```
begin
    import Pkg
    # activate the shared project environment
    Pkg.activate(Base.current_project())
    using Omega, Distributions, OmegaExamples, UnicodePlots
end
```

```
  Activating project at `~/Documents/GitHub/Omega.jl/OmegaExamples`
```

Models, simulation, and degrees of belief

One view of knowledge is that the mind maintains working models of parts of the world. 'Model' in the sense that it captures some of the structure in the world, but not all (and what it captures need not be exactly what is in the world—just what is useful). 'Working' in the sense that it can be used to simulate this part of the world, imagining what will follow from different initial conditions.

Building Generative Models

We wish to describe in formal terms how to generate states of the world. That is, we wish to describe the causal process, or steps that unfold, leading to some potentially observable states. The key idea of this section is that these generative processes can be described as computations—computations that involve random choices to capture uncertainty about the process. Programming languages are formal systems for describing what (deterministic) computation a computer should do. Modern programming languages offer a wide variety of different ways to describe computation; each makes some processes simple to describe and others more complex. However, a key tenet of computer science is that all of these languages have the same fundamental power: any computation that can be described with one programming language can be described by another. (More technically this Church-Turing thesis posits that many specific computational systems capture the set of all effectively computable procedures. These are called universal systems.)

Omega is a programming language that describes probabilistic computation. The key idea is that in Omega you can express stochastic processes (random variables) and not just deterministic functions (like logical AND gate). Deterministic models return unique values for the same set of inputs whereas stochastic models have random variables as inputs, which result in random outputs. Most models we encounter are better described by random variables, for example, tossing a coin. We can simulate a coin toss in Omega, unlike in a deterministic programming language.

A basic object in Omega is a class representing random variables, from which you can generate random variables. For instance, a random variable that is of Bernoulli distribution is generated by -

```
a = -7986080874440649478@Distributions.Bernoulli{Float64}(p=0.5)
```

```
a = @~ Bernoulli()
```

Here, `Bernoulli()` is a random class and to get a member (a random variable, which we can sample from) of the class we use `@~`. The long-form of writing this expression is -

```
id = 0
```

```
id = 0
```

```
a_ = 0@Distributions.Bernoulli{Float64}(p=0.5)
```

```
a_ = id ~ Bernoulli()
```

where `0` is an ID, which is used to refer to the particular random variable in Omega. IDs may be tuples, numbers, strings or julia symbols. `@~` is a syntactic sugar that automatically selects an ID.

A class is analogous to a *plate* in Bayesian networks.

Sampling

`randsample` is used to sample from a random variable:

```
false
```

```
randsample(a)
```

Run this program a few times. You will get back a different sample on each execution.

To know how random variables are constructed, a key concept to know is the Ω object in Omega.jl.

There are a few perspectives on what Ω is:

1. The sample space in probability theory (often denoted Ω)
2. Space of exogenous variables in as causal graphical modelling nomenclature (often denoted U)
3. The random number generator `rng::Random.AbstractRNG`. Implementation wise, Omega.jl tries to be compatible with any model written as a function `f(rng::AbstractRNG)`

Random variables in Omega are functions of the form - `f(ω::Ω)` and `randsample` can used to sample from them.

Primitive and Composite Classes in Omega

Primitive Classes

Omega comes with a set of built-in primitive random variable classes, such as `StdNormal` and `StdUniform`. There are parameterless infinite sets of random variables.

```
As = StdUniform()
```

```
As = StdUniform{Float64}()
```

```
rv = 1@StdUniform{Float64}()
```

```
rv = 1 ~ As
```

```
0.499340108090767
```

```
randsample(rv)
```

Composite Class

A class in Omega is actually just a function of the form `f(id, ω)`. Of course, you can specify your own classes simply by constructing a function.

```
μ = 2@StdNormal{Float64}()
```

```
μ = 2 ~ StdNormal{Float64}()
```

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

```
x(id, ω) = (id ~ Normal(μ(ω), 1))(ω)
```

```
x_ = 3@x
```

```
x_ = 3 ~ x
```

`x_` is a random variable of the class `x`

```
(-0.0347971, 0.26251)
```

```
randsample((μ, x_))
```

Alternatively, `@joint` is used to create the joint distribution of variables:

```
joint = #7 (generic function with 1 method)
```

```
joint = @joint μ x_
```

```
(μ = -0.771133, x_ = -0.78307)
```

```
randsample(joint)
```

Every time we create a new (independent) random variable, we need to provide a unique id to it. We could instead use `@~` to generate `id`s automatically. For example, to simulate a (possibly biased) coin toss, we can simply write:

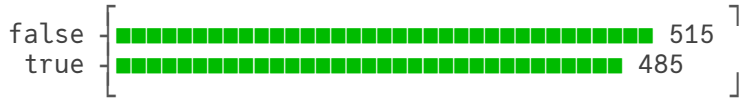
```
coin_toss = 2399916368207762249@Distributions.Bernoulli{Float64}(p=0.5)
```

```
coin_toss = @~ Bernoulli()
```

```
true
```

```
randsample(coin_toss)
```

If you run the program many times, and collect the values in a histogram, you can see what a typical sample looks like:




```
viz(randsample(coin_toss, 1000))
```

As you can see, the result is an approximately uniform distribution over true(1) and false(0). This way we can construct more complex expressions that describe more complicated random variables. For instance, here we describe a process that samples a number adding up several independent Bernoulli distributions:

```
b_sum =
+ₚ(+ₚ(-5568433208695521429@Distributions.Bernoulli{Float64}(p=0.5), 5186369825370610331@Dis
```

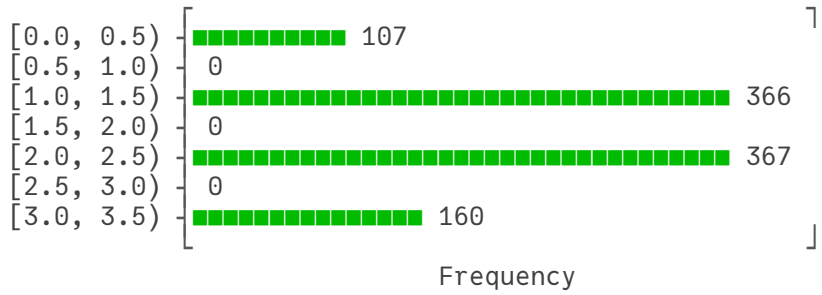
```
b_sum = (@~ Bernoulli()) .+ (@~ Bernoulli()) .+ (@~ Bernoulli())
```

`+ₚ` here is pointwise sum - the subscript p denotes the pointwise operation defined on a function

```
1
```

```
randsample(b_sum)
```

We have constructed a random variable that is a sum of three random variables and sampled from it. We can construct such complex random variables from the primitive ones.




```
viz(randsample(b_sum, 1000))
```

Complex functions can also have other arguments. Here is a random variable that will only sometimes double its input:

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

```
noisy_double(x) = ifelse.((@~ Bernoulli()), 2*x, x)
```

To perform pointwise operation, which is defined on the function `ifelse` here, we use `.` operator. This can also be done in the following way -

```
pw(ifelse, (@~ Bernoulli()), 2*x, x)
```

```
6
```

```
randsample(noisy_double(3))
```

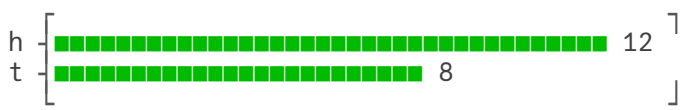
By using higher-order functions we can construct and manipulate complex sampling processes. We use the `ifelse` function: `ifelse.(condition, if-true, if-false)` to induce hierarchy. A good example comes from coin flipping...

Example: Flipping Coins

The following program defines a fair coin, and flips it 20 times:

```
fair_coin =
ifelseₚ(-7956923106182135926@Distributions.Bernoulli{Float64}(p=0.5), 'h', 't')
```

```
fair_coin = ifelse.((@~ Bernoulli()), 'h', 't')
```

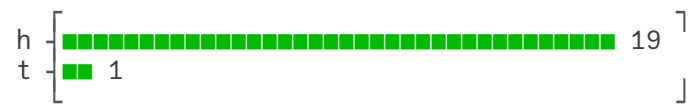



```
viz(randsample(fair_coin, 20))
```

This program defines a "trick" coin that comes up heads most of the time (95%), and flips it 20 times:

```
trick_coin =
ifelseₚ(-9095721144617615346@Distributions.Bernoulli{Float64}(p=0.95), 'h', 't')
```

```
trick_coin = ifelse.((@~ Bernoulli(0.95)), 'h', 't')
```




```
viz(randsample(trick_coin, 20))
```

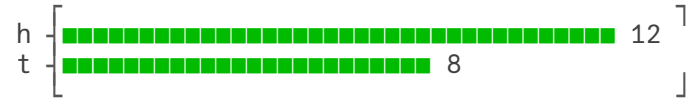
The higher-order function `make_coin` takes in a weight and outputs a function describing a coin with that weight. Then we can use `make_coin` to make the coins above, or others.

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

```
make_coin(weight) = ifelse.((@~ Bernoulli(weight)), 'h', 't')
```

```
ifelseₚ(-2806800054355869831@Distributions.Bernoulli{Float64}(p=0.25), 'h', 't')
```

```
begin
    fair_coin1 = make_coin(0.5)
    trick_coin1 = make_coin(0.95)
    bent_coin = make_coin(0.25)
end
```




```
viz(randsample(fair_coin1, 20))
```


```
viz(randsample(trick_coin1, 20))
```


```
viz(randsample(bent_coin, 20))
```

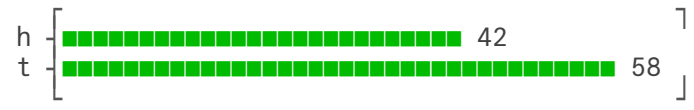
We can also define a higher-order function that takes a "coin" and "bends it":

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

```
bend(coin) = ifelse.((coin .== 'h'), make_coin(0.7), make_coin(0.1))
```

```
bent_coin1 =
ifelseₚ(==ₚ(ifelseₚ(-7956923106182135926@Distributions.Bernoulli{Float64}(p=0.5), 'h', 't')
```

```
bent_coin1 = bend(fair_coin)
```




```
viz(randsample(bent_coin1, 100))
```

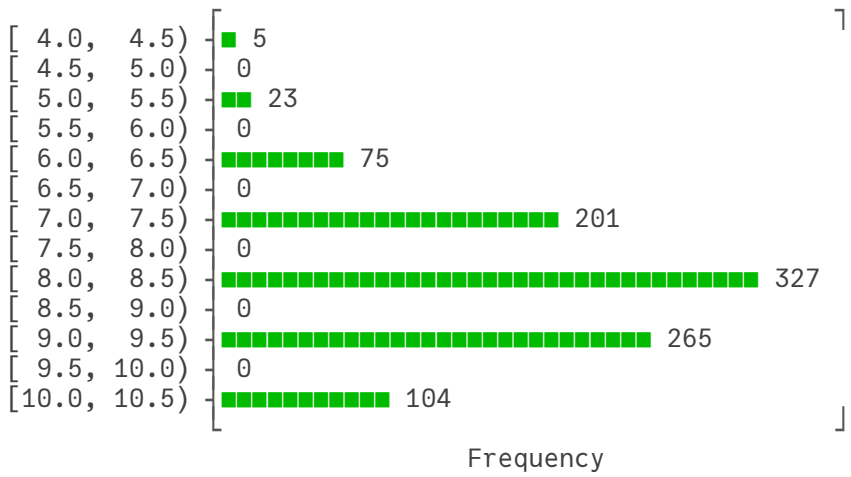
Here we visualize the number of heads we expect to see if we flip a weighted coin ($weight = 0.8$) 10 times. We'll repeat this experiment 1000 times and then visualize the results. Try varying the coin weight or the number of repetitions to see how the expected distribution changes.

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

```
begin
    make_coin_binary(weight, n) = n ~ Bernoulli(weight)
    coin(n) = make_coin_binary(0.8, n)
end
```

```
c = +ₚ(1@Distributions.Bernoulli{Float64}(p=0.8), 2@Distributions.Bernoulli{Float64}(p=0.8), 3@
```

```
c = pw(+, [coin(i) for i in 1:10]...)
```




```
viz(randsample(c, 1000))
```

Example: Causal Models in Medical Diagnosis

Generative knowledge is often causal knowledge that describes how events or states of the world are related to each other. As an example of how causal knowledge can be encoded in Omega, consider a simplified medical scenario:

```
false
```

```
let
    lung_cancer = @~ Bernoulli(0.01)
    cold = @~ Bernoulli(0.2)
    cough = cold .| lung_cancer
    randsample(cough)
end
```

This program models the diseases and symptoms of a patient in a doctor's office. It first specifies the base rates of two diseases the patient could have: lung cancer is rare while a cold is common, and there is an independent chance of having each disease. The program then specifies a process for generating a common symptom of these diseases – an effect with two possible causes: The patient coughs if they have a cold or lung cancer (or both). Here is a more complex version of this causal model:

```
-4572086551647114244@Distributions.Bernoulli{Float64}(p=0.1)
```

```
begin
    lung_cancer = @~ Bernoulli(0.01)
    TB = @~ Bernoulli(0.005)
    stomach_flu = @~ Bernoulli(0.1)
    cold = @~ Bernoulli(0.2)
    other = @~ Bernoulli(0.1)
end
```

```
cough = 
|ₚ(&ₚ(17284675711245273 67@Distributions.Bernoulli{Float64}(p=0.2), 583969151587176 7914@Dist
```

```
cough = pw(|,
    (cold .& (@~ Bernoulli())),
    (lung_cancer .& (@~ Bernoulli(0.3))),
    (TB .& (@~ Bernoulli(0.7))),
    (other .& (@~ Bernoulli(0.1)))
)
```

```
fever = 
|ₚ(&ₚ(17284675711245273 67@Distributions.Bernoulli{Float64}(p=0.2), -201491752392780388 2@Dis
```

```
fever = pw(|,
    (cold .& (@~ Bernoulli(0.3))),
    (stomach_flu .& (@~ Bernoulli())),
    (TB .& (@~ Bernoulli(0.1))),
    (other .& (@~ Bernoulli(0.01)))
)
```

```
chest_pain = 
|ₚ(&ₚ(42156538042057932 19@Distributions.Bernoulli{Float64}(p=0.01), -708107991231102089 9@Di
```

```
chest_pain = pw(|,
    (lung_cancer .& (@~ Bernoulli())),
    (TB .& (@~ Bernoulli())),
    (other .& (@~ Bernoulli(0.01)))
)
```

```
shortness_of_breath = 
|ₚ(&ₚ(42156538042057932 19@Distributions.Bernoulli{Float64}(p=0.01), -621613726682906887 8@Di
```

```
shortness_of_breath = pw(|,
    (lung_cancer .& (@~ Bernoulli())),
    (TB .& (@~ Bernoulli(0.2))),
    (other .& (@~ Bernoulli(0.01)))
)
```

```
symptoms = #11 (generic function with 1 method)
```

```
symptoms = @joint cough fever chest_pain shortness_of_breath
```

```
(cough = false, fever = false, chest_pain = false, shortness_of_breath = false)
```

```
randsample(symptoms)
```

Now there are four possible diseases and four symptoms. Each disease causes a different pattern of symptoms. The causal relations are now probabilistic: Only some patients with a cold have a cough (50%), or a fever (30%). There is also a catch-all disease category "other", which has a low probability of causing any symptom. Noisy logical functions—functions built from and ($\&_p$), or ($|_p$), and distributions —provide a simple but expressive way to describe probabilistic causal dependencies

between Boolean (true-false valued) variables. When you run the above code, the program generates a list of symptoms for a hypothetical patient. Most likely all the symptoms will be false, as (thankfully) each of these diseases is rare. Experiment with running the program multiple times. Now try modifying the function for one of the diseases, setting it to be true, to simulate only patients known to have that disease. For example, replace `lung_cancer = @~ Bernoulli(0.01)` with `lung_cancer = true`. Run the program several times to observe the characteristic patterns of symptoms for that disease.

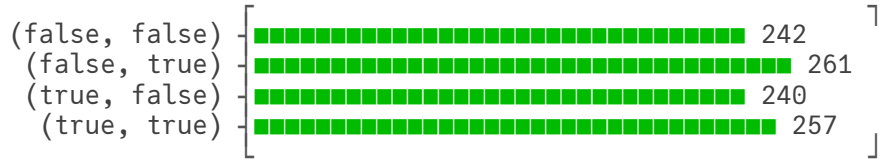
Prediction, Simulation, and Probabilities

Suppose that we flip two fair coins, and return the tuple of their values:

```
(false, false)
```

```
randsample(((@~ Bernoulli()), (@~ Bernoulli())))
```

How can we predict the return value of this program? For instance, how likely is it that we will see `(true, false)`? A probability is a number between 0 and 1 that expresses the answer to such a question: it is a degree of belief that we will see a given outcome, such as `(true, false)`. The probability of an event A (such as the above program returning `(true, false)`) is usually written as $P(A)$. A probability distribution is the probability of each possible outcome of an event. For instance, we can examine the probability distribution on values that can be returned by the above program by sampling many times and examining the histogram of return values:




```
begin
    random_pair = ((@~ Bernoulli()), (@~ Bernoulli()))
    viz(string.(randsample(random_pair, 1000)))
end
```

We see by examining this histogram that `(true, false)` comes out about 25% of the time. We may define the probability of a return value to be the fraction of times (in the long run) that this value is returned from evaluating the program – then the probability of `(true, false)` from the above program is 0.25.

The rules of probability

We can derive marginal distributions with the “rules of probability”. This is intractable for complex processes, but can help us build intuition for how distributions work.

Product Rule

In the above example, we take three steps to compute the output value: we create the first Bernoulli random variable, then the second, then we make a tuple of random variables and sample from it. To make this more clear let us re-write the program as:

```
(A = true, B = true)
```

```
let
    A = @~ Bernoulli()
    B = @~ Bernoulli()
    C = @joint A B
    randsample(C)
end
```

We can directly observe (as we did above) that the probability of true for A is 0.5, and the probability of false from B is 0.5. Can we use these two probabilities to arrive at the probability of 0.25 for the overall outcome `C = (true, false)`? Yes, using the product rule of probabilities: The probability of two random choices is the product of their individual probabilities. The probability of several random choices together is often called the joint probability and written as $P(A, B)$. Since the first and second random choices must each have their specified values in order to get `(true, false)` in the example, the joint probability is their product: 0.25.

We must be careful when applying this rule, since the probability of a choice can depend on the probabilities of previous choices. For instance, we can visualize the the exact probability of `(true, false)` resulting from this program by defining a new random variable as follows -

```
A = -1387394605752523029@Distributions.Bernoulli{Float64}(p=0.5)
```

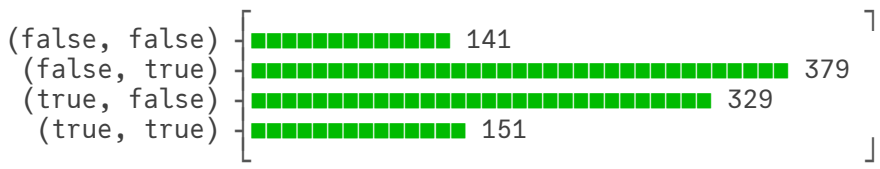
```
A = @~ Bernoulli()
```

```
B = 3474197976217959179@#1
```

```
B = @~ Bernoulli(ifelse.(A, 0.3, 0.7))
```

```
rs =
 [(false, false), (true, false), (true, true), (true, false), (false, false), (false, true), (t
```

```
rs = randsample((A, B), 1000)
```




```
viz(string.(rs))
```

In general, the joint probability of two random choices A and B made sequentially, in that order, can be written as $P(A, B) = P(A)P(B|A)$. This is read as the product of the probability of A and the probability of "B given A", or "B conditioned on A". That is, the probability of making choice B given that choice A has been made in a certain way. Only when the second choice does not depend on (or "look at") the first choice does this expression reduce to a simple product of the probabilities of each

choice individually: $P(A, B) = P(A)P(B)$. What is the relation between $P(A, B)$ and $P(B, A)$, the joint probability of the same choices written in the opposite order? The only logically consistent definitions of probability require that these two probabilities be equal, so $P(A)P(B|A) = P(B)P(A|B)$. This is the basis of Bayes' theorem, which we will encounter later.

Sum Rule

Now let's consider an example where we can't determine from the overall return value the sequence of random choices that were made:

```
s = |ₚ(3028430546181589271@Distributions.Bernoulli{Float64}(p=0.5), 3414675042352062654@Distrib
```

```
1 s = (@~ Bernoulli()) .| (@~ Bernoulli())
```

```
true
```

```
1 randsample(s)
```

We can sample from this program and determine that the probability of returning true is about 0.75. We cannot simply use the product rule to determine this probability because we don't know the sequence of random choices that led to this return value. However we can notice that the program will return true if the two-component choices are `(true, true)`, or `(true, false)`, or `(false, true)`. To combine these possibilities we use another rule for probabilities: If there are two alternative sequences of choices that lead to the same return value, the probability of this return value is the sum of the probabilities of the sequences. We can write this using probability notation as: $P(A) = \sum P(A, B)$ over all B, where we view A as the final value and B as a random choice on the way to that value. Using the product rule we can determine that the probability in the example above is 0.25 for each sequence that leads to return value true, then, by the sum rule, the probability of true is $0.25 + 0.25 + 0.25 = 0.75$. Using the sum rule to compute the probability of a final value is called is sometimes called marginalization, because the final distribution is the marginal distribution on final values. From the point of view of sampling processes marginalization is simply ignoring (or not looking at) intermediate random values that are created on the way to a final return value. From the point of view of directly computing probabilities, marginalization is summing over all the possible "histories" that could lead to a return value. Putting the product and sum rules together, the marginal probability of return values from a program that we have explored above is the sum over sampling histories of the product over choice probabilities—a computation that can quickly grow unmanageable, but can be approximated.

Stochastic recursion

Recursive functions are a powerful way to structure computation in deterministic systems. In Omega it is possible to have a stochastic recursion that randomly decides whether to stop. For example, the geometric distribution is a probability distribution over the non-negative integers. We imagine flipping a (weighted) coin, returning $N - 1$ if the first true is on the Nth flip (that is, we return the number of times we get false before our first true):

```
geometric (generic function with 2 methods)
```

```
1 geometric(p, ω, n = 0) = (n ~ Bernoulli(p))(ω) ? 0 : 1 + geometric(p, ω, n + 1)
```

1

```
randsample(ω -> geometric(0.6, ω))
```

There is no upper bound on how long the computation can go on, although the probability of reaching some number declines quickly as we go. Indeed, stochastic recursions must be constructed to halt eventually (with probability 1).

Persistent Randomness

In Omega, random variables are pure: reapplication to the same context (or ω) produces the same result.

```
ω = LazyΩ{@NamedTuple{rng::Random._GLOBAL_RNG}, Dict{Any, Any}}
    (rng = Random._GLOBAL_RNG(),)Dict{Any, Any}()
```

```
ω = defω()
```

tags:

data:

```
f = 1@Distributions.Bernoulli{Float64}(p=0.5)
```

```
f = 1 ~ Bernoulli()
```

```
g = 1@Distributions.Bernoulli{Float64}(p=0.5)
```

```
g = 1 ~ Bernoulli()
```

true

```
f(ω) == g(ω) # Always returns true
```

Independent random variables of a random class can be created in Omega by changing the `id` as follows -

false

```
let
    iid_1 = 1 ~ Bernoulli()
    iid_2 = 2 ~ Bernoulli()
    ω = defω()
    iid_1(ω) == iid_2(ω) # Does not always return true
end
```

Sometimes we require the results of the stochastic process to be random but persistent, for example: Eye colour of a person. We can represent the notion that eye color is random, but each person has a fixed eye colour as follows:

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

```
function eye_colour(n, ω)
    d = (n ~ DiscreteUniform(1, 3))(ω)
    if d == 1
        return :blue
    elseif d == 2
        return :green
    else
        return :brown
    end
end
```

```
[:blue, :green, :blue]
```

```
randsample(ω -> [eye_colour(:bob, ω), eye_colour(:alice, ω), eye_colour(:bob, ω)])
```

Bob's eye colour is consistent every time we call the above `randsample`.

This type of modeling is called random world style (McAllester et al., 2008). Note that we don't have to specify ahead of time the people whose eye color we will ask about: the distribution on eye colors is implicitly defined over the infinite set of possible people, but only constructed "lazily" when needed.

As another example, here we define a function `flip_a_lot` that maps from an integer (or any other value) to a coin flip. We could use it to implicitly represent the nth flip of a particular coin, without having to actually flip the coin n times.

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

```
flip_a_lot(n, ω) = (n ~ Bernoulli())(ω)
```

```
[[true, true, true, false], [true, true, true, false]]
```

```
let
    randsample(ω -> [
        [flip_a_lot(1, ω), flip_a_lot(12, ω), flip_a_lot(47, ω), flip_a_lot(1548, ω)],
        [flip_a_lot(1, ω), flip_a_lot(12, ω), flip_a_lot(47, ω), flip_a_lot(1548, ω)]
    ])
end
```

There are a countably infinite number of such flips, each independent of all the others. The outcome of each, once determined, will always have the same value.