Extracted from:

Property-Based Testing with PropEr, Erlang, and Elixir

Find Bugs Before Your Users Do

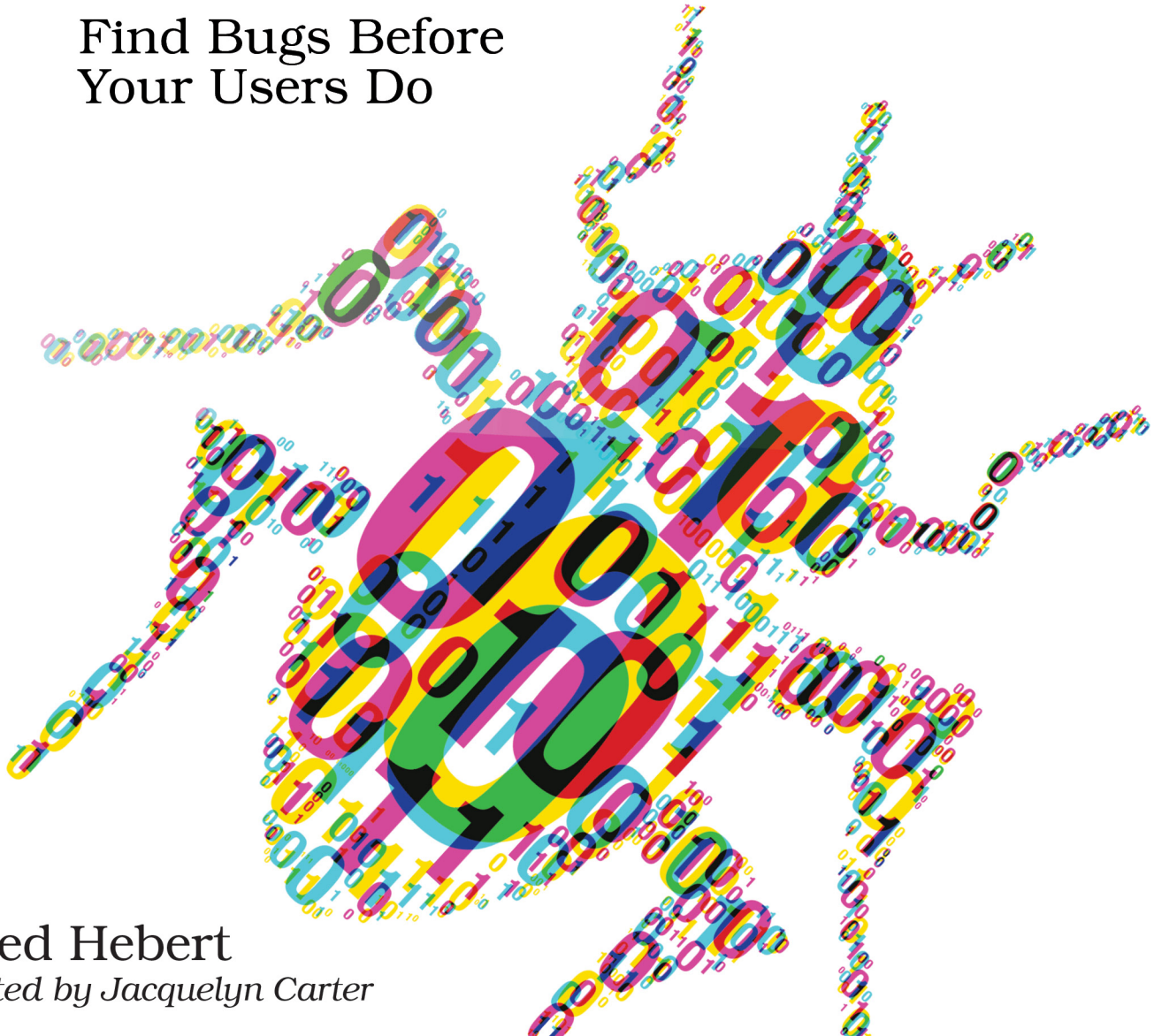
The Pragmatic Bookshelf

Raleigh, North Carolina

The Pragmatic Programmers

Property-Based Testing with PropEr, Erlang, and Elixir

Find Bugs Before Your Users Do



Fred Hebert
edited by Jacquelyn Carter

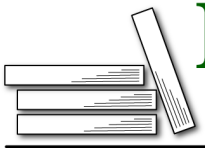
Property-Based Testing with PropEr, Erlang, and Elixir

Find Bugs Before Your Users Do

Fred Hebert

The Pragmatic Bookshelf

Raleigh, North Carolina



Our Pragmatic books, screencasts, and audio books can help you and your team create better software and have more fun. Visit us at *https://pragprog.com*.

The team that produced this book includes:

Publisher: Andy Hunt
VP of Operations: Janet Furlow
Managing Editor: Susan Conant
Development Editor: Jacquelyn Carter
Copy Editor: L. Sakhi MacMillan
Indexing: Potomac Indexing, LLC
Layout: Gilson Graphics

For sales, volume licensing, and support, please contact *support@pragprog.com*.

For international rights, please contact *rights@pragprog.com*.

ISBN-13: 978-1-68050-621-1
Book version: P1.0—January 2019

CSV Parsing

The first part of the program we'll work on is handling CSV conversion. No specific order is better than another in this case, and starting with CSV instead of filtering or email rendering is entirely arbitrary. In this section, we'll explore how to come up with the right type of properties for encoding and decoding CSV, and how to get decent generators for that task. We'll also see how regular example-based unit tests can be used to strengthen our properties, and see how each fares compared to the other.

CSV is a loose format that nobody really implements the same way. It's really a big mess, even though RFC 4180[5] tries to provide a simple specification:

- Each record is on a separate line, separated by *CRLF* (a \r followed by a \n).
- The last record of the file may or may not have a CRLF after it. (It is optional.)
- The first line of the file may be a header line, ended with a CRLF. In this case, the problem description includes a header, which will be assumed to always be there.
- Commas go between fields of a record.
- Any spaces are considered to be part of the record. (The example in the problem description doesn't respect that, as it adds a space after each comma even though it's clearly not part of the record.)
- Double quotes (") can be used to wrap a given field.
- Fields that contain line breaks (CRLF), double quotes, or commas must be wrapped in double quotes.
- All records in a document contain the same number of fields.
- A double quote within a double-quoted field can be escaped by preceding it with another double quote ("a""b" means a"b).
- Field values or header names can be empty.
- Valid characters for records include only

```
! #$%&'()*+-./0123456789:;<=>?@ABCDEFGHIJKLMNOPQRSTUVWXYZ[\\]^_`
abcdefghijklmnopqrstuvwxyz{|}~
```

Which means the official CSV specs won't let us have employees whose names don't fit that pattern, but if you want, you can always extend the tests later

5. https://tools.ietf.org/html/rfc4180

and improve things. For now though, we'll implement this specification, and as far as our program is concerned, whatever we find in the CSV file will be treated as correct.

For example, if a row contains `a,b,c`, we'll consider the three values to be `"a"`, `" b"`, and `" c"` with the leading spaces, and patch them up in our program, rather than modifying the CSV parser we'll write. We'll do this because, in the long run, it'll be simpler to reason about our system if all independent components are well-defined reusable units, and we instead only need to reason about adapters to glue them together. Having business-specific code and workarounds injected through all layers of the code base is usually a good way to write unmaintainable systems.

Out of the approaches we've seen in Chapter 3, *Thinking in Properties*, on page ?, we could try the following:

- Modeling—make a simpler less efficient version of CSV parsing and compare it to the real one.
- Generalizing example tests—a standard unit test would be dumping data, then reading it, and making sure it matches expectations; we need to generalize this so one property can be equivalent to all examples.
- Invariants—find a set of rules that put together represent CSV operations well enough.
- Symmetric properties—serialize and unserialize the data, ensuring results are the same.

The latter technique is the most interesting one for parsers and serializers, since we need encoded data to validate decoding, and that decoding is required to make sure encoding works well. Both sides will need to agree and be tested together no matter what. Plugging both into a single property tends to be ideal. All we need after that is to *anchor* the property with either a few traditional unit tests or simpler properties to make sure expectations are met.

Let's start by writing tests first, so we can think of properties before writing the code. Since we'll do an encoding/decoding sequence, generating Erlang terms that are encodable in CSV should be the first step. CSV contains rows of text records separated by commas. We'll start by writing generators for the text records themselves, and assemble them later. We'll currently stay with the simplest CSV encoding possible: everything is a string. How to handle integers, dates, and so on, tends to be application-specific.

Because CSV is a text-based format, it contains some escapable sequences, which turn out to always be problematic no matter what format you're handling. In CSV, as we've seen in the specification, escape sequences are done through wrapping strings in double quotes ("), with some special cases for escaping double quotes themselves. For now, let's not worry about it, besides making sure the case is well-represented in our data generators:

Erlang code/ResponsibleTesting/erlang/bday/test/prop_csv.erl

```
field() -> oneof([unquoted_text(), quotable_text()]).

unquoted_text() -> list(elements(textdata())).

quotable_text() -> list(elements([$\r, $\n, $", $,] ++ textdata())).

textdata() ->
    "ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789"
    ":;<=>?@ !#$%&'()*+-./[\\]^_`{|}~".
```

Elixir code/ResponsibleTesting/elixir/bday/test/csv_test.exs

```
def field() do
  oneof([unquoted_text(), quotable_text()])
end
```

```
# using charlists for the easy generation
def unquoted_text() do
  let chars <- list(elements(textdata())) do
    to_string(chars)
  end
end

def quotable_text() do
  let chars <- list(elements('\r\n",' ++ textdata())) do
    to_string(chars)
  end
end

def textdata() do
  'ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789' ++
    ':;<=>?@ !#$%&\'()*+-./[\\]^_`{|}~'
end
```

The field() generator depends on two other generators: unquoted_text() and quotable_text(). The former will be used to generate Erlang data that will require no known escape sequence in it once converted, whereas the latter will be used to generate sequences that may possibly require escaping (the four escapable characters are only present in this one). Both generators rely on textdata(), which contains all the valid characters allowed by the specification.

You'll note that we've put an Erlang string for textdata() with alphanumeric characters coming first, and that we pass it to list(elements()). This approach will randomly pick characters from textdata() to build a string, but what's interesting is what will happen when one of our tests fail. Because elements() shrinks toward the first elements of the list we pass to it, PropEr will try to generate counterexamples that are more easily human-readable when possible by limiting the number of special characters they contain. Rather than generating {#$%a~, it might try to generate ABFe#c once a test fails.

We can now put these records together. A CSV file will have two types of rows: a header on the first line, and then data entries in the following lines. In any CSV document, we expect the number of columns to be the same on all of the rows:

Erlang code/ResponsibleTesting/erlang/bday/test/prop_csv.erl

```
header(Size) -> vector(Size, name()).

record(Size) -> vector(Size, field()).

name() -> field().
```

Elixir code/ResponsibleTesting/elixir/bday/test/csv_test.exs

```
def header(size) do
  vector(size, name())
end

def record(size) do
  vector(size, field())
end

def name() do
  field()
end
```

Those generators basically generate the same types of strings for both headers and rows, with a known fixed length as an argument. name() is defined as field() because they have the same requirements specification-wise, but it's useful to give each generator a name according to its purpose: if we end up modifying or changing requirements on one of them, we can do so with minimal changes. We can then assemble all of that jolly stuff together into one list of maps that contain all the data we need:

Erlang code/ResponsibleTesting/erlang/bday/test/prop_csv.erl

```
csv_source() ->
    ?LET(Size, pos_integer(),
         ?LET(Keys, header(Size),
              list(entry(Size, Keys)))).

entry(Size, Keys) ->
    ?LET(Vals, record(Size),
         maps:from_list(lists:zip(Keys, Vals))).
```

Elixir code/ResponsibleTesting/elixir/bday/test/csv_test.exs

```
def csv_source() do
  let size <- pos_integer() do
    let keys <- header(size) do
      list(entry(size, keys))
    end
  end
end

def entry(size, keys) do
  let vals <- record(size) do
    Map.new(Enum.zip(keys, vals))
  end
end
```

The csv_source() generator picks up a Size value that represents how many entries will be in each row. By putting it in a ?LET macro, we make sure that whatever the expression that uses Size is, it uses a discrete value, and not the generator itself. This will allow us to use Size multiple times safely with always the same value in the second ?LET macro. That second macro generates one set of headers (the keys of every map), and then uses them to create a list of entries.

The entries themselves are specified by the entry/2 generator, which creates a list of record values, and pairs them up with the keys from csv_source() into a map. This generates values such as these:

```
$ rebar3 as test shell
«build output»
1> proper_gen:pick(prop_csv:csv_source()).
{ok,[#{[] => "z","&_f" => "t,:S","cH^*M" => "{6Z#"},
     #{[] => "kS3>","&_f" => "/","cH^*M" => "eK"},
     #{[] => "~","&_f" => [],"cH^*M" => "Bk#?X7h"}]}
2> proper_gen:pick(prop_csv:csv_source()).
{ok,[#{"D" => "\nQNUO","D4D" => "!E$0; )KL",
       "R\r~P{qC-" => "4L(Q4-N9","T6FAGuhf" => "wSP4jONE3Q"},
     #{"D" => "!Y7H\rQ?I7\r","D4D" => [],
       "R\r~P{qC-" => "}66W2I9+?R","T6FAGuhf" => "pF8/C"},
     #{"D" => [],"D4D" => "?'_6","R\r~P{qC-" => "j|Q",
       "T6FAGuhf" => "f$s7=sFx2>"},
     #{"D" => "e;ho1\njn!2","D4D" => ".8B{k|+|}",
       "R\r~P{qC-" => "V","T6FAGuhf" => "a\"/J\rfE#$"},
«more maps»
```

As you can see, all the maps for a given batch share the same keys, but have varying values. Those are ready to be encoded and passed to our property:

Erlang code/ResponsibleTesting/erlang/bday/test/prop_csv.erl

```
prop_roundtrip() ->
    ?FORALL(Maps, csv_source(),
            Maps =:= bday_csv:decode(bday_csv:encode(Maps))).
```

Elixir code/ResponsibleTesting/elixir/bday/test/csv_test.exs

```
property "roundtrip encoding/decoding" do
  forall maps <- csv_source() do
    maps == Csv.decode(Csv.encode(maps))
  end
end
```

Running it at this point would be an instant failure since we haven't written the code to go with it. Since this chapter is about tests far more than how to

implement a CSV parser, we'll go over the latter rather quickly. Here's an implementation that takes about a hundred lines:

Erlang code/ResponsibleTesting/erlang/bday/src/bday_csv.erl

```
-module(bday_csv).
-export([encode/1, decode/1]).

%% @doc Take a list of maps with the same keys and transform them
%% into a string that is valid CSV, with a header.
-spec encode([map()]) -> string().
encode([]) -> "";
encode(Maps) ->
    Keys = lists:join(",", [escape(Name) || Name <- maps:keys(hd(Maps))]),
    Vals = [lists:join(",", [escape(Field) || Field <- maps:values(Map)])
            || Map <- Maps],
    lists:flatten([Keys, "\r\n", lists:join("\r\n", Vals)]).

%% @doc Take a string that represents a valid CSV data dump
%% and turn it into a list of maps with the header entries as keys
-spec decode(string()) -> list(map()).
decode("") -> [];
decode(CSV) ->
    {Headers, Rest} = decode_header(CSV, []),
    Rows = decode_rows(Rest),
    [maps:from_list(lists:zip(Headers, Row)) || Row <- Rows].
```

First, there's the public interface with two functions: `encode/1` and `decode/1`. The functions are fairly straightforward, delegating the more complex operations to private helper functions. Let's start by looking at those helping with encoding:

```
%%%%%%%%%%%%%%%
%%% PRIVATE %%%
%%%%%%%%%%%%%%%

%% @private return a possibly escaped (if necessary) field or name
-spec escape(string()) -> string().
escape(Field) ->
    case escapable(Field) of
        true -> "\"" ++ do_escape(Field) ++ "\"";
        false -> Field
    end.

%% @private checks whether a string for a field or name needs escaping
-spec escapable(string()) -> boolean().
escapable(String) ->
    lists:any(fun(Char) -> lists:member(Char, [$",$,,$\r,$\n]) end, String).
```

```
%% @private replace escapable characters (only `"') in CSV.
%% The surrounding double-quotes are not added; caller must add them.
-spec do_escape(string()) -> string().
do_escape([]) -> [];
do_escape([$"|Str]) -> [$", $" | do_escape(Str)];
do_escape([Char|Rest]) -> [Char | do_escape(Rest)].
```

If a string is judged to need escaping (according to escapable/1), then the string is wrapped in double quotes (") and all double quotes inside of it are escaped with another double quote. With this, encoding is covered. Next there's decoding's private functions:

```
%% @private Decode the entire header line, returning all names in order
-spec decode_header(string(), [string()]) -> {[string()], string()}.
decode_header(String, Acc) ->
    case decode_name(String) of
        {ok, Name, Rest} -> decode_header(Rest, [Name | Acc]);
        {done, Name, Rest} -> {[Name | Acc], Rest}
    end.

%% @private Decode all rows into a list.
-spec decode_rows(string()) -> [[string()]].
decode_rows(String) ->
    case decode_row(String, []) of
        {Row, ""} -> [Row];
        {Row, Rest} -> [Row | decode_rows(Rest)]
    end.

%% @private Decode an entire row, with all values in order
-spec decode_row(string(), [string()]) -> {[string()], string()}.
decode_row(String, Acc) ->
    case decode_field(String) of
        {ok, Field, Rest} -> decode_row(Rest, [Field | Acc]);
        {done, Field, Rest} -> {[Field | Acc], Rest}
    end.

%% @private Decode a name; redirects to decoding quoted or unquoted text
-spec decode_name(string()) -> {ok|done, string(), string()}.
decode_name([$" | Rest]) -> decode_quoted(Rest);
decode_name(String) -> decode_unquoted(String).

%% @private Decode a field; redirects to decoding quoted or unquoted text
-spec decode_field(string()) -> {ok|done, string(), string()}.
decode_field([$" | Rest]) -> decode_quoted(Rest);
decode_field(String) -> decode_unquoted(String).
```

Decoding is done by fetching headers, then fetching all rows. A header line is parsed by reading each column name one at a time, and a row is parsed by reading each field one at a time. At the end you can see that both fields and names are actually implemented as quoted or unquoted strings:

```
%% @private Decode a quoted string
-spec decode_quoted(string()) -> {ok|done, string(), string()}.
decode_quoted(String) -> decode_quoted(String, []).

%% @private Decode a quoted string
-spec decode_quoted(string(), [char()]) -> {ok|done, string(), string()}.
decode_quoted([$"], Acc) -> {done, lists:reverse(Acc), ""};
decode_quoted([$",$\r,$\n | Rest], Acc) -> {done, lists:reverse(Acc), Rest};
decode_quoted([$",$, | Rest], Acc) -> {ok, lists:reverse(Acc), Rest};
decode_quoted([$",$" | Rest], Acc) -> decode_quoted(Rest, [$" | Acc]);
decode_quoted([Char | Rest], Acc) -> decode_quoted(Rest, [Char | Acc]).

%% @private Decode an unquoted string
-spec decode_unquoted(string()) -> {ok|done, string(), string()}.
decode_unquoted(String) -> decode_unquoted(String, []).

%% @private Decode an unquoted string
-spec decode_unquoted(string(), [char()]) -> {ok|done, string(), string()}.
decode_unquoted([], Acc) -> {done, lists:reverse(Acc), ""};
decode_unquoted([$\r,$\n | Rest], Acc) -> {done, lists:reverse(Acc), Rest};
decode_unquoted([$, | Rest], Acc) -> {ok, lists:reverse(Acc), Rest};
decode_unquoted([Char | Rest], Acc) -> decode_unquoted(Rest, [Char | Acc]).
```

Elixir translation on page ?.

Both functions to read quoted or unquoted strings mostly work the same, except quoted ones have specific rules about unescaping content baked in. And with this, our CSV handling is complete.

The code was developed against the properties by running the tests multiple times and refining the implementation iteratively. For brevity, we'll skip all the failed attempts that did some dirty odd parsing, except for one failing implementation that's particularly interesting since it had a failure against the following input:

```
\r\na
```

This is technically a valid CSV file with a single column, for which the empty name "" is chosen (commas only split values, so a single \r\n means a 0-length string as a value on that line), and with a single value "a". The expected output from decoding this is [#{"" ⇒ "a"}]. The first version of the parser had no way to cope with such cases, since I couldn't imagine them either. The parser shown previously is handling such cases, but the digging and rewriting has been skipped for brevity.

If you run the property over the previous (correct) implementation, you'll find it still fails on this tricky test:

```
bday_csv:encode([#{""=>""},#{""=>""}]) => "\r\n\r\n"
bday_csv:decode("\r\n\r\n")            => [#{"" => ""}]
```

This is an ambiguity embedded directly in the CSV specification. Because a trailing \r\n is acceptable, it is impossible to know whether there is an empty trailing line or not in the case of one-column data sets. Above one column, at least one comma (,) is going to be on the line. At one column, there is no way to know.

Under fifty lines of tests were enough to discover inconsistencies in RFC 4180 itself, inconsistencies that can't be reconciled or fixed in our program. Instead, we'll have to relax the property, making sure we don't cover that case by changing csv_source() and adding +1 to every Size value we generate. That way, we shift the range for columns from 1..N to 2..(N+1), ensuring we always have two or more columns in generated data.

Erlang code/ResponsibleTesting/erlang/bday/test/prop_csv.erl

```
csv_source() ->
    ?LET(Size, pos_integer(),
         ?LET(Keys, header(Size+1),
              list(entry(Size+1, Keys)))).
```

Elixir code/ResponsibleTesting/elixir/bday/test/csv_test.exs

```
def csv_source() do
  let size <- pos_integer() do
    let keys <- header(size + 1) do
      list(entry(size + 1, keys))
    end
  end
end
```

After this change, the property works fine. For good measure, we should add a unit test representing the known unavoidable bug to the same test suite, documenting known behavior:

Erlang code/ResponsibleTesting/erlang/bday/test/prop_csv.erl

```
-module(prop_csv).
-include_lib("proper/include/proper.hrl").
-include_lib("eunit/include/eunit.hrl").
-compile(export_all).

«existing code»

%%%%%%%%%%%%%
%%% EUnit %%%
%%%%%%%%%%%%%

%% @doc One-column CSV files are inherently ambiguous due to
%% trailing CRLF in RFC 4180. This bug is expected
```

```
one_column_bug_test() ->
    ?assertEqual("\r\n\r\n", bday_csv:encode([#{""=>""},#{""=>""}])),
    ?assertEqual([#{"" => ""}], bday_csv:decode("\r\n\r\n")).
```

Elixir code/ResponsibleTesting/elixir/bday/test/csv_test.exs

```
## Unit Tests ##
test "one column CSV files are inherently ambiguous" do
  assert "\r\n\r\n" == Csv.encode([%{"" => ""}, %{"" => ""}])
  assert [%{"" => ""}] == Csv.decode("\r\n\r\n")
end
```

The Erlang suite can be run with rebar3 eunit as well as rebar3 proper. Using prop_ as a prefix to both the module and properties lets the proper plugin detect what it needs. For EUnit, the _test suffix for functions lets it do the proper detection. If you also wanted to use the common test framework in Erlang, the _SUITE suffix should be added to the module.

There is a last gotcha implicit to the implementation of our CSV parser: since it uses maps, duplicate column names are not tolerated. Since our CSV files have to be used to represent a database, it is probably a fine assumption to make about the data set that column names are all unique. All in all, we're probably good ignoring duplicate columns and single-column CSV files since it's unlikely database tables would be that way either, but it's not fully CSV-compliant. This gotcha was discovered by adding good old samples from the RFC into the EUnit test suite:

Erlang code/ResponsibleTesting/erlang/bday/test/prop_csv.erl

```
rfc_record_per_line_test() ->
    ?assertEqual([#{"aaa" => "zzz", "bbb" => "yyy", "ccc" => "xxx"}],
                 bday_csv:decode("aaa,bbb,ccc\r\nzzz,yyy,xxx\r\n")).

rfc_optional_trailing_crlf_test() ->
    ?assertEqual([#{"aaa" => "zzz", "bbb" => "yyy", "ccc" => "xxx"}],
                 bday_csv:decode("aaa,bbb,ccc\r\nzzz,yyy,xxx")).

rfc_double_quote_test() ->
    ?assertEqual([#{"aaa" => "zzz", "bbb" => "yyy", "ccc" => "xxx"}],
                 bday_csv:decode("\"aaa\",\"bbb\",\"ccc\"\r\nzzz,yyy,xxx")).

rfc_crlf_escape_test() ->
    ?assertEqual([#{"aaa" => "zzz", "b\r\nbb" => "yyy", "ccc" => "xxx"}],
                 bday_csv:decode("\"aaa\",\"b\r\nbb\",\"ccc\"\r\nzzz,yyy,xxx")).

rfc_double_quote_escape_test() ->
    %% Since we decided headers are mandatory, this test adds a line
    %% with empty values (CLRF,,) to the example from the RFC.
    ?assertEqual([#{"aaa" => "", "b\"bb" => "", "ccc" => ""}],
                 bday_csv:decode("\"aaa\",\"b\"\"bb\",\"ccc\"\r\n,,")).
```

```
%% @doc this counterexample is taken literally from the RFC and cannot
%% work with the current implementation because maps have no dupe keys
dupe_keys_unsupported_test() ->
    CSV = "field_name,field_name,field_name\r\n"
          "aaa,bbb,ccc\r\n"
          "zzz,yyy,xxx\r\n",
    [Map1,Map2] = bday_csv:decode(CSV),
    ?assertEqual(1, length(maps:keys(Map1))),
    ?assertEqual(1, length(maps:keys(Map2))),
    ?assertMatch(#{"field_name" := _}, Map1),
    ?assertMatch(#{"field_name" := _}, Map2).
```

Elixir translation on page ?.

The last test was impossible to cover with the current property implementation, so doing it by hand in an example case still proved worthwhile. In the end, ignoring comments and blank lines, twenty-seven lines of example tests let us find one gotcha about our code and validate specific cases against the RFC, and nineteen lines of property-based tests that let us exercise our code to the point we found inconsistencies in the RFC itself (which is not too representative of the real world).[6] That's impressive.

All in all, this combination of example-based unit tests and properties is a good match. The properties can find very obtuse problems that require complex searches into the problem space, both in breadth and in depth. On the other hand, they can be written in a way that they're general enough that some basic details could be overlooked. In this case, the property exercised encoding and decoding exceedingly well, but didn't do it infallibly—we programmers are good at making mistakes no matter the situation, and example tests could also catch some things. They're great when acting as *anchors*, an additional safety net making sure our properties are not drifting away on their own.

Another similar good use of unit tests are to store *regressions*, specific tricky bugs that need to be explicitly called out or validated frequently. PropEr with Erlang and Elixir both contain options to store and track regressions automatically if you want them to. Otherwise, example tests are as good of a place as any to store that information.

With the CSV handling in place, we can now focus on filtering employee records.

6. http://tburette.github.io/blog/2014/05/25/so-you-want-to-write-your-own-CSV-code/