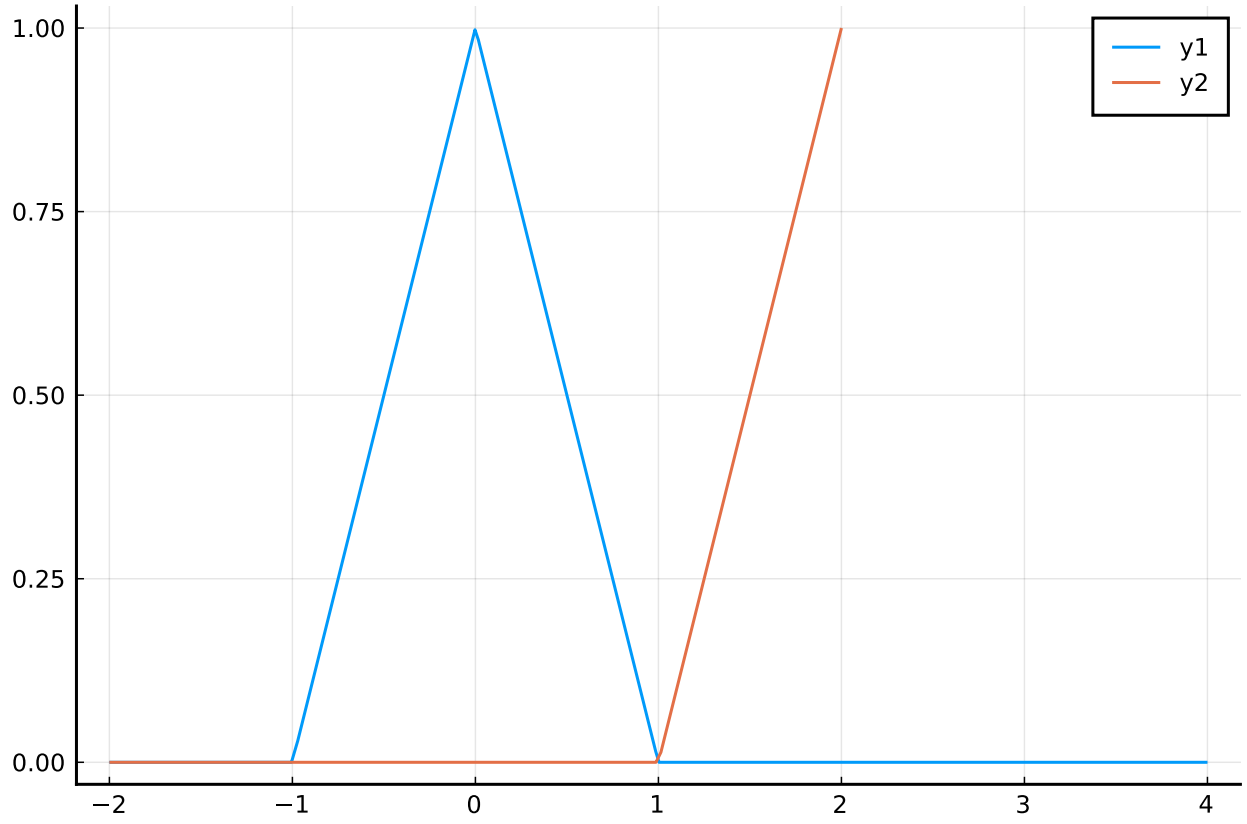
The same graph of f and its image shifted up by 2 units would be given by:

```
plot(f, -2, 2)
plot!(up(f, 2), -2, 2)
```



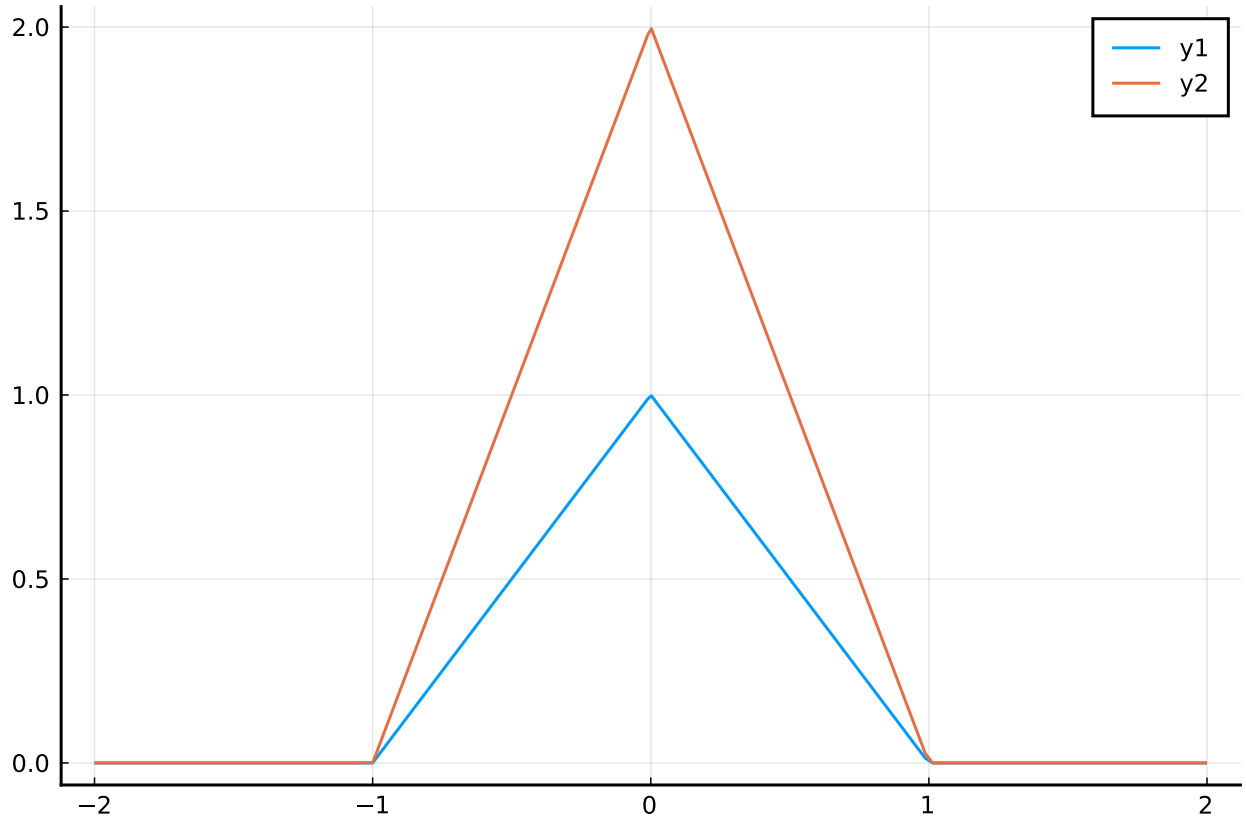
A graph of f and its shift over by 2 units would be given by:

```
plot(f, -2, 4)
plot!(over(f, 2), -2, 2)
```



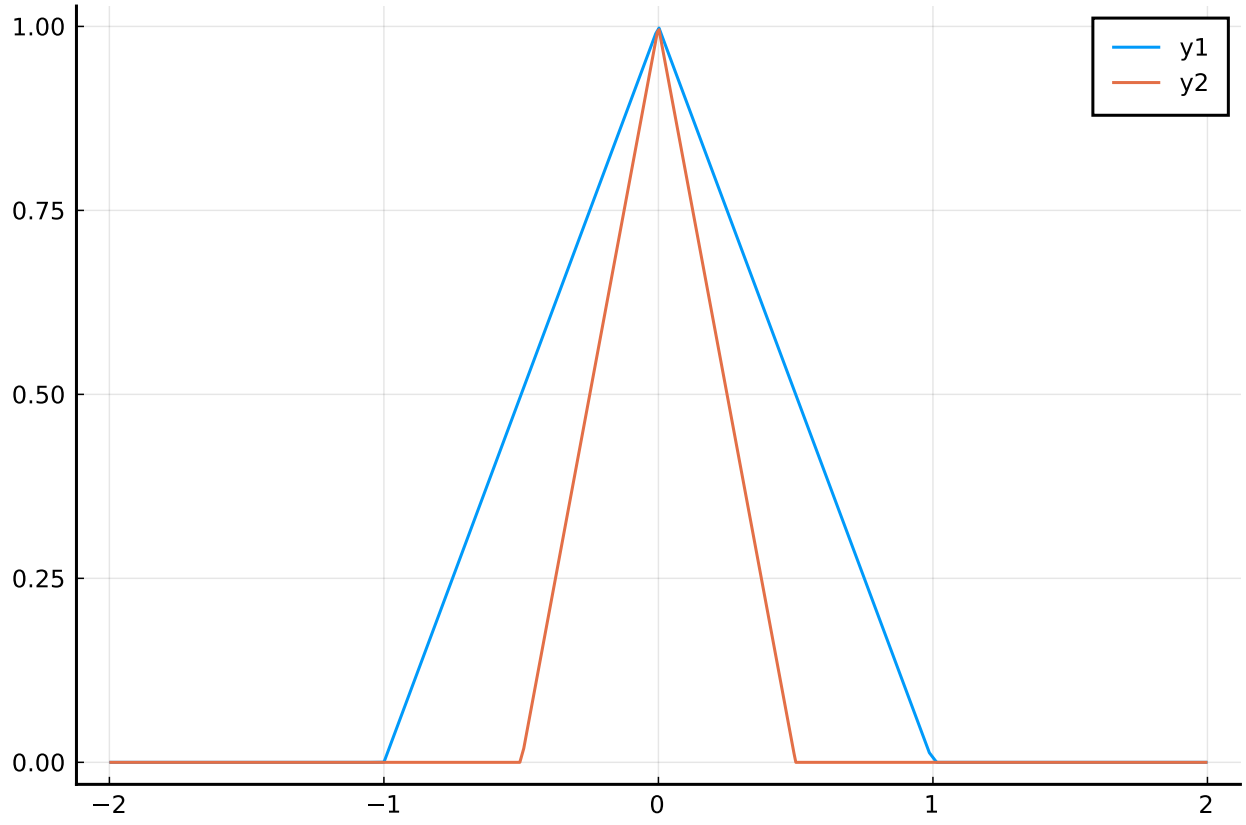
A graph of f and it being stretched by 2 units would be given by:

```
plot(f, -2, 2)
plot!(stretch(f, 2), -2, 2)
```



Finally, a graph of f and it being scaled by 2 would be given by:

```
plot(f, -2, 2)
plot!(scale(f, 2), -2, 2)
```

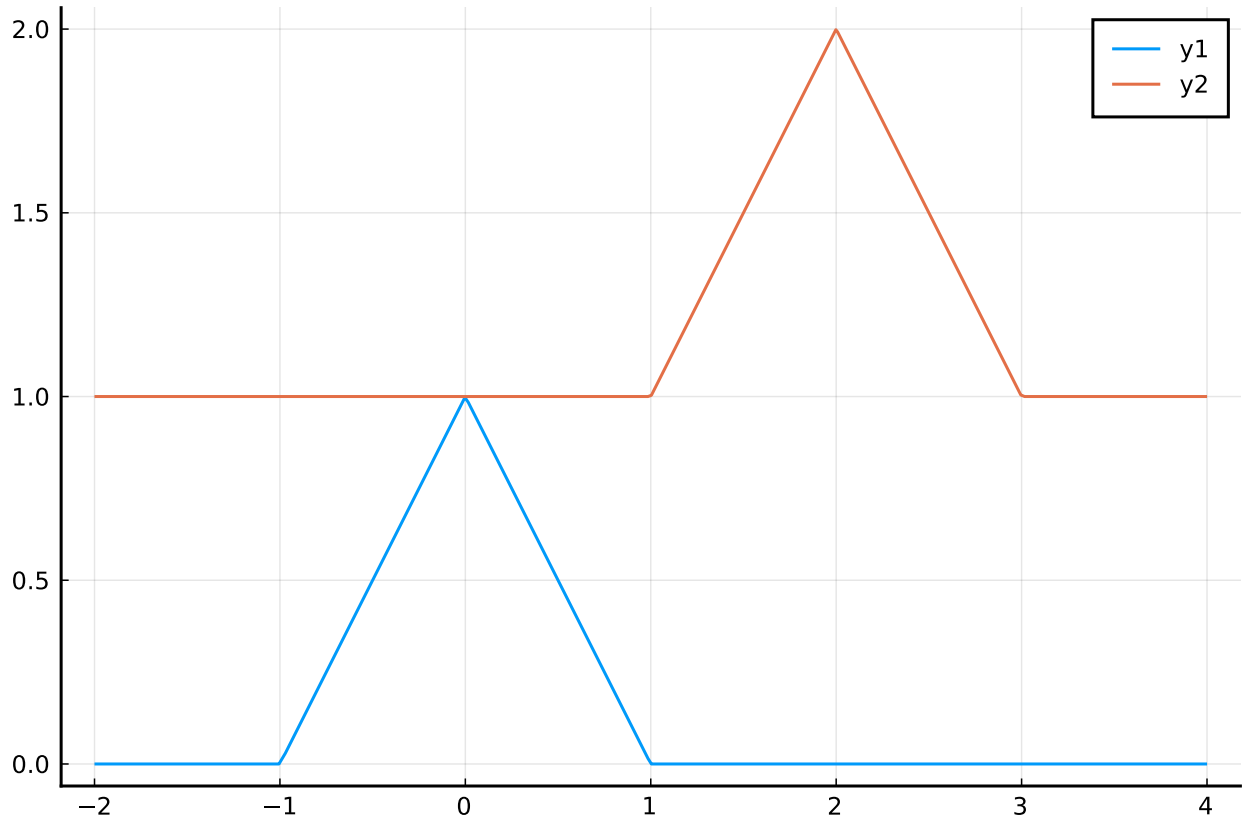


Scaling by 2 shrinks the non-zero domain, scaling by 1/2 would stretch it. If this is not intuitive, the defintion `x-> f(x/c)` could have been used, which would have opposite behaviour for scaling.

More exciting is what happens if we combine these operations.

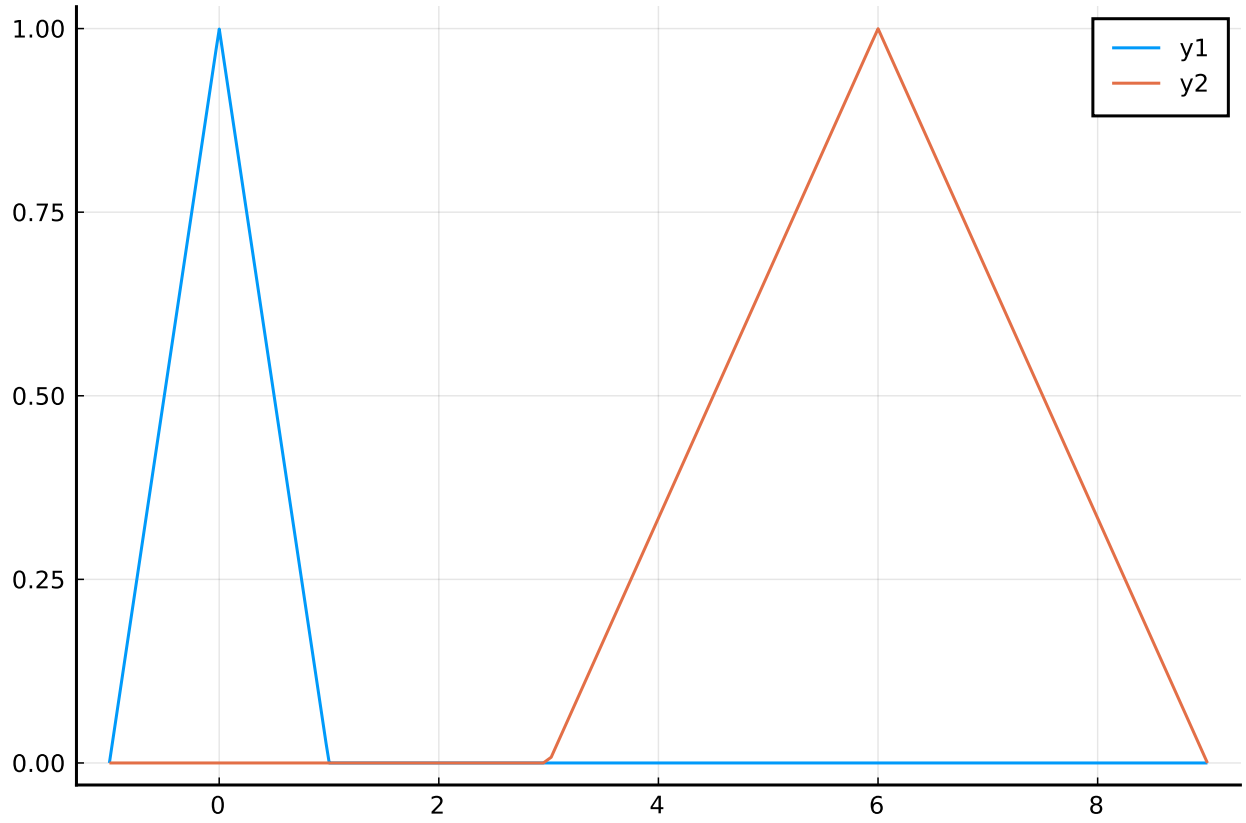
A shift right by 2 and up by 1 is achieved through

```
plot(f, -2, 4)
plot!(up(over(f,2), 1), -2, 4)
```



Shifting and scaling can be confusing. Here we graph `scale(over(f,2),1/3)`:

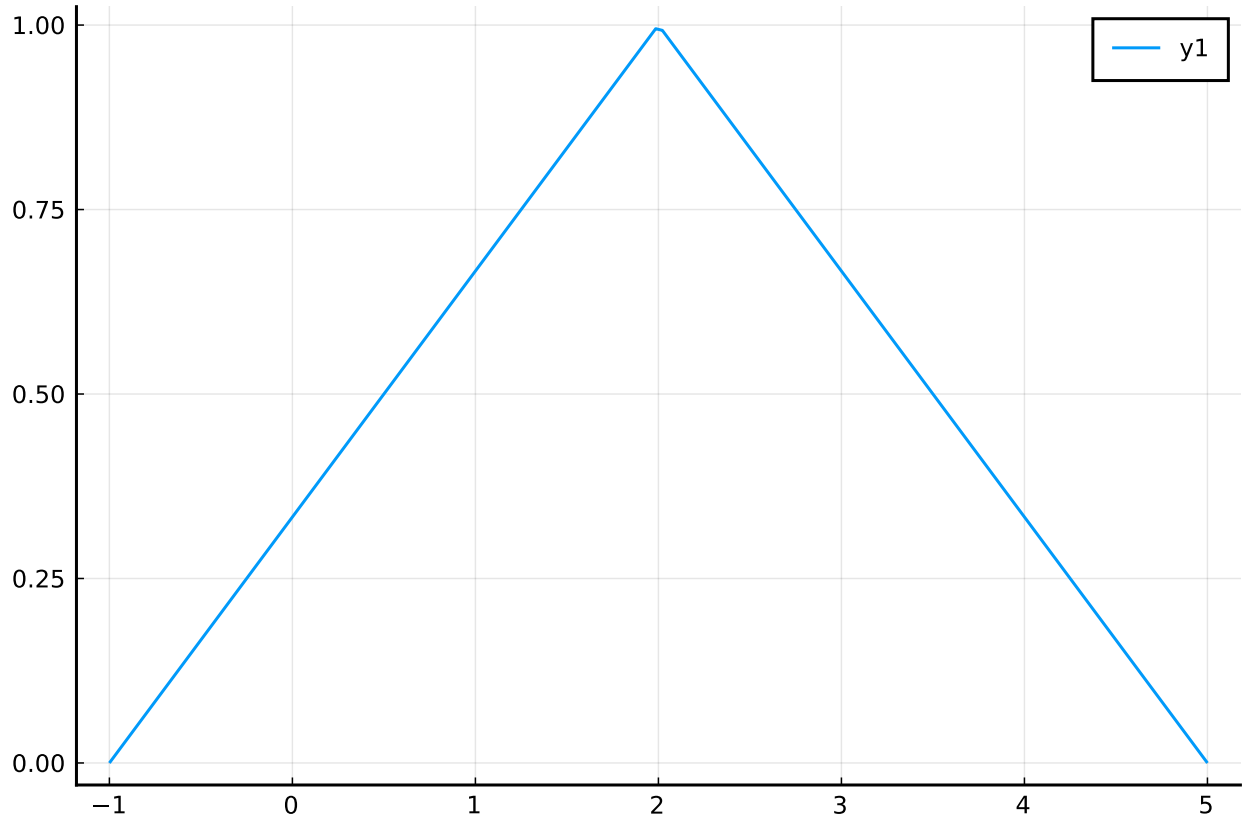
```
plot(f, -1,9)
plot!(scale(over(f,2), 1/3), -1,9)
```



This graph is over by 6 with a width of 3 on each side of the center. Mathematically, we have $h(x) = f((1/3) \cdot x - 2)$

Compare this to the same operations in opposite order:

```
plot(f, -1, 5)
plot(over(scale(f, 1/3), 2), -1, 5)
```



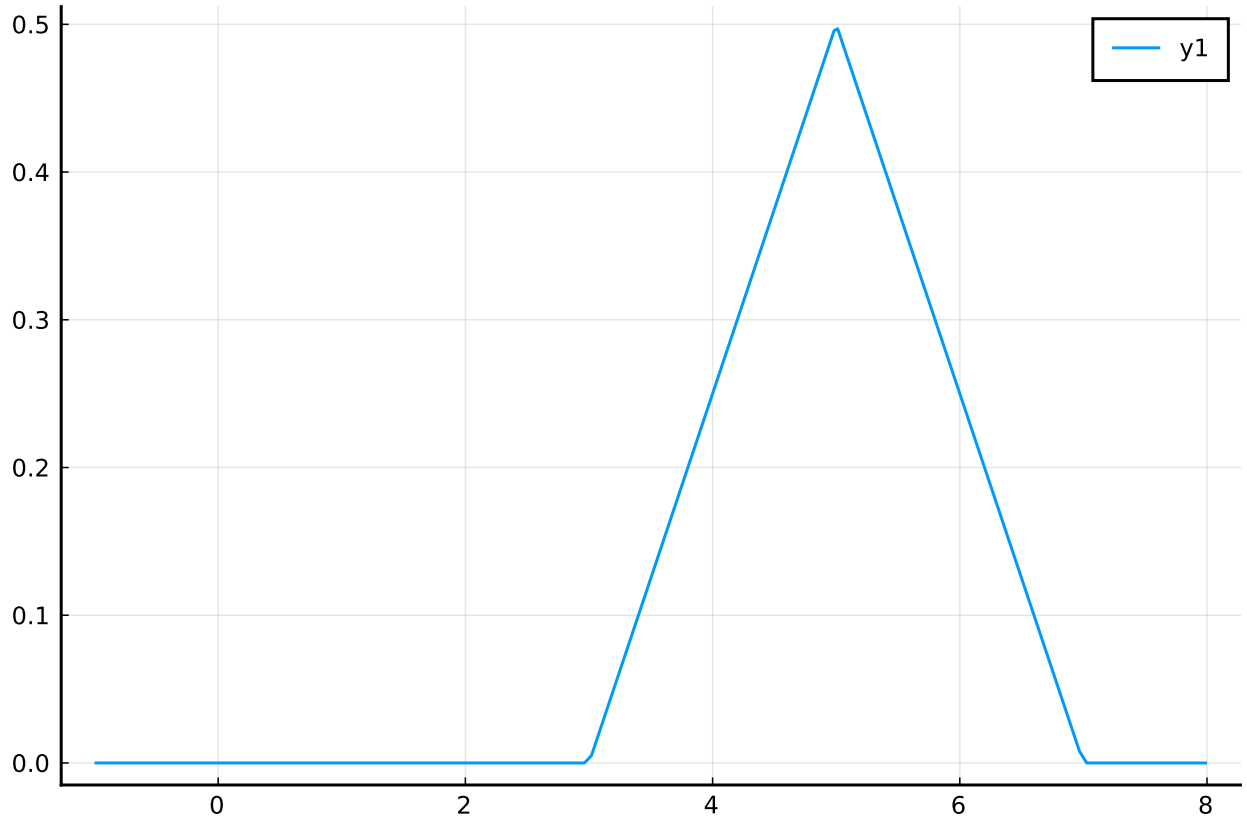
This graph first scales the symmetric graph, stretching from -3 to 3, then shifts that right by 2. The resulting function is $f((1/3)\cdot(x-2))$.

As a last example, following up on the last example, a common transformation mathematically is

$$h(x) = \frac{1}{a} f(\frac{x-b}{a}).$$

We can view this as a composition of "scale" by $1/a$, then "over" by b, and finally "stretch" by $1/a$:

```
a = 2; b = 5
h(x) = stretch(over(scale(f, 1/a), b), 1/a)(x)
plot(f, -1, 8)
plot(h, -1, 8)
```



(This transformation keeps the same amount of area in the triangles, can you tell from the graph?)

Example A model for the length of a day in New York City must take into account periodic seasonal effects. A simple model might be a sine curve. However, there would need to be many modifications: Obvious ones would be that the period would need to be about 365 days, the oscillation around 12 and the amplitude of the oscillations no more than 12.

We can be more precise. According to dateandtime.info in 2015 the longest day will be June 21st when there will be 15h 5m 46s of sunlight, the shortest day will be December 21st when there will be 9h 15m 19s of sunlight. On January 1, there will be 9h 19m 44s of sunlight.

A model for a transformed sine curve is

$$a + b\sin(d(x - c))$$

Where b is related to the amplitude, c the shift and the period is $T = 2\pi/d$. We can find some of these easily from the above:

```
a = 12
b = ((15 + 5/60 + 46/60/60) - (9 + 19/60 + 44/60/60)) / 2
d = 2pi/365
```

0 . 0 1 7 2 1 4 2 0 6 3 2 1 0 3 9 9 6

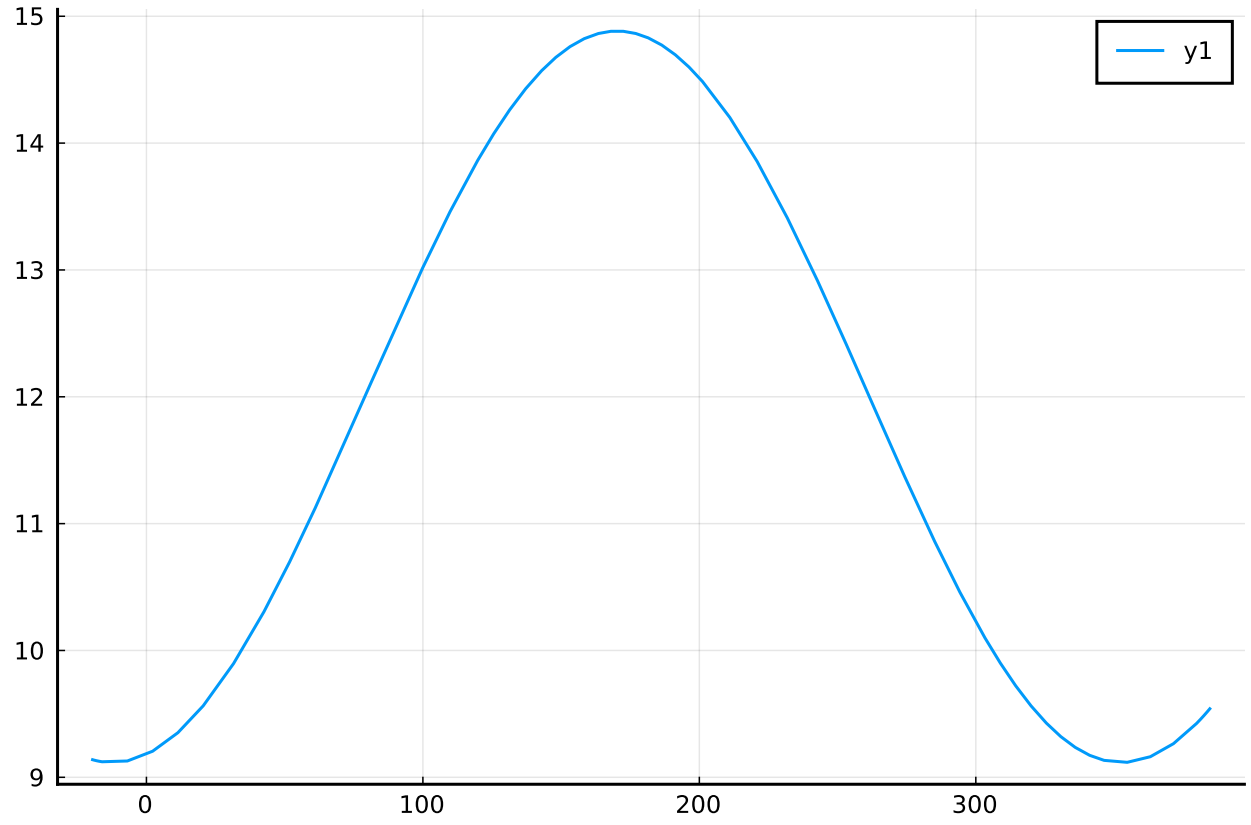
If we let January 1 be $x = 0$ then the first day of spring, March 21, is day 79 (`Date(2017, 3, 21) - Date(2017, 1, 1)`). This day aligns with the shift of the sine curve. This shift is 79:

```
c = 79
```

79

Putting this together, we have our graph is "scaled" by d, "over" by c, "stretched" by b and "up" by a. Here we plot it over slightly more than one year so that we can see that the shortest day of light is in late December ($x \approx -10$ or $x \approx 355$).

```
newyork(t) = up(stretch(over(scale(sin, d), c), b), a)(t)
plot(newyork, -20, 385)
```



To test, if we match up with the model powering dateandtime.info we note that it predicts "15h 0m 4s" on July 4, 2015. This is day 184 (`Date(2015, 7, 4) - Date(2015, 1, 1)`). Our model prediction has a difference of

```
datetime = 15 + 0/60 + 4/60/60
delta = (newyork(184) - datetime) * 60
```

-11.874016689526263

This is off by a fair amount - almost 12 minutes. Clearly a trigonometric model, based on the assumption of circular motion of the earth around the sun, is not accurate enough for precise work.

Example: the pipeline operator In the last example, we described our sequence as scale, over, stretch, and up, but code this in reverse order, as the composition $f \circ g$ is done

from right to left. A more convenient notation would be to have syntax that allows the composition of g then f to be written $x \to g \to f$. `Julia` provides the pipeline operator for chaining function calls together.

For example, if $g(x) = \sqrt{x}$ and $f(x) = \sin(x)$ we could call $f(g(x))$ through:

```
g(x) = sqrt(x)
f(x) = sin(x)
pi/2 |> g |> f
```

0 . 9 5 0 0 2 4 4 2 7 4 6 5 7 8 3 4

The output of the preceding expression is passed as the input to the next. This notation is especially convenient when the enclosing function is not the main focus. (Some programming languages have more developed fluent interfaces for chaining function calls. Julia has some macros provided in packages.)

Example: a growth model in fisheries The von Bertanlaffy growth equation is $L(t) = L_\infty \cdot (1 - e^{k\cdot(t-t_0)})$. This family of functions can be viewed as a transformation of the exponential function $f(t) = e^t$. Part is a scaling and shifting (the $e^{k\cdot(t-t_0)}$) along with some shifting and stretching. The various parameters have physical importance which can be measured: L_∞ is a carrying capacity for the species or organism, and k is a rate of growth. These parameters may be estimated from data by finding the "closest" curve to a given data set.

1.4 Operators

The functions `up`, `over`, etc. are operators that take a function as an argument and return a function. The use of operators fits in with the template `action(f, args...)`. The action is what we are doing, such as `plot`, `over`, and others to come. The function `f` here is just an object that we are performing the action on. For example, a plot takes a function and renders a graph using the additional arguments to select the domain to view, etc.

Creating operators that return functions involves the use of anonymous functions, using these operators is relatively straightforward. Two basic patterns are

- Storing the returned function, then calling it:

```
l(x) = action1(f, args...)(x)
l(10)
```

- Composing two operators:

```
action2( action1(f, args..), other_args...)
```

Composition like the above is convenient, but can get confusing if more than one composition is involved.

Example: two operators (See Krill for background on this example.) Consider two operations on functions. The first takes the *difference* between adjacent points. We call this `D`:

```
D(f::Function) = k -> f(k) - f(k-1)
```

```
D (generic function with 1 method)
```

To see that it works, we take a typical function and check:

```
f(k) = 1 + k^2
D(f)(3), f(3) - f(3-1)
```

```
(5, 5)
```

That the two are the same value is no coincidence. (Again, pause for a second to make sure you understand why `D(f)(3)` makes sense. If this is unclear, you could name the function `D(f)` and then call this with a value of `3`.)

Now we want a function to cumulatively *sum* the values $S(f)(k) = f(1) + f(2) + \cdots + f(k-1) + f(k)$, as a function of k. Adding up k terms is easy to do with a generator and the function `sum` to add a vector of numbers:

```
S(f) = k -> sum(f(i) for i in 1:k)
```

```
S (generic function with 1 method)
```

To check if this works as expected, compare these two values:

```
S(f)(4), f(1) + f(2) + f(3) + f(4)
```

```
(34, 34)
```

So one function adds, the other subtracts. Addition and subtraction are somehow inverse to each other so should ”cancel” out. This holds for these two operations as well, in the following sense: subtracting after adding leaves the function alone:

```
k = 10    # some arbitrary value k >= 1
D(S(f))(k), f(k)
```

```
(101, 101)
```

Any positive integer value of `k` will give the same answer (up to overflow). This says the difference of the accumulation process is just the last value to accumulate.

Adding after subtracting also leaves the function alone, save for a vestige of $f(0)$:

```
k=15
S(D(f))(k),  f(k) - f(0)
```

```
(225, 225)
```

That is the accumulation of differences is just the difference of the end values.

These two operations are discrete versions of the two main operations of calculus - the derivative and the integral. This relationship will be known as the "fundamental theorem of calculus."

1.5 Questions

⊛ Question
If $f(x) = 1/x$ and $g(x) = x - 2$, what is $g(f(x))$?

1. $$-2$$
2. $$x - 2$$
3. $$1/(x - 2)$$
4. $$1/x - 2$$

⊛ Question
If $f(x) = e^{-x}$ and $g(x) = x^2$ and $h(x) = x - 3$, what is $f \circ g \circ h$?

1. $$(e^x - 3)^2$$
2. $$e^{-(x-3)^2}$$
3. $$e^{-x^2-3}$$
4. $$e^x + x^2 + x - 3$$

⊛ Question
If $h(x) = (f \circ g)(x) = \sin^2(x)$ which is a possibility for f and g:

1. $$f(x) = x^2; \quad g(x) = \sin^2(x)$$
2. $$f(x) = \sin(x); \quad g(x) = x^2$$
3. $$f(x) = x^2; \quad g(x) = \sin(x)$$

⊛ Question
Which function would have the same graph as the sine curve shifted over by 4 and up by 6?

1. $$h(x) = 6\sin(x - 4)$$
2. $$h(x) = 4 + \sin(6x)$$
3. $$h(x) = 6 + \sin(x + 4)$$
4. $$h(x) = 6 + \sin(x - 4)$$

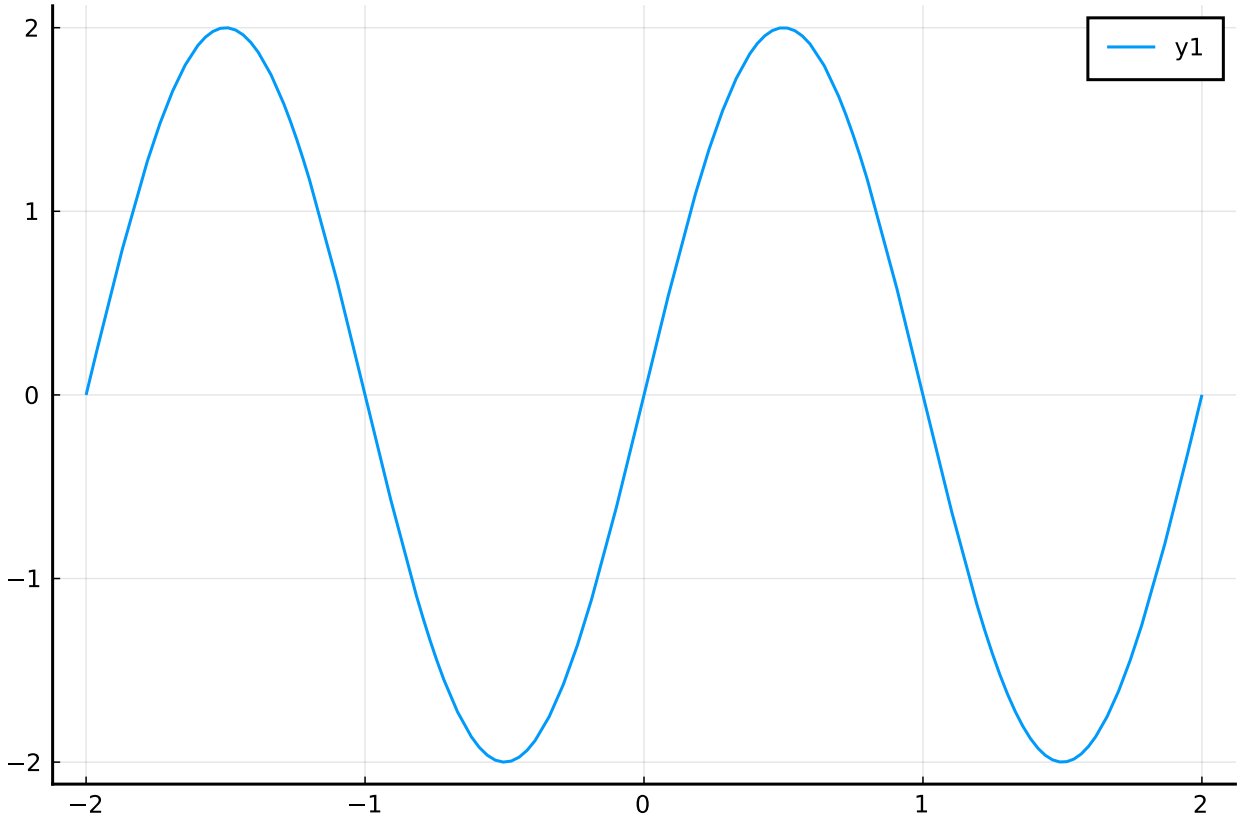
⊛ Question
Let $h(x) = 4x^2$ and $f(x) = x^2$. Which is **not** true:

```
Error: MethodError: no method matching latexinline(::Base.GenericIOBuffer{A
rray{UInt8,1}}, ::Symbol)
Closest candidates are:
  latexinline(::IO, !Matched::Markdown.LaTeX) at /Users/julia/buildbot/work
er/package_macos64/build/usr/share/julia/stdlib/v1.5/Markdown/src/IPython/I
Python.jl:31
  latexinline(::IO, !Matched::Markdown.Code) at /Users/julia/buildbot/worke
r/package_macos64/build/usr/share/julia/stdlib/v1.5/Markdown/src/render/lat
ex.jl:42
  latexinline(::IO, !Matched::Array{T,1} where T) at /Users/julia/buildbot/
worker/package_macos64/build/usr/share/julia/stdlib/v1.5/Markdown/src/rende
r/latex.jl:109
  ...
```

⊛ Question
The transformation $h(x) = (1/a) \cdot f((x - b)/a)$ can be viewed in one sequence:

1. scaling by $1/a$, then shifting by b, then stretching by $1/a$
2. shifting by a, then scaling by a, and then scaling by b
3. shifting by a, then scaling by b, and then scaling by $1/a$

⊛ Question
This is the graph of a transformed sine curve.



What is the period of the graph?

What is the amplitude of the graph?

What is the form of the function graphed?

1. $$2\sin(x)$$
2. $$2\sin(\pi x)$$
3. $$\sin(2x)$$
4. $$\sin(\pi x)$$

⊛ Question
Consider this expression

$$(f(1)-f(0))+(f(2)-f(1))+\cdots+(f(n)-f(n-1)) = -f(0)+f(1)-f(1)+f(2)-f(2)+\cdots+f(n-1)-f($$

Referring to the definitions of `D` and `S` in the example on operators, which relationship does this support:

1. `D(S(f))(n) = f(n)`
2. `S(D(f))(n) = f(n) - f(0)`

⊛ Question
Consider this expression:

$$(f(1) + f(2) + \cdots + f(n-1) + f(n)) - (f(1) + f(2) + \cdots + f(n-1)) = f(n).$$

Referring to the definitions of `D` and `S` in the example on operators, which relationship does this support:

1. `D(S(f))(n) = f(n)`
2. `S(D(f))(n) = f(n) - f(0)`