clojure.spec

- **Speaker: Rich Hickey**
- **Event: [LispNYC meeting]**
- **Date: November 2016**
- **Video: https://vimeo.com/195711510**

[Time 0:00:15]

Hi! Thanks for having me. Thanks for inviting me.

As Pierre said, I did talk about Clojure in 2007, so 9 years ago I came to LispNYC, which was tiny, and gave the first talk I ever gave to anybody, anywhere, about Clojure. And that was a lot of fun.

So today I am here to talk about Clojure spec, and I just wanted to try to get a measure of who came today. How many people saw my talk at Clojure NYC? Only a couple. So hopefully it is not too redundant for you.

How many people know Clojure? How many people would call themselves Clojure programmers? How many people do not know Clojure at all? Oh, not too many. OK. So I will go relatively quickly through the slides for "I don't know Clojure yet." Can everybody hear me OK?

[About half a minute of audio adjustments.]

[Time 0:01:37]

```
slide title: Overview

+ What is spec?

+ Why does it exist?

+ Key ideas

+ Other Features

+ Leveraging spec
```

So I am here to talk about Clojure spec. And it is an introduction, so I am just going to talk about some aspects of it, but not really take you through, "here is how you do this, and here is how you do that." So I am going to talk about what it is, and why it is, what the key ideas are inside it. And I would like to focus on that mostly. A little bit about some other features, and then a little bit more about how to take advantage of it if you want to try to use it.

[Time 0:02:08]

```
slide title: What is spec?

+ a _library_ for Clojure

+ to define and leverage predicative specs

+ integration: macro-expansion and documentation

+ small tools
```

So the first thing to know about Clojure spec, and it is an important characteristic of Lisps in general, is that

you usually extend them by making libraries, and Clojure spec is just a library. We are adding a few things into Clojure to allow it to integrate with spec, but we have not changed the language in order to make spec work.

And it is a library that allows you to define, and then take advantage of in many different ways, predicative specs.

And integration points are relatively small. So we have built a little bit of code inside Clojure itself to allow macro expansion to talk to specs. And we have built a little bit more support so that the documentation string system will look to specs and enhance the doc strings with specs.

But overall, spec is a set of small tools. It is not like a giant monolithic thing.

[Time 0:03:14]

```
slide title: Why spec?

+ Robust programs

+ Communication

  + to each other

  + over wires

+ Better error reporting

+ Better testing

+ Flexible programs
```

So why do we want spec?

Fundamentally we want spec so we can write more robust programs. And Clojure has always been about writing robust programs. Its emphasis on immutability and functional programming is about making programs that do what you expect them to do.

But it ends up that there are challenges to doing that in any dynamic language that has dynamic typing, in that it is on you as a developer to communicate about how things work. Because there is not a type system, there is not a sort of standard for: here is how we are going to communicate about what the arguments are, what the expectations are of functions, and whatnot.

So it is about talking to each other: here is how this works. But it is also about talking between programs, between parts of programs, so that they can both produce data that is conformant to the expectations of the other end, and validate that what they have been handed complies with expectations.

Some of the things that it is responding to are, the community wanted to say, "Oh, we want to get better errors from macros." And I will talk about that in a second. It ends up that that problem is not a different problem than specifying how data structures should be assembled. If you have error reporting and validation for data structures, you have error reporting and validation for macros.

As well as enabling better testing. And doing all of this while retaining flexibility in programming. I think many times people presume that the reason why programmers prefer dynamically typed languages is because they are lazy, and they do not feel like writing all of the types out. And I am sure that happens in some languages, but I think the real reason why you would stay in a dynamic language is because you want flexibility.

As you build larger systems, you realize that so much happens at run time, and so many things happen over wires. And there is sort of presumption in many languages that the type system is going to solve every problem. It is just not practical. There is the famous adage about every large C program has a little crappy Lisp implementation inside it. That is not my adage. That is an old one. But it does, and I used to write crappy Lisps inside C++ programs.

And after you have done that for a while, you realize: I am doing this because I do not have sufficient flexibility. My systems are large. I need them to be malleable. I cannot afford to change the whole world whenever some small thing changes.

So when you get to the dynamic edges of those programs, you end up doing this stuff. In Clojure, we do it this way all of the time. But there are still edges to our programs, and if you do something in the language that is just about the language, you run up against the wire, and then it stops helping you.

[Time 0:06:21]

```
slide title: Why Clojure?

+ Data orientation

+ Simplicity

          [Clojure logo]

What matters is not just what a programming
language makes possible, but what it makes _practical_
and _idiomatic_.
```

So this sort of connects us, and I will go through this pretty quickly, because it seems like everybody has at least touched Clojure, and half of you would call yourselves Clojure programmers, but why do we use Clojure? There are a bunch of choices. There are a lot of dynamic languages.

I think there are two fundamental reasons. And this is really abstracting out a lot. One is Clojure has data orientation. A lot of people would not necessarily choose Clojure for this, but they get inside Clojure and they are like, "Whoa! I mean, everybody is doing everything with maps. Like everywhere, all of the time. What is going on?" And it is initially a hurdle, and then it becomes confusing. Eventually it becomes enabling, and finally it is like, "Of course this is the way we should be doing this."

The other is simplicity, and I think that is a covering thing for all kinds of stuff, including immutability and functional programming. Functional programming is about simplicity. It is about saying: the inside of this function is not connected to anything else but itself. You look at it, and it stands alone. It is not entangled. So functional programming is inherently simple, and immutable data is inherently simple.

But what matters is not just what is possible, but what is idiomatic and practical, and Clojure tries to make these two things that way.

[Time 0:07:41]

```
slide title: Data Processing

+ Most programs acquire, transform, store, search,
  manage, transmit data

+ Data is raw, immutable information

+ Many langs turn into something much more
```

```
elaborate - with types, 'methods' etc

+ esp. OO conflates process constructs and
  information constructs

+ misfits for wires
```

I think again, in programming, we get sort of high-falutin' about our programming languages and programming itself. It is all about us. It is really not. We used to call programming "data processing", and then we were too cool to do that any more, but this is what we do. How many people do data processing, do any of these things I just listed. Yeah, I mean that is what we do. We could call it whatever we want: I am in software development, I am an engineer. But almost every program is going to take some data in, transform it in some way, stick it somewhere, try to find it again later, manage its life cycle, and send some subset of it to somebody else at some point.

And I think being honest about this essential characteristic of programming is something that Clojure does. And when you talk about data then, you really want to say, "What is that? Is it a class? Is data classes?" And I would say, "Definitely not." If you go and look at a form that somebody fills in at a medical center, or something like that, there are no classes, there are no Java-isms. There are no language-isms at all. If we impose them, we are superimposing them.

But there are sort of fundamentals to data. There is the idea that things are sequential, that one thing came after another thing. And then there is the idea that things are associative: this person's address is that. The mapping from this thing to that thing. And actually, there are not a lot of others. There are sets, which have logical characteristics. And a lot of the rest of data structures is about performance characteristics, but not about sort of the essentials of it. So information has a certain set of primitive properties, and Clojure tries to tap into these, and supplies data structures for those.

So by not conflating the processing we are trying to do with the information, we end up with very simple informational classes. If you don't do that, I think you end up with a big misfit when you reach the wire. You have things like ORMs and all kinds of mismatches. You can replace the O and the R whenever you want, the M is always there. And the reason why languages have Ms is because they just do not have a way to talk about data that is direct and simple and unadorned. So every time they hit a wire, there is going to be a loss of precision compared to the language view of the thing, and every time something comes over the wire they are going to have to superimpose something.

[Time 0:10:24]

```
slide title: Data

+ Clojure embraces data

+ Data literals

+ Majority of functions take/return data

+ _Information_ is represented in Clojure systems as
  _plain data_

+ transparent language-independent serialization
  over wires - edn, Transit
```

So Clojure embraces data. Everybody knows about the data literals. How many people are also Common Lisp programmers, or Scheme programmers? How many people are Common Lisp or Scheme programmers who are

horrified by Clojure's "things that aren't in parentheses"? No longer. Still one! No longer. Early on, there were many more hands.

So there are data literals, and that is the thing that a Lisp or Scheme programmer would be like, "Whoa! What did you do there?" I like Common Lisp a lot, and I do not believe in "I want to just see this in a different prettier way." Those data structures are as first class to Clojure as lists are to Lisp. And that was important to me. And also, the idea of conj working on maps, that is a fundamental Lisp idea, cons. It is just not only about cons cells. It is a bigger idea that had a certain implementation in Common Lisp, but I think was a bigger idea that they had. So Clojure values that stuff.

But by having this set of data literals, and this very small set of data structures, we can have this giant library of functions that manipulate these few data structures. And that is important. And that means that in practice what happens is, at least for the informational part of your program – so programs are not just information. They do all of the stuff I said before, but a lot of times there are parts of your program that are like little machines, or like conveyor belts. You do other stuff with that, but in general when you have information represented in your Clojure program, you use data.

And a big advantage in Clojure of having data is that we have this language-independent representation. In edn, it just goes over wires. The reader is a built in thing. We now have Transit also as a binary high performance thing that reaches the browser really well, both of which take all the data structures of Clojure and move them around without you having to think about it.

And it is straightforward, because unlike Java serialization there is no extra language gook there. It is just the pure informational part. For instance, there are not object trees and reference cycles and things like that.

[Time 0:12:42]

```
slide title: Data for ...

+ Code

+ Syntactic extension

+ Communication

+ Configuration

+ HTML and markup

+ Schema, type annotations, DSLs, etc
```

So we end up using data for everything. We represent our programs as data structures. We use data for syntactic extension. The macro system of Clojure are functions that take code as a data structure, and return a different data structure for evaluation. That is just a data to data transformation, which is why error reporting for macros turns into this problem, because that is all that macros do.

We obviously use data for communication over wires. But we use it everywhere. If you want a config file in Clojure, do you use XML? No. You use edn. A lot of libraries support representing markup languages and HTML as Clojure data.

And so on, and so forth. Schemas, transactions, query languages, type annotations, DSLs. We do this everywhere.

[Time 0:13:36]

```
slide title: The Clojure Premise
```

```
+ We can build a substantial portion of our programs
  using just data structures and pure functions
  thereof

+ These programs will be substantially smaller,
  simpler and more robust than OO program that do
  the same jobs

+ We should make programming with data idiomatic

+ spec leverages and extends Clojure's embrace of
  data
```

So this is what Clojure is about. It is an important preamble to understanding spec. Because we use data everywhere, anything you do that can enhance working with data enhances all of this stuff. And that is something to keep in mind, because I am just going to show you a tiny little slice of spec, but everywhere you see it working, you can imagine it working in these contexts.

[Time 0:14:06]

[switches back to previous slide "Data for . . . "]

Over a wire. After you get something back from a wire. When you want to create a config file. When you need to parse a DSL. When you want to have something that writes code in your DSL. It does all of that.

[Time 0:14:20]

[switches back to slide "The Clojure Premise"]

Blah, blah, blah.

So the last point is the only thing I want to say out loud. Spec leverages the fact that Clojure is data oriented, and it extends that power. And I think also, in a sense – maybe we will revisit this at the end of the talk – I think spec validates this approach to programming, because Spec is a small library that has an amazing amount of utility, because of the uniform and universal use of data throughout Clojure, and Clojure programming. This is Clojure winning. And this is the value proposition of Clojure exposed, that you can have a thing like Spec and have it work in so many places.

[Time 0:15:07]

```
slide title: Ideas

+ Predicative <-> Generative

+ Structural <-> Functional

+ Validation <-> Destructuring


             (not today)

+ Openness <-> Change

+ Choice <-> Paths
```

So there are a bunch of ideas behind Spec that I want to talk about, and I will not be able to talk about all of them. Some of them have duals. This is sort of fake, this yin yang thing.

The predicative generative one isn't, because there is a certain sense in which Spec is bidirectional. So Spec is a predicative specification system. It means – and we will talk about this more in a second – but it means you say, "This thing succeeds, or is valid, because this predicate – just like any predicate you can think of, like map? or int? or odd? or whatever – says true of it."

So in one direction, specification is about validation. It is about saying, "I have some stuff. Does it match this specification? Are these predicates true?"

But the cool thing about Spec is that in all cases, it requires that that go the other way. If I have this predicate, make me stuff. Go make me stuff. And that bidirectionality is really sort of the source of all of the power in Spec. Otherwise it would just be a validation library. So we want to talk about that.

It is both a structural validator, and a functional validator. So when we talk about functions, on one level we can say, "It takes a this, and it returns a that." That is useful, but it is not actually terribly interesting. More interesting is saying what it does. So how do you talk about what a function does? So we will dig into that.

The other thing that the Spec design sort of noticed is that when you validate something, you are only half done. You are like, "Is this big hairy data structure you sent me valid? Yes!" I still have a problem now, which is what? Now I have to write a bunch of code that co-aligns in a significant degree with that validator in order to parse it, or walk through it. And I have conditionals in my code that branch in all of the places where I branch in my specification. I wish I did not have to do that. And it ends up that specifications should yield destructuring, in my opinion. And Spec does.

[Time 0:17:28]

The other two big things that motivate Spec, and sort of underlying the design, is this idea of openness and change. Spec sort of requires specifications to be open. There are very few places in Spec, or maybe none, where you can say, "You cannot do something."

And this is because – there is one way to look at Spec, which is that it is kind of a logic library. It is kind of a very simple logic library. And in logics, we generally do not say, "This cannot be true of something." Because we do not know everything. So you rarely see that operation in logic: this is the only thing that can ever be known about this. You do not do that in logic systems.

And the reasons why do not, is because you cannot change anything that was ever written that way. It is concrete. If you ever said something like that, you would be pouring concrete, and you could never make an extension to that system. Because any extension would invalidate the first thing you said. You said this is the only thing that could ever be possibly said, and now I want to say one more thing. They both cannot ever be true.

So openness is the path to change. There are many other things about Spec that are oriented towards change. For instance, when we want to be able to change things, we would like to understand the compatibility relationships between what we had, and what we are changing it to. There are some changes that are compatible. There are other changes that are not compatible.

But how do you know that? Right now what do we have? We have Semantic Versioning. How many people believe in Semantic Versioning? Yeah. I mean, it is a belief system. It is not science. Let us just say that. It is not mathematics or anything. Because really there are no raw materials for assessing that. "I changed it to 2.0" "Well, what happened?" "Well, I don't know, but you know something dangerous did, and you better retest everything."

"Oh, I only changed it from .1 to .2." "Oh, I am fine!" How many people believe that?

[Audience laughter]

[Time 0:19:28]

No! No. So as we talk about it, we have chosen constructs in Spec for which there are mathematical tools that would allow us to say, "This set is either a subset, or not, of this other set." And that is a compatibility test. Or, "this regular expression accepts the things that this other one did." If it accepts more, it remains compatible. If it cannot accept everything the other one did, it is incompatible. That is a solved problem in mathematics.

The other thing that Spec does is it makes sure that when you make a choice, you get a realization of that in data. And again, I am not going to talk too much more about this, but it goes back to the point I made earlier about a validator that deals with "or", or optionality, is branching. And later when you want to manipulate that data, you want to know which branches it took. And unless there is a reification of that, you are not going to have that power.

[Time 0:20:32]

```
slide title: Predicative

+ specify data and functions by stating what must be
  true of them

+ arbitrary, open set of predicates

  + using code (pos-int?, map? etc)

  + and composites thereof

+ logical combinators (and, or et al)

  + that remember their constituents
```

So we will dig into some of these. So what does it mean for Spec to be predicative? It means we are going to specify what data and functions mean by stating what must be true of them. And we are going to do that with just regular predicates. Ordinary predicates and code, like map?, or pos-int? is now in Clojure. Things like that, or composites of those things. So it is a positive int, and it is greater than this number, or something like that.

There are, built into Spec, some combinators like "and" and "or". Now this is something we already have in Clojure. We have "and" and "or". And this goes back to that choice and branching thing that I just talked about, which is: if there is an "or", it could be this or that, and they are both Specs. So either this Spec is true, or that spec is true. And you go down and the first Spec is not true, and the second Spec fails for another reason. Or you say "and", and the first one is true, and the second one is true, and the third one is not, coming back and just saying, "It did not work." How many people like it when your users say, "It did not work."

Right. You wanted some details. And of course if we used the "and" and "or" of Clojure directly, we would only have that sort of thumbs up kind of characteristic.

[Audience member: some question]

No negation has got that closedness characteristic, so no. You can obviously make a predicate that is not. Evens are not odd, in a sense.

So the ones built into Spec are there so you can get the branches remembered, and you will get detailed errors that are inside the branch. So this "and", it is the third one it did not satisfy. And so on and so forth. It will navigate down into Specs to be able to do that.

[Time 0:22:31]

```
slide:

(require '[clojure.spec :as s])

(s/def ::big-even (s/and int? even? #(> % 1000)))

(s/valid? ::big-even 42)
false

(s/valid? ::big-even 4200)
true
```

So we use Spec, we say require clojure.spec as "s". And that will be used throughout this. There were not too many people who did not know Clojure at all, so I hope everybody else just sort of hangs on. Clojure code is written in parens, and the first thing is the verb. That is sort of the main way to look at it. Things that begin with a colon, or two colons, are just symbolic identifiers that signify themselves. And everything is variadic, pretty much everywhere you look.

So we get Spec, we say define big-even to be Spec "and", so the and-ing of int?, even?, and it is greater than 1000. So we take three predicates and we "and" them together. So we say all of these things must be true. If they are true, it satisfies the specification big-even.

And then Spec has a function called valid? that you can use to test something. You say "valid?", then you say the spec, and then the stuff. And it says no, or it says yes.

And as I said before, that is cool, but as we will see when we get into conform, that is not enough. Especially when you start saying "no", you would really like to understand more about it. Or if there was branching in the validation, you might want to recover that. So there is another primitive called "conform" that does that. It also does the validity test.

[Time 0:24:07]

```
slide title: Generative

+ a spec is made out of predicates

  + but is not (just) a predicate

+ should be able to generate data (and functions!)
  that satisfy spec

  + many uses: testing, sample data

+ mostly for free, sometimes help required
```

So we make specs out of predicates. We can compose them together. We can "and" and "or" them. They can be arbitrarily deep, and I will talk about some of the data structure ones in a second that allow you to make tree-like things.

But then given a Spec, which is the predicate, all Specs go the other way. So all of the baseline predicates built into Clojure now have corresponding generators for them. And then all of the Spec constructs like spec/and and spec/or can compose those generators to make generators of the composed thing. And this is what allows Spec to sort of "go both ways", to both validate stuff against a Spec, or to take a Spec and make stuff.

So this is one of the fundamental principles of Spec: you should be able to generate data. It is from this that we get the testing leverage that Spec provides. So Spec should be able to generate data and functions that satisfy Specs, and Spec can do that.

And I mentioned that testing is a primary use for this, and we will look at that later. The other thing is just sample data generation. I make a Spec. This is going to be my wire protocol. It is really nice to be able to push a button and get an instance of data that corresponds to your Spec, and look at it and say, "Oh, no! I left out whatever." As opposed to writing it by hand and then just calling validate? and having it saying no, no, no, until you get it right. You just say no, make me a right thing and I will look at it.

In general, this is for free. So you will always get a generator for free. Sometimes, though, the generator has to work too hard. Like it can generate it, but it might take your entire lifetime to do so, because the space of possibilities is so large. And basically it is doing random generation until it generates something that satisfies a predicate. So sometimes you have to help it out and make a more specific thing, depending on the density of the space.

So it is pretty straightforward to say, "generate all of the numbers and throw away the ones that are not even." That is relatively efficient. It is much more difficult to generate very specific numbers like astronomical constants. A Spec for that would try hard for a long time.

[Time 0:26:25]

```
slide:

(pprint (s/exercise ::big-even))

([69396 69396]
 [425440 425440]
 [515646 515646]
 [2016 2016]
 [11286 11286]
 [31766 31766]
 [1640226 1640226]
 [59598 59598]
 [29654 29654]
 [18370 18370])
```

So what does generation look like? So we have big-even from before. "exercise" is the simplest generator access that we have. And it takes a Spec and it makes instances of the Spec. In this case, it is exercising a piece of data Spec so there is nothing interesting, no interesting difference between these tuples of things it gives back. But when we have function Specs, we will see this returns a pair of what I started with and what the conformed value is.

So this is a pair of values. It is the value it generated, and the conformed value when it is passed through the Spec. For a primitive atomic Spec like this, those two things are not different.

[Audience member: some question]

That is built into "int?" here.

[Switches back to slide starting "(require '[clojure.spec :as s])"]

[Audience member: So you are generating all sorts of data types?]

No, "int?" has got a generator for it. So that is what I was saying. A bunch of the primitive bottom atomic predicates have generators built into them already. [tbd] the domains. That is correct. That is right, exactly.

So this is generating two things, a generated value and the conformed version. We have not really talked about conforming. When we are talking about something like big-even, they are not different. So you can get a bunch of big-even numbers just by asking for it.

I guess this is demonstrating another important part of Spec, which is: when would you do this? What is this for, "exercise"? It is for while you are programming. While you are writing your program. This is helping you. And that is something you really want to keep track of with Spec.

And I think with Lisps in general, I think people, when they come to Lisp, they do not appreciate being able to use Lisp while you are programming, just to make things more straightforward. To generate data, or to look at something, or to do whatever. It is not like you go into the REPL and you call the thing that you just wrote. You have all of the power of the language available to you, to help you make decisions, assess correctness, and things like that, in a very interactive way.

[Time 0:28:46]

Oh, I forgot to ask. How many people have tried to use Spec? That number is still low. OK.

[Audience member: some comment]

No, it is alpha, so I only want people to use it who are aware of what that implies. But once you start using Spec, "exercise" is like your favorite thing in the whole world. When you write a Spec, this is the first thing you do. Make it make some stuff. And you will immediately find bugs in your logic, right away, from this. We are not even running Spec's testing capability. So you can generate data.

[Audience member: Can you define generators for your own data types?]

Yeah, yeah, yeah. It is completely extensible. You can define generators for your own predicates. And you can also define generators for composite specifications. And you can do this a la carte. Like you can make one that is connected to your Spec forever and ever, but also in a particular context you can override the generator. And say, right now, I do not want to make crazy looking data. I want to make data that looks nice for a screen or a presentation that I want to do, and narrow the domains.

[Audience member: some question]

That is right. It is aligned with the predicate or the Spec. So you can connect it to either of those things. That is right.

[Time 0:30:10]

```
slide title: Structural

+ Homogeneous container-like collections (every,
  coll-of, map-of)

+ Heterogeneous maps (keys)

  + and their flattened kwargs analogues (keys*)

+ Syntactic sequences, where order determines
  meaning (regex ops)

+ Clojure code is structural!
```

So we just saw something atomic. How do we talk about data structures? Because atomic data, it is at the bottom, but it is not that interesting. And it does not feel like information yet. 42. I am glad I know that. You are not there yet.

So we have a bunch of things that will specify structural predicates. And these predicates are built into Spec. "every", "coll-of", which is collection of something, or "map-of" something. And the target here: everything is this, or everything is a collection of those, or map of whatever, that thing is either a predicate or a Spec itself.

So you are not saying some built-in type. You are saying a predicate. So you could have a collection of big-even's, which was not a type. It was just some predicate that you satisfy. But you can say, "Everything in this collection is a big-even".

So the other thing we do in Clojure, because we love our collections, and we have this big library, is we use collections for two very distinct things, especially maps.

We use maps as collections, container-like collections. I have a whole bunch of people's friend lists keyed by their email address. So it is a map of email address to friend list. And the whole thing is email address to friend list. That, to me, is a map, an associative data structure that is being used as a container. A homogeneous container.

[Time 0:31:49]

But we take that same data structure in Clojure, which you could put a million email addresses in, or whatever, we also use it in a really tiny way where we say, "somebody's name is this, and their email address is that, and their phone number is this, and whatever is that." And that seems kind of record-y or object-y. It is still a map. It is the same data structure, it is the same everything, and all of the functions work on either. But the pattern of use is completely different.

Here we have heterogeneous maps, where the map's keys could have different types, but they have keywords with different values, mapping to various values. So the name is this, and the age is a number, and the other thing is the other thing. It is not a homogeneous collection or container any more. So we use maps as data records.

And we use a different Spec primitive for that. It is called "keys". And "keys" says "these are the keys that are in this map," that you can find in this map. And there are various ways it can say that. We will look at that in a second.

The other sort of fundamental structural thing are sequences. So we already have "coll-of". With coll-of you can say more things about the coll. Like you can say "the kind of the collection is a vector." So again, if you are dealing with homogeneous data, you want to use coll-of.

But a major thing that we do, and I prefixed this sequence thing with "syntactic" is, the other thing that we do, especially in programs and DSLs and code, is we have places where order matters. What did I say about Clojure code? "The first thing is the verb." That is syntax. That is me describing syntax. The order matters.

And so how do you talk about that, because that is not a homogeneous anything, and order matters. And it ends up that regular expressions are the math of "order matters".

And another thing to remember from before is that Clojure code is structural. So again all of this stuff is going to apply to code.

[Time 0:33:55]

slide:

```
(s/def ::a int?)
(s/def ::b int?)
(s/def ::x string?)
(s/def ::y float?)
(s/def ::z int?)
(s/def ::a int?)
```

```
(s/def ::m (s/keys :req [::x ::y ::z] :opt [::a]))

(s/valid? ::m {::x 1 ::y 2})
false

[Note: The output below may differ from what is shown on the slide.
It was created about a year after the talk was given using the
officially released Clojure 1.9.0, and the versions of
clojure.spec.alpha that it was released with as dependencies.]

(s/explain ::m {::x 1 ::y 2})
In: [:user/x] val: 1 fails spec: :user/x at: [:user/x] predicate: string?
In: [:user/y] val: 2 fails spec: :user/y at: [:user/y] predicate: float?
val: #:user{:x 1, :y 2} fails spec: :user/m predicate: (contains? % :user/z)
```

So let us look at something more involved.

[Audience member: some question]

It is, except it also has recursion. So then it is context free at that point. Right. Regular expressions would support all regular grammars. And then the recursive aspect takes you to the [gestures thumb up in the air] next. You do not need to know that to use Spec, if you are not into that kind of stuff.

So here we are defining a bunch of Specs. So you would wonder, "Why would you say"a" is int, and "b" is int, and "x" is string and "y" is float and "z" is int?" Why wouldn't you just use int later? And we are going to see the first thing that seems really weird about Spec.

So what I am trying to do here is: I am trying to spec "m". That one [points to line beginning "(s/def ::m"].

And when I spec a map, I am going to use that "keys" primitive I talked about before. And what "keys" says is exactly that. The thing you can spec about heterogeneous maps is, "what keys can it have?" In fact you can say two things about it: what keys *must* it have, and also what keys *might* it have. This is *not* an exclusive list, again due to the openness principle I talked about before.

So if we work backwards from "m", we are going to say "m" is something that must have the keys "x", "y", and "z", and it can optionally have the key "a". This spec for "m"s, for this kind of map, does *not* talk about what "x", "y", "z", or "a" mean.

How many people have used schema, or something like it for Clojure?

[referring to the Clojure library Plumatic schema: https://github.com/plumatic/schema]

And I am not picking on schema, because I do not know of anything that works this way [i.e. like Spec], except RDF, quite interestingly.

[RDF is Resource Description Framework - https://www.w3.org/RDF]

The RDF people were really on to stuff. They just liked XML too much, but they really were on to stuff. One of the things they were on to was decoupling the specification of the meaning of an attribute from anything that was a composition of that attribute with other stuff.

[Time 0:36:17]

Yeah.

[Audience member: What is the rationale for . . .]

So the question is: Why do we have another database, and not using Vars as the database?

[Audience member: Yes.]

Because putting everything in one database becomes tiresome after a while. It is overloaded, there is too much stuff there.

The beautiful thing about fully namespaced names is: anybody can have a database anywhere that they want. And so Spec has a database where it wants, of its stuff.

[Audience member: some question]

So these could have been Vars. That is a kind of database. I think that is a database that is overloaded. It is also reified, and if I was going to do anything with the Var system I would reduce the amount of reification it had. In which case, I really do not want to put more stuff in there.

So those are two answers. I probably have ten more.

So everybody may not know: the double colon means these are fully namespace qualified. In this case it is saying: the full name of this is "the namespace that we are in", say user, slash "a". user/b, user/x.

As soon as you start using fully namespaced keywords in Clojure, or fully namespaced Var names, as long as you have not chosen a really common word for your namespace like "ring" – if you follow the Java rules, and either use "com.something-you-own", or a trademark that you own, or something like that. That is a universal name that is human readable. There is a tremendous amount of power in that.

Clojure supports namespace names. I think we are underutilizing them. Because one of the really cool things you can do with namespace names is you can have a database anywhere, about anything. It does not need to be: "Everything about this has been poured into this Var." Everything you could possibly say about this. The documentation string here is there. People say, "Oh, we should have examples on Vars." No. The Var has a fully qualified name. Make a web service, examples-of-whatever, take the Vars a parameter, and return whatever you want. This should be a path to many databases. Not: the Var system is the dumping ground for all knowledge.

[Time 0:38:54]

So back to keys.

Is that good? Is that enough? You need more?

[Audience member: . . .]

Maybe later. We squashed that.

There is something very important. I sort of skipped over it. So Spec has a database of keywords mapping to Specs. And it is built in, and the way you talk to it is spec/def, saying, "I am going to define the Spec for this fully qualified name." You cannot define the Spec for something that is not fully qualified.

So we are back down to map. We are saying this map has these keys. We are saying the keys "x", "y", and "z" are required, and the key "a" is optional. But we are not saying what is in them. And this ends up being super important in making systems that compose data.

And so therefore, where are the definitions for what "x" means, and what "y" means, and "z"? They are the things that happened first [referring to the s/def lines on the slide before the one for "m"]. They all have their own definitions.

But it means that if there is some other part of my system, I want to make a data structure that has user/y and user/z, and user/a is required, I do not make another thing, and I do not re-say what "x" and "y" and "a" mean again. Because that is a bug. That is just a bug waiting to happen. And redundant.

And it cannot be done dynamically. We will talk a little bit more about the dynamic case later. So basically, map specs in spec are sets of keys. They are multiple sets, because some are required and some are optional.

And it is not closed. That is what that means, that "keys" thing. It says "m" is a map, and it must have "x", "y", and "z". Their meanings are defined somewhere else. And it may have "a", and other stuff. And we will talk about why you would even have optional if it can also have other stuff it does not talk about, in a second.

[Audience member: question]

[Time 0:40:50]

Allow other keys, right, except there is no false optioning spec. There is no disallow other keys, and that would be the closed problem that I had before. You will think that you want that. There will be a time in your life where you will think you will want that. Then there will be a time, if you ever make that for yourself, which you can do, but I am not going to do for you, where you will regret doing that.

[Audience laughter]

[Audience member: comment]

Yeah. It is just going to happen. So I am not going to help you do that. And it has happened already, and we have had consulting clients where we have had, "Oh, we want to have this." And when you do it, and then this happens.

So what happens? What happens is: it is the concrete boots. You have an in-extensible system. And you *cannot* fix it, except by going back to the original spec and saying, "You know what? I should not disallow this." You really do want to make open systems. You really do want to make systems where, if it is not the keys you care about, OK! You just do not read them. Right? You don't care. Let it flow. If you let it flow, you can make a system where somebody can add something at the other end, and everybody in the middle does not care. That is super important.

[Audience member: something]

Yeah. Except it is still sort of contextual, and the beautiful thing about this is you can go back to the – I mean, it does not really define what the keys mean. These independent key definitions mean that you can compose arbitrary sets and unions and intersections of key sets, and you do not need to redundantly state what the keys mean.

Yes.

[Audience member: question]

How do you catch typos?

[Audience member: question]

So it depends on where you want to catch it. Like your spec is broken, or your use is broken?

[Audience member: question]

So if your use is broken, if you are calling validate and it is a required key, it will say you passed "zz" instead of "z". You did not pass "z".

[Time 0:43:08]

[Audience member: question]

You can do that. Yes, it does not help you with that. If somebody really needed it, then they would have to ask for it. It does not help you with that problem at all.

[Audience member: question]

That is the other side of an open system, because "a", "b", it could be useful to somebody else. This person specified this. Didn't do that.

If it was important for you to produce data that had “a”, and you wanted to validate that, you could take data that you intended to meet this specification, and specify it more stringently with your own spec where “a” was required, and catch that problem.

If it was part of your contract that every time I call this other person who has asked for an “m”, I want to supply an “a”, you could make “my-m” spec where “a” was required, and you could validate your data before you sent it. But otherwise it is an open system.

[Audience member: question]

Yes.

Yep!

[Audience laughs.]

Of course! And that is why it is there, for generators and for testing. So the generators will generate it, and tests will see this data. What you are saying is, the thing that interoperates with “m”s will use “a” if you supply it. So you want to supply it sometimes, so you test that part of the code.

OK, so let us start using “m”, and we will dig into some of these other details as we go.

So the first thing we can do is just call that valid? function from before. Is this a valid “m”? And we pass a map that has an “x” and a “y”, and it says no. And again, yes and no is not that great. So something went wrong.

[Time 0:45:08]

So we have another new operator we have not seen yet, which is called “explain”, which takes the same spec and a value, and it will tell you what about it does not satisfy the spec. It says, quite verbosely, and you can get a data version of this, which you can process programmatically and make it colored or print whatever strings you want. But it is telling us all that information, which is that this value you passed fails the “m” predicate, that it must contain user/y as a key. I mean, user/z as a key. It does not have that. And it does not have that in the first thing that you passed. Oh, it also says that the user/x value that you passed fails the fact that user/x is supposed to be a string. And the “y” that you passed, 2, was supposed to be a float.

So all these things are wrong. So explain is sort of nice. It finds all the problems. And it tells you about them. There is explain data, and other things like that. We are not going to dig into those.

[Time 0:46:06]

```
slide:

(s/valid? ::m {::x "str" ::y 2.0 ::z 11})
true

(s/valid? ::m {::x "str" ::y 2.0 ::z 11 ::b "42"})
false

[Note: The output below is difficult to read on the talk video.  It
was created with officially released Clojure 1.9.0, after the talk.]

(s/explain ::m {::x "str" ::y 2.0 ::z 11 ::b "42"})
In: [:user/b] val: "42" fails spec: :user/b at: [:user/b] predicate: int?
```

So we fix our problem. We try another thing. We make “x” a string, we make “y” a float and include it, and we include “z”, and now it is valid.

We can try another version of it where there is "x" and "y" and "z", and "b", and it says it is not valid. What is interesting about this? Who has a really good memory, from one slide ago?

[Audience member: something]

"b" was not listed! Wait, we did not talk about "b" in the spec for "m". What is happening?

What is happening is, when a map is checked against the spec, any spec, it is an open system. All of the predicates for the map spec itself are tested. Must have "x" and "y" and "z". It may have "a", but that is not something you actually validate.

But *all* keys in a map will be looked up in the spec database to see if there is a spec for them. And *any* value that is supplied under that key will be validated against that spec.

So here [previous slide] there is a spec for "b". We did define one. We did not use it here [in the definition of "m"], but there is a spec for user/b. It says user/b must be an int. And here we are passing user/b as a string. And when we ask for an explanation it says users/b is not an int.

Why could this possibly be valuable? Why would we do something like this?

This has to do with composability, and flow, and flexible systems. How many people have ever built a system where there is a front side of the system, and it is getting data off of the wire, and it has to do some initial processing, and it looks at some of the keys of the map that comes in. Then it hands it off to somebody else, and somebody else, and blah, blah, blah. And somebody way over here at the end of some processing chain looks for some other key, and tries to use those keys, and process that data.

Anybody ever do that? Yeah, you do that all the time. That is how you make a flexible system. Otherwise everything time you touch anything, you touch everything in between.

[Time 0:48:21]

So, if you do not have something like this, when are you going to find out that "b" was not an int? *All* the way at the end. It came in here, and went through all of these things, and then somebody tried to use "b" and it was like, "Whoa! That is not an int, that is a string. That is wrong."

So how do you know what went wrong? Where it was wrong? Oh, you spend a few hours looking at all of the code in between these two things, which you do not want to do. So the other beautiful thing about this system is that it allows these tests to run anywhere a map spec is run, which means this can be checked all the way up front. We will find out about this problem at the door. Somebody supplied a "b" for consumption way down in the chain, and it is not correct. And therefore it gets flagged. This is a big deal for composability.

[Audience member: question]

Requires a field?

[Audience member: question]

So what we found here was an instance where they may not need "b", but when they look at "b" they are finding the wrong stuff. So it is sort of two different problems. This says, " 'b' is not the right type." It does not satisfy its spec.

The thing you are talking about is, "I need 'b'." I need "b", you need to flow all the way up. You need to include that in your door, if you need "b". Well you will discover it whenever you say, "I need it." If you want to make sure you reject it at the door, you will need to include it in your at-the-door spec.

[Audience member: question]

[Time 0:50:37]

So I am going to summarize what you said, because I did not hear it.

[Audience laughs]

Which is, the nice way to do this is to build a system that allowed programmatic composition of dependent specs, and not have the master knower of all things, the big bad spec.

So this is one of the big benefits of, that the specs for keys are independent of the specs for the composite. In addition to arbitrary composition, intersection, unions, and things like that.

[Time 0:51:11]

```
slide:

[Note: The output below is difficult to read on the talk video.  It
was created with officially released Clojure 1.9.0, after the talk.]

(map first (s/exercise ::m))

(#:user{:x "", :y ##NaN, :z -1, :a -1}
 #:user{:x "4", :y 0.5, :z -1, :a -1}
 #:user{:x "Q5", :y -0.5, :z -1, :a -2}
 #:user{:x "", :y 0.75, :z 1, :a 1}
 #:user{:x "H", :y 1.0, :z 0}
 #:user{:x "LMxr4", :y -1.0, :z -2}
 #:user{:x "J", :y -1.5, :z -2, :a 9}
 #:user{:x "9khfA", :y 0.8671875, :z -4}
 #:user{:x "D0", :y -6.5, :z -13, :a 0}
 #:user{:x "DC", :y 1.5, :z 1, :a -2})
```

So let us try that exercise function again. We are going to use "m" now, and we get a bunch of maps that correspond to the spec. And somebody was asking about "a", and you see sometimes we have "a", and sometimes we do not, because "a" was optional. And this is just the beginning of exercise, but if you run the generator, you will get big hairy combinations of everything, with big and rich data.

So we can exercise the map spec. If you cannot read this, this is just lifting "user" up off the nested keys because they have a shared namespace prefix. So that is just new syntactic support for maps.

[Audience member: question]

Yeah, you could. You could, absolutely, have a predicate. So you would start with something like the structural predicate, it is a collection of whatever, and then "and" that with some assessment of the distribution. With "and".

That is the point of composition, because otherwise all of these specs would have so many knobs on them. So they do not. Everybody good? Yep.

[Audience member: question]

1.9 will have this. If you get 1.9 to use spec, this will just start happening to you. You can turn it off in the REPL. I think it is on in the REPL, and off over wires, by default right now.

OK, so we see exercise at work.

[Time 0:52:54]

```
slide title: Context independence

+ arbitrary composition, subsets, intersections
```

```
+ reuse vs restatement

+ flow, with early error detection

+ self-describing data


          _flexibility_
```

So I already talked about a lot of these points. I am not going to re-dig into them. So we have arbitrary composition: a, b, c. b, c, d. a, d, e. We do all that. We can subset things. We can have intersections.

We do not need to make named things that restate the key information in order to get these. In addition, we can dynamically make unions of things that satisfy two specs, and we do not need to ever have made "the union of these two specs, spec", necessarily. As long as the consumers can satisfy their specs, we are fine, because they are open.

[Audience member: question]

Somebody is going to give a talk about that at the Conj. Yes, he has apparently turned spec into a type system. Spec is *not* a type system.

I did not ask. Are there any Haskell or Idris people, or whatever? Haskell. OK.

[Audience member: question]

Yeah, that is OK. I am going to bash on versions more, later. It is great. Obviously, any tool that can help you check is really good. The thing I would like to see at the end of the day with spec, is that if you have broken the spec from before, you do not use that name again. Call it dash 2, whatever.

[Time 0:54:42]

How many people ever did COM programming back in the dark ages? One of the things COM got right was: you do not change stuff like that. There was 2 and 3, and ex. They called it "my thing, ex". It is better today. My thing ex 2.

People that did that, how often did you get to 3? Or exx? Or ex2? Not very often. It is pretty rare that you break things that way. But if you really do break things, stop using the same name. That is just update in place programming. It really is. And having the user have to have this out of band information about versions, and all of the crap around versions that pollute how dependencies work, it is a catastrophe.

And this is not just about code. How many people have versioned web services? You do! Come on. You can raise your hands. This is a standard thing right now. It could be better, right? It would be better a lot of times if the entry points just were called 2. What is better about it? Your old clients could still call 1, because it is not gone. You can support them both transparently. You would not have to have this event: Oh, we are starting API 2 next Tuesday. Make sure you move all of your clients to API 2. Because it is out of band.

So versions are really problematic, but you need something to be able to make decisions about: Can I use the same name, or not?

[Audience member: question]

Liquid Haskell is trying to do as much of this as it can ahead of time. Yeah, it is.

[Audience member: question]

Yeah, sure. So there is a bunch of cool work being done with these SAT solvers and whatnot to provide sort of ... It is sort of the same space, this generative thing to go figure out. But yeah, they are trying to do more of this early. And there is a subset of this that it can do. Obviously parameterized types can do things this

cannot do, but this can do arbitrary things, which it will take a long time for SAT solvers to figure out. There is sort of a world of things that SAT solvers can do, and a world of things that they are going to take a long time before they can do.

[Look for "solver" on this Wikipedia page for a description of the capabilities of, and a list of, SAT solvers.

https://en.wikipedia.org/wiki/Boolean_satisfiability_problem

SMT solvers are used in Liquid Haskell, according to its README here: https://github.com/ucsd-progsys/liquidhaskell

SMT solvers can do everything SAT solvers can, and sometimes more, e.g. handle integers, real numbers, strings, etc. often in a more direct and efficient way than by representing them as bit vectors.]

[Time 0:57:20]

But I really have not talked about the context of use much here, because I really would not like this to be mapped in people's minds as a correspondency to a type system. A type system does certain things really well, but has almost no application outside of some context. This has a wide contextual application, and it does what it does.

So let us just talk a little bit more about those keys. So arbitrary composition, subsets, intersections.

The reuse vs. restatement. This is a high reuse system, then. You reuse the same keyword keys. But it also means, if you have a fully qualified key, you are going to nail down the semantics of that key, and you are not going to change them.

You get this flow. I talked about that before. We check "b" as early as we saw "b", against "b"s spec.

And in general this yields self describing data. Fully namespace qualified data is self describing. I think the sum of these things is a path towards flexibility. I guess I did not talk about it here.

How many people use fully qualified keys, all of the time? Not too many. It is great, but I do not think this is common practice. What is more common practice? Not doing that.

So what happens? Are you stuck? And it ends up, no you are not stuck. I will go back to here.

[earlier slide defining spec for ::m]

There is a require dash un [:req-un], and opt dash un [:opt-un], which is, "I require these keys unqualified." But what it takes is these same qualified names. And what you are saying there is: when "m" sees unqualified "x", it means user/x, and go look up user/x's spec and use it. So it is a connection, because you cannot find what "x" means, with no qualifier. Because we will just race to define what "x" means and yell at each other.

[Time 0:59:21]

So what you are saying is: for this map spec, "x" means user/x, unqualified "x" means user/x.

[Audience member: question]

Yes, you can say the spec for "b" is the spec for "a", is "a" spec. Yes. Pretty much anywhere you see a predicate here, you can put a spec.

[Audience member: question]

There is a function called merge. It may not be in this talk. There is a function called merge, which is to actually go and take two map specs and say, yeah, both.

[Audience member: What if there is a conflict?]

That is up to you.

[Audience member: long question]

No, you do not. You would not.

[Audience member: long question continued]

Yeah, so I hate the context dependence of JSON. edn says, "That was a *big* mistake." It is just awful. And namespaced keywords and the way edn works, and the way extensibility works in edn, is completely about that being evil. Because JSON can only use what JavaScript had, they end up with this context dependence. So that nesting is strictly like context qualifying.

And there is lots of context qualifying. People use prefix strings, and all kinds of stuff, or "if it has the letters D A T E in it in a row, use this parser." So there is just a lot of "ick" there.

There are sort of two problems. One is, I do not think you would be doing that in Clojure and edn in the first place, but if you have to deal with it, the mapping to spec allows you to deal with it. You could say, "I am accepting these things." Up at the top you say: the db spec says these unqualified keys mean these other things, and you sort of connect it all back to a namespace qualified world, using the contexts to make the mapping. But you end up with something that can do everything I talked about so far except catch that "b" error. But everything else you can do.

[Time 1:02:16]

So the support for unqualified keys in spec is actually quite good. You do not lose any of the power, except the ability to detect that "b" is wrong, early, because if it is all context dependent, you have to wait for the context validator to run, the context-specific validator to run.

[Audience member: question]

Yeah, you can say: int or float. There is a number predicate already that is int or float. But sure, you could "or" those two things together. And if you generated it, you get some ints and some floats.

[Audience member: question]

You can use "or", and "or" has all the power of disjunction.

[Audience member: question]

There are no references. This is functional programming with values, so there is no reference stuff.

But whereas the structural predicates themselves are not going to do what you want, "and" the thing you want to double check later, you can do. Just use "and". Say it has this structural thing, "and" let me make sure all friends are mutual. And that is a predicate you can write that would run against the structure. But except for using "and", there is no support for that check inside this. You use composition to get that.

But you can. You can check out arbitrary stuff. Sure. You can go and double check, for instance, that: mutual friendship. If that was just something you had, you have a predicate called mutual-friends? you would say "and" it is a mapping of user name to friend list, "and" then mutual-friends? And can get it that way.

[Time 1:04:25]

So you can use real code. That is the nice thing about this. You can write code and do arbitrary stuff. That is why we use Lisp, right? Eventually we just want the power. You can make the fanciest system you want. It is only going to include the things it includes. Tomorrow when I want something else, I do not want to have to do research and extend the compiler. I want to just write code.

[Time 1:04:53]

`slide title: Syntax`

```
    1565-75; short for earlier syntaxis < Late Latin <
         Greek sýntaxis an arranging in order

+ regex ops

cat - concatenation of predicates/patterns
alt - choice among alternative predicates/patterns
* - 0 or more of a predicate/pattern
+ - 1 or more of a predicate/pattern
? - 0 or 1 of a predicate/pattern
& - qualify a regex op with further predicates
```

Let us talk about syntax. So we talked about maps. And we have homogeneous collections. We have coll-of and things like that. And that is what you use. If you want a vector of integers, you are not going to use this. This is for syntax.

What does syntax mean? The Greeks and the Romans, they figured out what everything meant, and they put it in words, and we pillaged all of the roots. But if you go back, you end up with a lot of resources to use.

And so the word "syntax" actually means "an arranging in order", where the order is where the meaning comes from.

[Audience member: question]

So they do not actually use syntax for their language as much, but they know what it is. But we use it in programming a lot, in code, and hopefully almost not at all in information. This is a big difference. If you are going to make a wire spec, I hope you do not need to use this. The first thing I will send you will be that, and then the third thing will be this or that. No one wants to be on the other end of that.

So why do we use syntax? We use it in our functions when we have multiple arity languages like Clojure. We use it in our functions for positional calling, because positional calling is really convenient, and the scope is small, and the documentation is right there. So it is not like being on the other end of the wire, where the third thing must be "x".

But I have already shown you a ton of code that was positional. Like def, the first thing is the name and the second thing is a spec or a predicate. That is syntax right there. So verb, thing, predicate. The order matters.

[Time 1:06:50]

So when we are dealing with syntax ... So when would that be? When we are dealing with languages, like Clojure itself, or DSLs. When we are making our own languages, we might want that same convenience of syntax for our users, where we want to provide order matters.

So what do we do? And it ends up, again, there is this beautiful thing. We used set logic for maps. Sets of keys are what maps are in spec. And for order, regex is the math of order. It can describe all regular grammars that involve ordered things.

["regex" is a common abbreviation for "regular expression": https://en.wikipedia.org/wiki/Regular_expression]

So the trick is, regexes to most people mean strings. And the jumping point for that is, now just imagine that instead of being strings and saying, "it is this character or that character, or one of these characters," you can say predicates. That is the first leap you have to make. And the other is that they nest. So they can be subexpressions, which also are predicative.

And when you do that, then you can say, well I understand the idea of regexes applied to predicates.

[Audience member: question]

Yes, because regular expressions have nice syntax. Because Spec is a language itself.

[Audience member: question]

So one of the problems with grammars is, you always have two things. The grammar, it does not work in-line. It does not work like code. So regexes are much more Lispy. They really are much more Lispy.

With grammars it is, "here is your grammar," and now combine it. And regexes can be in-line.

[Time 1:08:57]

[Audience member: question]

That is an implementation detail.

So what are the operations? Concatenation. This, then that, then that, then that.

Alternation. And here we are talking, it is not "or". We are talking about regular expression alternation. This pattern could appear, that pattern could appear. And those patterns dictate what else could appear, because they are going to turn into expansions. So it is like "or" that splices in place.

Repetitions. 0 or more. 1 or more is not even primitive. 0 or 1. Those are not primitive. Really regex is just the first three of these. So you have those two.

Then there is a special one called and [&], which extends this system of regexes to say, "it is this regex, plus these predicates," which are not regex. But we want them to be true anyway.

[Audience member: question]

No, it is to specify specifications for syntaxes.

[Time 1:10:13]

slide:

```
(s/def ::a int?)
(s/def ::b int?)
(s/def ::c int?)
(s/def ::even? (s/and int? even?))
(s/def ::odd? (s/and int? odd?))

(s/def ::syntax
  (s/cat
    :forty-two #{42}
    :odds (s/+ ::odd?)
    :m (s/keys :req-un [::a ::b ::c])
    :oes (s/& (s/* (s/cat :o ::odd? :e ::even?))
              #(< (count %) 3))
    :ex (s/* (s/alt :o ::odd? :e ::even?))))

(s/explain ::syntax [42 11 13 15 {:a 1} 1 2 3 42 43 44 11])

[Note: The output below is difficult to read on the talk video.  It
was created with officially released Clojure 1.9.0, after the talk.]

In: [4] val: {:a 1} fails spec: :user/syntax at: [:m] predicate: (contains? % :b)
In: [4] val: {:a 1} fails spec: :user/syntax at: [:m] predicate: (contains? % :c)
```

Let us look at one. So what we are going to say here is, we expect this string of numbers. I am not showing you the syntax for code, but I expect a string of numbers. And we will start from the middle here.

This says: My syntax is, you must start with 42. And then you can have one or more odd numbers. And then I want to see a map that must have "a", "b", and "c", unqualified, in it. So we can look back up above for the specs of the keywords.

And then I want some odds and evens. And here I am going to say: I want to see an odd and an even, in a row, and as many of them as you want to supply, to a max of 3. So that is the use of the "and" [&]. So I say odd, even, odd, even, odd, even, but when the count of that set reaches 3, then we are done. Got that?

And then the rest, I am just going to say: as long as it alternates between it is either an odd or an even. Not alternates, sorry. It can be an odd or an even, and we will just take all of the rest there.

So the first thing I am going to do is just try that here. So let us just look at the numbers and see if it works. So we got a 42. Then we had 11, so we are good. And 13 and 15, that is a bunch of odds. We are still good on that next thing. Then a map comes up, so we must be done with the odds.

We are going to say, what is the spec for that map? It needs "a", "b", and "c". OK, we can already see what our problem is going to be, but explain will tell us. And we are done.

[Time 1:12:02]

So we run explain, and it says no. At the 4th index, the 5th value, you gave us this map. It was supposed to match the "m" syntax from here [maybe "m" spec?]. And it does not. It is missing "b", and it is missing "c". So this is a regular expression parse of this.

So I have talked a bunch about the difference between validation and destructuring. So now we are in a place where, wow. Imagine if you had to consume these syntaxes in your code. And you ran the validator, and instead of saying no it said, "yes, it is valid". What is your code processing this data structure going to look like? The same. It is going to have all of this trickiness to it, right? Because it is going to need to say how many odds did I get, and is it not an odd any more, and you are going to rewrite the regex logic in your parser. That is really unfortunate.

[Time 1:13:05]

```
slide title: Destructuring

(s/def ::syntax
  (s/cat
    :forty-two #{42}
    :odds (s/+ ::odd?)
    :m (s/keys :req-un [::a ::b ::c])
    :oes (s/& (s/* (s/cat :o ::odd? :e ::even?))
              #(< (count %) 3))
    :ex (s/* (s/alt :o ::odd? :e ::even?))))

(def v [42 11 13 15 {:a 1 :b 2 :c 3} 1 2 3 42 43 44 11])

(s/valid? ::syntax v)
true

(s/conform ::syntax v)
{:forty-two 42,
 :odds [11 13 15],
 :m {:a 1, :b 2, :c 3},
```

```
:oes [{:o 1, :e 2} {:o 3, :e 42}],
:ex [[:o 43] [:e 44] [:o 11]]}
```

So this is the same as before [the def of ::syntax is the same on this slide as on the previous slide]. You do not have to try to read it again. And now we are going to have a valid one. So 42, three odds, a valid map, odd, even, odd, even, and odd, even, odd. So we have one, two, three pairings, and then we switch. Oh, so we only do two, because we want less than three, fewer than three.

So it is valid. We say it is valid. If this was the front door to our system, we would be done. It is valid. Then we have to process it, though, we have no help. So now we are looking at the power feature on validation in spec, which is conform.

"conform" does not give you a simple yes or no. For every predicate, and in particular for the composite predicates "and", "or", the map predicates, and all the regex predicates, there is a definition of what that spec does when conforming data. And it is not going to give you the same data back. It will give you labeled, or what we call conformed, data back.

So let us look at the conformed value of the spec. So the first thing that is interesting is: the spec for syntax was specifying what, fundamentally, with "cat"? What is it saying the top level data structure should be, that satisfies the spec?

A sequence. And what does it return? What does conform return? Not a sequence. Conform of cat always returns a map.

And the other thing we will notice about this regex in the first place. I did not even talk about it while we are here, because we did not know why, but one of the things you might have wondered was, "Why are there all of those names?" Why isn't this regex just 42, odd? plus, "m", then o e star, whatever, star? Why did you make me label these things?

[Time 1:15:20]

And the reason is: because I am going to give you back those labels when I conform this. And if I did not match a pattern, the key would not be in the map. So only things we found are there, and everything that we found is labeled.

[Audience member: question]

No, there is no generation yet. So this is just, instead of just saying yes or no in validation, we are going to conform the data. So I went back a screen, but the spec is right up here. Let us look at the spec.

So the overall cat has every element in the sequence labeled. "+" has no labels. "keys", well keys is a set of labels in the first place. Maps are self labeling. "cat" again, nested, has labels. Star does not have any labels, but "alt" has labels. Way back at the beginning of this now long talk, I talked about choice and paths. I am not going to dig too much into it, but that is choice and paths in action.

Everywhere in spec there is a choice, there must be a label, and when you get conformed data back, the paths will include the labels. And then you know which branches were taken. And your data is now sitting essentially destructured for you in a way that makes ... Who would prefer to consume this map versus that sequence? Me. For sure.

[Audience member: question]

It is not doing any work that "valid?" was not doing. It is just capturing it.

[Audience member: question]

Well I call it "conform".

[Audience member: question]

[Time 1:17:18]

Well it is more just like a label.

[Audience member: question]

Well, it is not exactly, because the keys thing, like “a”, “b”, “c”, they really are specs, like rules. These other things are just labels. We make you label the inside of “cat”. We make you label all the choice points in spec. And the reason why we do is: when we conform it, we give you the conformed value back labeled. Organized by the labels.

[Audience member: question]

No.

[Audience member: question]

No. It is just keys. It is in whatever order you get. It is a map. It is a Clojure map, so no order.

[Audience member: question]

No, it will give you the first valid conformation.

[Audience member: question]

No, we do not do that. It ends up that if you have a syntax for human beings to do that is like that, that is not going to be a happy syntax. It makes a lot more sense for strings than it does for this. I mean, I understand the idea, but I rejected it for this.

[Audience member: I have a question. If you were going to call conform without validate. Let us say instead of 42, the first value was 43.]

Yes.

[Audience member: Would it return false, or . . .]

No. It returns a distinguished invalid value. And there is a checker for that value.

[Audience member: question]

So it is a distinguished value. So the question is: what does it return when it does not conform? It cannot return nil, for instance, because that might be the conformed value of this certain spec. So it returns spec invalid, a keyword. I am sorry, a distinguished value. So that you can look at it and say: I know that cannot possibly be the conformed value, so it is invalid.

[Time 1:19:11]

[Audience member: question]

It is a way to look at it. So the question was: is it an abstract syntax tree for it? You can consider it that way. Certainly I am going to make the case that the beauty of using spec for deciding the user input for your DSL was valid is: you get this parser for free.

Whether or not you consider this an abstract syntax tree, it depends. It is not enriched with any other information. But it certainly is a tree with all of the branches labeled. So it is that much of an abstract syntax tree.

[Audience member: But if you needed to restore the order of the values, you can get that from the spec.]

If only there was a function called “unform”, which there is.

[Audience laughter]

which takes this conformed value and the spec, and it will return this back, which is so super cool, and so much fun.

[Audience member: question]

Yes, you do, as long as you are still conformant with the spec with your transformation, you can do transformations between

[Audience member: question]

You can call unform with a different spec, and you can conform that value to go back. You can also do transformations. So for instance let us say you had optional data. This is advanced spec now. So there is unform, and it will take the conformed value and the spec, and put it back into an original form.

But let us say you had a spec, and there was optional data, and your downstream consumers did not consider it optional. You could take user data, conform it. It is conformant, but they have left the options out. You could supply the options in this form [the return value of conform], then unform it and get back data that the next person wants to see with all of the defaults filled in. That is one application of that stuff.

[Time 1:21:24]

[Audience member: question]

Yes. So I spent all of that time early on and said: look where we use data in all of these places. So that is the wire. You were talking about a DSL, a language. Yes, yes. Yes yes yes yes yes! *All* application domains for spec. Exactly.

[Audience member: question]

Yes. Because otherwise you are not really destructuring. Otherwise you are still going to have this job. So a lot of the conditionality about processing this, or at least duplicating the work of parsing this, is not necessary in your handling code.

So conform. When do we call this? Whenever you want. While you are developing, you can call it. If you have written a DSL, you can call it and use this as your parser. Like the first ten seconds, David Nolen had this library, he took his entire parser for Om next and threw it away. He spec'd his protocol and got a free parser with way better error messages than he had before. I recommend doing that.

[Time 1:22:53]

```
slide:

(s/exercise ::syntax)

[Note: The output below is difficult to read on the talk video.  It
was created with officially released Clojure 1.9.0, after the talk.]

([(42 -1 {:a 0, :b 0, :c 0})
  {:forty-two 42, :odds [-1], :m {:a 0, :b 0, :c 0}}]
 [(42 -1 -1 {:a 0, :b -1, :c -1} -1 -2 -1)
  {:forty-two 42,
   :odds [-1 -1],
   :m {:a 0, :b -1, :c -1},
   :oes [{:o -1, :e -2}],
   :ex [[:o -1]]}]
 [(42 -1 {:a 0, :b 0, :c -1} -1 0 -2 0)
  {:forty-two 42,
```

```
   :odds [-1],
   :m {:a 0, :b 0, :c -1},
   :oes [{:o -1, :e 0}],
   :ex [[:e -2] [:e 0]]}]

 ...)
```

So all of this exercise stuff, it works. It does not matter how thick and gooey your specs get. We can still generate stuff. We can still exercise that spec and we get exemplars.

So what if you made wire protocol specs, and you need to test your wire? Hmmm. Do you want to write stuff that generates data for your thing, or do you want to take the spec that you wrote that says: is the wire OK, turn it around and say "make stuff", and send it to the wire? That is what I want to do.

If you want to look at the language you said you were going to write, and you wrote the spec for the DSL, you could push this exercise button and get programs in your DSL, written for you. They would be gobbledygook, but they would have the right shape. Which can really help you, because you can say, "ooh, look. That is ambiguous, and I do not think people are going to want to write that." Blah, blah, blah.

[Audience member: Can you teach exercise about the edge cases you want to stress . . .]

So that would be about the generators used. Right now we do not have generator overrides in exercise, but it is on the punch list. So there are a lot of places where we do allow generator overrides. Very particularly we allow them in the testing framework. And if you call generate yourself by hand, and exercise is just a wrapper for calling generate. If you call generate yourself, you can supply overrides. When you supply overrides, you can make things that generate only nice data, only subsets of data, only so far, various kinds of limits.

So if you want to generate data that does not look horrible . . . Because these generators by default, they are oriented towards testing, so they are going to try to push . . .

[Audience member: question]

[Time 1:25:01]

So you can use the paths system, which I am not really talking about tonight, to talk about a sub-part of a spec, and provide a generator override just for that part. There is a lot more stuff, let us just say.

[Audience member: question]

No, it basically will try a certain number of times and say, "I cannot satisfy this spec, and I am giving up." It just punts. It does not run forever. It will stop and say no, I cannot do it. And that is when you say, "Oh, I need a custom . . . I need to help the generator out a little bit."

And that often does not mean doing the whole job. It can often mean just narrowing the space somewhat, and letting the rest of the auto stuff kick in. Because whatever you generate is still subject to validation. So you could make a partial generator, and it gets you close, so 80% of the values that this generator generates are valid. "valid?" gets called on it, and invalid values get thrown away. So it is not like you have to write a perfect generator.

But you could at least narrow the domain so that it can succeed, and then the normal predicate based filtering will happen, which is what is built in. Which is quite nice. You can just take care of the hard part. You are never going to make a correct looking URL. Although there are cool libraries that have generators for regexes.

[Audience member: question]

It is just a number. There is a knob. You can set it.

[Audience member: question]

No dumb questions.

[Audience member: question]

[Time 1:26:57]

So the question is: can you use this to make a constraint satisfaction DSL? Whereas I said it is sort of like a logic engine, obviously it has generative capabilities ... Well, yes you could. You could. Is that like a recommended way to do it? Probably not.

But some fun stuff has been done. Christophe Grand took some example program that played Set, the game Set where you match three things in a row. It is something that has been implemented in a bunch of languages, and they compare the implementations. So there is a Clojure one, which is just sort of your ordinary functional, but somewhat imperative: make a thing, generate the things, call whatever.

And he turned it into a spec program, which was just like a bunch of specs, and a call to generate, and validate. And it played Set. It did everything the other ones did. It shuffled the cards. It dealt out hands. It found the sets.

So you could treat this like core.logic or miniKanren, or something like that. And it has cool properties. One of the properties is that it uses random generation for exploration, as opposed to the typical Prolog tree exploration.

[Time 1:28:20]

```
slide title: Functional

+ arg and ret 'types' say little about what a function
  _does_

+ a spec for a function includes predicates about
  _relationship_

  + between args

  + between args and ret

+ spec supports both
```

But there is more!

[Audience member: question]

I think it gets you through the space faster. Invariably in something like miniKanren or whatever, they eventually get to a constrained domains, and eventually they add a randomizer. They always do. It always comes.

[Audience member: question]

So you get around faster, let us just say. That is the simplest way to say it.

Let me keep going, because it is already taking too long.

So I talked earlier about being structural and functional. So far I have only talked about structural stuff. So one of the things I think it would be too easy to do is look at spec and say: this is a poor man's dynamic type system. You showed me checks for int, checks for stuff that I could definitely do with type systems. And especially with some of these more advanced systems, you can do a lot.

One of the things that I think is a lot trickier is to talk about this part, which is: if you just say the type it takes this, and it returns that, that does not really say what the function *does*. And what I mean "does" is, does from the person who is paying you to write this program does. And it does not tell you if this function is correct, again from the perspective of the person who is paying you to write this program, who is not going to buy into mathematical correctness as satisfying the types, it is a solid program. In the end, the people who pay you, they care about the difference between less than and greater than, and only a few type systems do.

So what does a function do? Well there are a bunch of stuff that matters in terms of making better statements about the arguments. Because a lot of times there are preconditions on the relationships between the arguments. But what a function does fundamentally is a predicate over the relationship between the arguments and the return value. It crosses values. So this value crossing is an important thing, and spec supports both of these things.

[Time 1:30:22]

```
slide title: fn specs

(defn ranged-rand
  "Returns random int in range start <= rand < end"
  [start end]
  (+ start (long (rand (- end start)))))

(s/fdef ranged-rand
  :args (s/and (s/cat :start int? :end int?)
               #(< (:start %) (:end %)))
  :ret int?
  :fn (s/and #(>= (:ret %) (-> % :args :start))
             #(< (:ret %) (-> % :args :end))))
```

So imagine we were trying to write a function called ranged-rand, and it takes a start number and an end number, and it is supposed to generate things greater than or equal to the start, and less than the end. And we just write ordinary code.

This is the other thing about spec. That is ordinary code. Maybe the person who wrote it did not spec it. Who could write the spec for it? Anybody else. These things are not connected. It does not get compiled into the Var space when the code gets compiled. You will very much be encountering unspec'd code from other people. You can write the specs for that code, either to validate stuff, or just to validate your understanding of what it is supposed to do.

So we have the function. This does what it is supposed to do. Then we want to spec the behavior of the function. And it ends up there are three aspects to the behavior of a function that you might want to spec. You might want to spec the argument list. You might want to spec the return value. And then this new thing I just talked about, which is: what does a function do? The actual function of the function. The transformation.

So the args spec is a spec of the argument list, and they will almost always be a kind of a regex spec, because our argument lists in Clojure are positional, and they are variadic. So it ends up regexes are a good way to spec the argument lists, which are syntactic.

So we are saying it has an int and an int. We are calling the first one start and the second one end. And now, isn't it nice we have those labels, because what is the next thing we want to do? We want to make sure the start is less than the end. So we can add that additional predicate on the values. So now not only are we spec'ing the values, but also the relationship.

The return value is just an int. And then we say ranged-rand is working when it returns a value that is greater than the start and less than the end. And that is what this says. So what actually gets passed to the fn spec

are the conformed arguments and the conformed return value, which means you get labeled conformed stuff, and you can write predicates on those conformed values. And that is what this last thing is.

[Time 1:32:43]

The conformed value of the arguments will have something called start and end, and the conformed value of the return will have something called ret. And you can use those keys to talk about the conformed values and then make predicates over those. This is a lot of power for talking about whether or not functions are correct from a stakeholder perspective.

Now some things are harder than others. The testing of this, John Hughes has been in this space for a long time with QuickCheck and things like that. There are a lot of relationships between this and property specs, for property based testing. And some things have straightforward properties. Other times you are like, well my property seems to be a reimplementation of my implementation. And that is true, but sometimes you can make a model. None of those problems go away. You have a path at least to talk about this, if you want. That is optional.

Are there any questions on this?

[Audience member: It sounds a lot like contract programming. What is the difference, or what did you add?]

There are lots of flavors of contract based programming, so you would have to talk about a particular flavor. But yeah, this is like that. Absolutely.

[Audience member: question]

For data. Yes. Right. But it is not just types, that is the thing. This fn one is not just types.

[Audience member: question]

So ranged-rand is just a regular function. We have not touched it. When you say "fdef", you are saying: I am making a spec for the function named ranged-rand. That is what fdef does.

[Audience member: question]

No. It is actually restating it here. I could have called it fred and ethel. It does not look at this. It does not look at the function definition. Yeah, these labels could be anything I want. It could be x and y. No, it does not look at the source code for the function, because you could write spec first, then write the function and see if you did it right. That will become what you do.

[Time 1:35:15]

```
slide title: Enhanced docs

(doc ranged-rand)

[Note: The output below is difficult to read on the talk video.  It
was created with officially released Clojure 1.9.0, after the talk.]

-------------------------
user/ranged-rand
([start end])
  Returns random int in range start <= rand < end
Spec
  args: (and (cat :start int? :end int?) (< (:start %) (:end %)))
  ret: int?
  fn: (and (>= (:ret %) (-> % :args :start)) (< (:ret %) (-> % :args :end)))
```

I am going to keep moving, because it is going slowly.

When you do this, and you want to get the docs for ranged-rand, if there is a spec, the docs will be enhanced. This function will also return the spec, in addition to the doc string.

[Time 1:35:29]

slide:

```
(s/exercise-fn `ranged-rand)
;;[args ret]
([(-1 0) -1]
 [(-2 0) -2]
 [(0 1) 0]
 [(-4 -3) -4]
 [(-62 7) -24]
 [(-1 5) 1]
 [(-7 -6) -7]
 [(3 39) 22]
 [(-82 4) -10]
 [(0 2) 1]
```

And now we can exercise functions. So we are saying: exercise this function. When you exercise fn, the pair of things you are going to see is a generated argument list, and the conformed return value. So when I called ranged-rand with -1 and 0, I got -1. And blah, blah, blah, etc. etc. etc.

So you can exercise your functions. Again, as soon as you have this, and you use it, you will use it all of the time. Write your function, and the first thing you will do is do this. Because what do you do otherwise? You type these things in the REPL, and you look at it yourself manually. So calling exercise fn is way faster, and will exercise more of your stuff.

[Audience member: So this thing here has nothing to do with spec on the last page?]

Yeah, it uses both. So it is calling this function, and it is using the spec. It is using the spec for two purposes. It is using the spec to generate arguments to call the function with, and it is using the spec to validate, to conform the values that are returned. So everything on this page, it is using.

Because it is calling ranged-rand. So this is a generated argument list. It looked at the args spec and said: one number, another number. But you notice the first number is always less than the second number, because it used that spec.

[Audience member: I was confused because I do not see the spec being passed to this.]

Right. So that is a good question. So look at the fdef. It is not a keyword. It is an ordinary symbol. When you say fdef you are going to pass a fully qualified symbol name, or it will use the namespace that it is in, and it is going to imply a relationship between that name and the function, which has the same namespace qualified name. Because all functions are in namespaces.

So when we use the backtick there, that is saying: user/ranged-rand, and that is going to find the function called that, and the spec called that. Use the spec to generate arguments, call the function, use the spec to conform the return values, and conform it, and return what came out. If your function does not work, it will tell you.

[Audience member: question]

So it is not going to say what will happen when the data is bad, but there are a bunch of things about calling a function, and a function running, that can be tested. We are going to talk about that in a second.

[Time 1:38:18]

```
slide title: Robustness

+ Does a program do what its stakeholders think it
  should?

+ spec approach is _ahead-of-time_ generative testing

  + _not runtime checking_!

+ check - does a function work when given correct
  args?

+ instrument - is it being called with correct args?

+ assert - dev/test-time check of spec predicates
```

So we are trying to approach robustness from the perspective of stakeholders. That is what you should be working for with a robust program. It is not about satisfying yourself. It is not about satisfying a compiler. It is about satisfying your business requirements. So as much as you can capture in specs, it is a good deal.

The key here is that: it is not a type system. It is not for use at the same point. It actually cannot do the job of a type system, which flows information about the types, and statically is able to do some checks against the validity of a program versus the types that are defined.

Instead, this is a set of tools for doing robustness checking. So how do we do that? What we do not do, is we do not turn on all of this stuff at run time, and ship a program with all of this overhead. That is *not* what spec is about. Spec is about making sure your program works before you ship it. And trying to force out any problems with your program.

So how are we going to do that? The way we are going to do that is with ahead-of-time generative testing, not run-time checking. Spec is not run-time type checking.

[Time 1:39:37]

So what are the tools we have for that? The main tool we have is something called "check". And it is in a namespace called test. And yes, it is very much based around QuickCheck. And it uses Clojure's test.check, which was a port of QuickCheck.

So the purpose of check is to say: if I give this function correct arguments, does it do what its function spec says? Is the function itself correct? [last sentence guessed from lip-reading while audio dropped out]

We can test that with generative testing. We can generate a whole bunch of argument lists, call the function, validate the return spec and the function spec, if it is present, and make sure everything is OK. So that is check.

The other thing we have is called instrument. And instrument says: did you call this function correctly? Did you call it with arguments that satisfy its spec? And that is something that a type checker would check via flow analysis. We are going to test by turning on instrument during testing, or during interactive development, not at run-time.

So check calls a function generatively hundreds and thousands of times and makes sure it does what it is supposed to. instrument says: if anybody calls this function, check what they passed, make sure it satisfies the arg spec for the function.

And finally we have something called assert, which allows you to sort of make dev time assertions about things that will not be present in production.

[Time 1:41:05]

slide:

```
(require '[clojure.spec.test :as test])

;; 1000 tests you didn't have to write
(-> `ranged-rand test/check test/summarize-results)
{:tbd 1, :tbd 1}

(test/instrument `ranged-rand)

;; bad caller
(ranged-rand 42 11)

[long output giving details about how you violated the args part of
ranged-rand spec, because 42 is less than 11]
```

So that is what this looks like. We say require spec.test as test, and then we are using the piping operator. We are saying: pass ranged-rand to test/check, and then summarize the results. test/check can test whole namespaces, and everything that has been spec'd, and various other sets of things. So this is a summary of running one function through it.

So this will call it a thousand times with a bunch of different arguments. And if it does not pass, it does all of the QuickCheck style shrinking, so you get the smallest possible argument list that causes your function to fail, and a nice pretty version of that. And it ends up that ranged-rand worked, and this just says: it is fine. But that is a lot of tests you did not have to write. And the tests are way better, and way weirder, than the ones you would think to write. And weird generated tests do a better job of testing your stuff than: I tried it with 42 and it returned 7, or something.

Then the other thing we can do is we can instrument ranged-rand. Again this is not something you do in production, but you might do while you are running the test suite of a different namespace, or just leave instrument on during REPL-based development. You can say instrument this namespace, instrument everything that is spec'd. So you just turn that on and keep programming. And as you program, if you ever call a function the wrong way, it will tell you that.

So now we are saying: I called ranged-rand wrong. I did. It is nothing wrong with ranged-rand, but this thing I typed in my REPL did it wrong. And if instrument is on, it will give me the failure report. And there is something called unstrument which turns that off.

[Time 1:42:46]

[Audience member: question]

Yeah. So we are going to do that in pieces. We are starting with the macros, because that was the thing that people wanted better error reporting around. But yeah, there will be specs for a lot of the language functions.

[Audience member: question]

What? No you have to say instrument. It would never be on by default, because I am going to have a slide here where I am going to say: *do not* do this during production.

[Time 1:43:14]

```
slide title: spec in Production

+ do not use check, instrument or (turn on)
  assert in production

  + they are for making sure your program works -
    use in advance

+ but spec _can_ play a runtime role

  + validating inputs from external systems or users

  + as part of DSL parsing

  + to generate data for various purposes
```

There you go. Don't do this in production! I am going to make T-shirts. Don't call check or instrument or turn on assert in production.

[Audience member: question]

No, no, no. So you have a lot of power. You have regex, you have nested specs, you can use sets. You can say this could be 42 or 47. All sets are predicates. So you can use just sets. I expect fred, ethel, or lucy. You can do that, and you can even deal with variadics without "or", because you can put "alt" in this thing. You can deal with multiple arities. You will find it highly co-aligned.

That is not to say that it will do everything. A lot of times when there are sort of sub-domain relationships between arguments, you are going to need to help it out. But you have all of test.check there, and that has combinators like bind that allow you to build a model, and then have the model flow through the subsequent generators.

So when you want to make dependent data – somebody talked earlier about: I want mutual friend lists. But a lot of times you have dependent data, like there is a set of things here, and then below, everything should be one of those things. So that you could do with bind. You can generate a model, and then have other generators dependent on the generation of that. And that will allow you to make correlated relationships. So they are connected.

And also it will shrink, which is super cool. So yeah, all of that stuff is in there.

[Time 1:45:11]

So don't use this stuff in production. The recipe here is: check is for running tests before you ship. It is generative testing. It is actually pretty expensive to run. But it is quite comprehensive, and automatic, and mechanical.

instrument is something you can have on while you are developing. It is something you can turn on during testing, including during testing that uses check. I am testing this thing, and it will call other things, and it will make sure we are calling them correctly. Just instrument everything. Then you turn it off and you ship your program.

But there are plenty of things you can do at run-time, like your program might parse data that comes in over the wire. There is no problem. You can validate inputs from human beings. You can have a DSL, and use spec to do your DSL. You can call conform on stuff that comes over wires. You can generate data at run-time, if it was useful to do that.

But the testing part and the validation part of your program is to be done in advance. If you do this at

run-time, then all of the other languages will laugh at us. So don't do it.

[Audience laughter]

It is not what it is for. The reason why all of this generation stuff is in there is because it is the path to doing robustness in advance. If there was not a generator, you would only have validators. You would put them everywhere, you would have them turned on all of the time. But you do not have to do that because of generation.

[Time 1:46:40]

```
slide title: Other features

Custom generators

multi-specs

keys*

nilable

instrument gen overrides, replacements and stubs

unform

fdef

the gen namespace

merge

the registry, tuple, conformers ...
```

Although we talked about a lot, and for a long time, there is way more stuff. unform we talked about a little bit, custom generation we talked about a little bit. There is a keys* which does keyword args style stuff, the same as the maps. There is nilability. You can override generators during instrument. You can do replacements. You can do stubbing.

This is really cool stuff. How many people have ever been frustrated when you are trying to specify an API that calls over a wire. And you have got to mock things out, or whatever. It is really difficult.

So it is really easy to take the spec for what you are supposed to send, and the spec for what you are supposed to get, and build a little U-turner thingy out of spec that does not use the wire at all, and validates both things, and gives you back stuff that you expect. It is quite straightforward, just by using instrument plus check, with overrides in place. Just sort of slice off the wire, and put a validate, generate U-turn in there.

Unform we talked about and blah, blah, blah.

[Time 1:47:51]

```
slide title: Leveraging spec

+ don't put spec in a box

  + 'dynamic type system', 'contract system', 'schema
    validator'
```

```
+ spec is a set of small tools oriented at maximizing
  the power and flexibility of _dynamic_ development

+ furthers Clojure's pursuit of simplicity and
  robustness
```

So this is just a flavor. I started with all of that stuff about data. The reason why Clojure exists, and the reason why it is the way it is, is because you can write things like spec, which is a small-ish thing. It really is small. But look where we can use it! Everywhere. It is just like a library. And it is just like what people find in their own programs. The generality here has huge benefits. Yes, a spec is something you have to invest in to write, but the amount of leverage you get for that investment is huge.

You get to exercise your functions. You get conformers. You get parsers. You get generation. It works over wires. It works with data from other people. It is just much different.

The other thing is: we looked at all of these functions a la carte. They are a la carte. You should think about: I can use spec to do that right now to just help me with something at the REPL. Or I can build a really cool testing framework that combines instrument and test and check and whatnot. So it is a set of tools that you can put together. It is not just a data structure validator. It is not a pretend type system. It is not a wire protocol checker, necessarily.

[Time 1:49:14]

```
slide title: 'just write a spec'

+ can be the simplest path to:

  + understanding something

  + getting a parser

  + getting data

  + mocking up a system

  + trying your new function
```

The one thing I would encourage you is to sort of think out of the box. A lot of times, once you have spec in hand, the easiest way to do something is just to write a spec. You look at some data like that 1, 2, 3, 5, map, you are like, "Ugh! I have got to write a parser." No! You do not have to. You do not have to write a parser. You get this weird data. The first thing you do, "Oh! I will just write a spec for that." I got a parser for free. I get conform for nothing.

I have some function that somebody wrote, and the docs are terrible. I think it does this. Well, spec it, and then run check on it. It does! Or no, it does not do that. That fails. Maybe it is broken. Maybe I do not want to use this library. It does not work.

You can get data. You can mock up a system. You can pretend you have wires and other people to talk to when you do not. You can just try out the thing that you just wrote.

[Time 1:50:04]

```
slide title: clojure.spec

       validation
```

```
  parsing

 generation

communication

destructuring

  testing

    ...
```

So the point of spec is, you apply some effort to specify things, you get a ton of leverage: validation, parsing, data generation, communication, destructuring, testing, etc. etc. etc.

And I think it validates the Clojure model of programming against data, because you see this kind of leverage, this kind of reuse. And I think, as Clojure programmers, you see this all of the time. But I think this makes it sort of clear *why* you use a dynamic language like Clojure.

So, thank you!

[Audience applause]

[Time 1:50:42]