# Property-based testing in Rust with Proptest

Posted on June 19, 2020

Table of Contents:



Note: I originally wrote this article for LogRocket. You can find the original here.

Software testing is an industry standard practice, but testing methodologies and techniques vary dramatically in their practicality and effectiveness. Today you'll learn about property-based testing (PBT) including how it works, when it makes sense, and how to do it in Rust.

## What is property-based testing?

To illustrate how PBT works, let's first look at a basic unit test. Let's say that you've written a function `maybe_works` that you want to compare against a function that you know works properly, say `definitely_works`. A unit test comparing these two functions for some input `input` would look like the test below.

```
fn test_maybe_works() {
    let input = ...;
    assert_eq!(maybe_works(input), definitely_works(input));
}
```

That test was very easy to write, but there are some issues to be aware of. One issue is that you need to know which exact inputs to use in your tests that will alert you to the presence of bugs. Another issue is that this becomes tedious when you want to test `maybe_works` against `definitely_works` for a variety of inputs.

A more efficient way to test `maybe_works` against `definitely_works` would be to run the test several times with a variety of randomly generated inputs. With these randomly generated inputs you don't know the precise value being supplied to `maybe_works` or `definitely_works`, but you can often make a general statement such as "the outputs of `maybe_works` and `definitely_works` should be the same given the same input." In practice the "randomly generated inputs" are rarely truly random. You typically constrain the inputs somehow so that you can target a certain piece of code and avoid false positives or negatives.

It's common, however, not to have a reference implementation such as `definitely_works`. In this case you need to think harder and more abstractly about the properties of your code (hence property-based testing) and how you can verify them. Yet another way to use PBT is as a guide in the design of a piece of code. In this article we're going to use PBT to help us write a parser.

## Getting started

Create a new `cargo` project called `sentence-parser`.

```
$ cargo new --lib sentence-parser
```

You'll use the `pest` crate to write the parser, so add `pest` to your `Cargo.toml`.

```
[dependencies]
pest = "~2.1"
pest_derive = "~2.1"
```

The `pest` crate generates a parser from a user-defined grammar file, so create a file called `sentence.pest` and put it in the `src` directory. Paste the following contents into the file.

```
word = { ASCII_ALPHA+ }
WHITESPACE = _{" "}
```

The `pest` crate is not the focus of this tutorial, but it's helpful to have a cursory understanding of this grammar file. Each rule in the file tells `pest` how to parse a certain type of input. You can use rules within the definition of other rules to eventually build a parser that understands complex input. For a detailed overview of the syntax, see the `pest` syntax guide.

For simplicity's sake we're going to significantly relax the definitions of "word" and "sentence." We'll define a "word" as "any sequence of one or more ASCII alphabetical characters." This means that `flkjasdfAQTCcs` is a valid word for our purposes.

To create a parser from this file add the following snippet to `lib.rs`.

```
#[macro_use] extern crate pest_derive;
use pest::Parser;

#[derive(Parser)]
#[grammar = "sentence.pest"]
pub struct SentenceParser;
```

The `#[derive(Parser)]` and `#[grammar = "``sentence.pest``"]` attributes will read the grammar file and generate a parser based on that grammar. This will also generate an enum called `Rule` which has a variant for each of the rules in the grammar file. When you want to parse the contents of a string, you pass one of the `Rule` variants to `SentenceParser::parse` along with the string. We will use this feature to test individual rules in the grammar rather than testing the whole thing at once.

Finally, add `proptest` to your `Cargo.toml`.

```
[dev-dependencies]
proptest = "~0.9"
```

## Using `proptest`

Doing PBT with `proptest` can look mostly the same as writing normal tests. There are two major differences which I'll illustrate with the snippet below.

```
#[cfg(test)]
mod tests {
    use super::*;

    #[test]
    fn a_normal_test() {
        ...
        assert!(...)
    }

    proptest!{
        #[test]
        fn a_property_based_test(foo in some_strategy()) {
            ...
            prop_assert!(...)
        }
    }
}
```

The first thing to note is that you can mix PBT with normal unit tests; PBT isn't all or nothing. Next, note that your property-based tests must be wrapped in the `proptest!` macro. Another point of note is that `prop_assert!` is used in place of `assert!`. The `proptest` test runner will attempt to find a minimal failing input by causing the test to fail with simpler and simpler inputs. Using the `prop_assert!` and `prop_assert_eq!` macros will cause the test failure message (panic message) to be printed only for the minimal failing input rather than multiple times for a single

test. Finally, notice that the function arguments to the property-based test are of the form `argname in strategy`.

Conceptually, a “strategy” is a way of generating instances of a type, possibly with some constraints. More concretely, a strategy is a type that implements the `Strategy` trait. In addition, implementing the `Arbitrary` trait defines the default or main strategy for a type. This `Arbitrary` implementation will typically be the strategy that generates the widest range of values for the type. The `any<T>()` function provided by `proptest` uses this `Arbitrary` implementation to generate truly arbitrary instances of the type `T`. If you would like to generate instances that are subject to some constraints, you need to define functions that return `impl Strategy`.

The `proptest` crate provides implementations of `Arbitrary` for a wide variety of Rust primitives and collections. Implementing `Arbitrary` for your own type is straightforward. The example below is mostly boilerplate.

```
impl Arbitrary for T {
    type Parameters = ();
    type Strategy = BoxedStrategy<Self>;

    fn arbitrary_with(_: Self::Parameters) -> Self::Strategy {
        ...
    }
}
```

The only piece that really requires thought on your part is the `…`, which defines how to generate the values. You may parameterize how the strategy generates values using the `Parameters` type definition. In most cases this isn’t necessary, so define it as `()`. The concrete type of the strategy returned by `arbitrary_with` is defined by the `Strategy` type definition. You may declare a specific type, or you may choose the simplest route and return a trait object via the `BoxedStrategy` type.

The strategy defined by the `Arbitrary` trait may generate values that are too general for some scenario, or you may simply want to define more than one strategy for a given type. In either case you will want to write a function that returns a strategy.

```
fn my_strategy() -> impl Strategy<Value = T> {
```

```
    ...
}
```

Both methods above share some common patterns represented by … . The Transforming Strategies section of the `proptest` book illustrates how this is done in great detail, but we'll discuss it briefly since it's such a common practice when using `proptest`.

## Strategies

The `proptest` crate comes with several built-in strategies, namely `any<T>`, the `proptest::collection` module, `proptest::string::string_regex`, and a variety of others. In addition, tuples and arrays of strategies are themselves strategies. This means that you can generate a tuple of values where each value is drawn from a different strategy, then apply methods from the `Strategy` trait and turn the tuple's elements into another type.

### `Strategy::prop_map`

The `prop_map` method allows you to transform the values generated by a strategy and transform them into a new type. This is one of the primary methods you'll use to build up a custom type from primitives or other types. The example below illustrates how you can generate a struct by transforming a tuple of primitive types.

```
struct Point {
    x: f32,
    y: f32
}

fn point_strat() -> impl Strategy<Value = Point> {
    (any<f32>::(), any<f32>::()).prop_map(|(x, y)| {
        Point {
            x: x,
            y: y
        }
    })
}
```

In short, a strategy produces a random value and `prop_map` uses that random value to compute a different value.

### `Strategy::prop_filter`

The `prop_filter` methods lets you constrain the values produced by one strategy by filtering with a predicate. The function below returns a strategy that generates `u8` s that are greater than 100.

```
fn u8_greater_than_100() -> impl Strategy<Value = u8> {
    any::<u8>().prop_filter("reason for filtering", |x| x > &100u8)
}
```

### `Strategy::prop_flat_map`

Sometimes you want to generate values that depend on each other somehow. This is a scenario in which you'll want to reach for the `prop_flat_map` method. When called on a strategy, the `prop_flat_map` method takes a random value from that strategy and uses it to produce another strategy. Note the difference between `prop_map` and `prop_flat_map` . With `prop_map` a value from a strategy is used to produce a new value, whereas with `prop_flat_map` a value from a strategy is used to produce a new strategy. The canonical example use case from the `proptest` documentation is a strategy that produces a `Vec` and a random valid index into that `Vec` .

```
fn vec_and_index() -> impl Strategy<Value = (Vec<String>, usize)> {
    prop::collection::vec(".*", 1..100)
        .prop_flat_map(|vec| {
            let len = vec.len();
            (Just(vec), 0..len)
        })
}
```

### `Just`

This type creates a strategy that always returns the same value. One use case is shown in the `prop_flat_map` example above. Another use case is with the `prop_oneof!` macro.

### `prop_oneof!`

This macro creates a strategy that produces values from a list of strategies. The most common use case is generating enum variants.

```
enum MyEnum {
    Foo,
    Bar,
    Baz,
}

fn myenum_strategy() -> impl Strategy<Value = MyEnum> {
    prop_oneof![
        Just(MyEnum::Foo),
        Just(MyEnum::Bar),
        Just(MyEnum::Baz),
    ]
}
```

Note that you must supply a list of strategies, not a list of values. This is why you must wrap the enum variants in `Just`.

## Writing the parser

Now that you know some common `proptest` patterns you can get down to the business of writing and testing a parser.

### Word

Create a `proptest!` block in the `test` module. First we'll create a function called `valid_word` that returns a strategy that produces, you guessed it, a valid word.

```
#[cfg(test)]
mod test {
    use super::*;

    proptest!{
        fn valid_word() -> impl Strategy<Value = String> {
```

```
            proptest::string::string_regex("[a-zA-Z]+").unwrap()
        }

        #[test]
        fn parses_valid_word(w in valid_word()) {
            let parsed = SentenceParser::parse(Rule::word, w.as_str(
            prop_assert!(parsed.is_ok());
        }
    }
}
```

The `proptest::string::string_regex` function returns a strategy that produces strings that match the provided regular expression. In this case that's a sequence of one or more ASCII letters. You could have used the shorthand `w in "``[a-zA-Z]+``"` to accomplish the same thing, but I find that writing the strategy as a function allows you to give a descriptive name to the regular expression. Note that in `_ in valid_word()` the `_` defines the name of the value produced by the strategy so that you can use it in the body of your test.

The test attempts to parse the generated word using the rule `Rule::word`. If you run `cargo test` you should see that your test passes (along with other compilation statements).

Tests that pass when fed valid inputs are important, but tests that detect errors when your code is fed invalid inputs are equally important. This time you'll create a test that feeds non-letter characters to the parser as well as strings of length zero. Create another test that looks like this.

```
fn invalid_word() -> impl Strategy<Value = String> {
    proptest::string::string_regex("[^a-zA-Z]*").unwrap()
}

#[test]
fn rejects_invalid_word(w in invalid_word()) {
    let parsed = SentenceParser::parse(Rule::word, w.as_str());
    prop_assert!(parsed.is_err());
}
```

Run the test suite again and see that the tests pass.

## Words

Now create a `words` rule that looks like this.

```
words = { word+ }
```

This rule will match sequences of `word` s joined by spaces. We'll follow the same pattern as before to test whether we've created a good rule. First create a test that attempts to parse valid input. Note that this time we'll use `Rule::words` instead of `Rule::word` .

```
fn words() -> impl Strategy<Value = String> {
    proptest::string::string_regex("[a-z]+( [a-z]+)*").unwrap()
}

#[test]
fn parses_valid_words(ws in words()) {
    let parsed = SentenceParser::parse(Rule::words, ws.as_str());
    prop_assert!(parsed.is_ok());
}
```

Run the test to make sure everything works.

Next we'll create a test that attempts to parse invalid input, but first we need to decide what that invalid input should look like. We already know that we can parse a `word` properly, so let's test whether the `words` rule can parse zero `word` s and whether it can parse `word` s separated by characters other than spaces. You don't need a property-based test for the empty string, so go ahead and create the following test inside the `test` module, but outside the `proptest!` block.

```
#[test]
fn words_rejects_empty_string() {
    let parsed = SentenceParser::parse(Rule::words, "");
    assert!(parsed.is_err());
}
```

What's left is to test what happens when we have characters other than just a space between `word`s such as `\t`, `\r`, and `\n`. Unfortunately, supplying an input string like this and expecting parsing to fail doesn't work. Consider the string `"``a\tb``"`. The issue is that `\t` isn't a valid character that can appear in a `word`, so the parser stops consuming characters at `\t`, having only consumed the `a`. However, `a` matches the `word`, which matches `words` ("at least one `word`"). So, our string that contains invalid word separators still successfully matches the `words` rule because `words` is only matching part of the input. We can rectify this by changing our rule to parse until the end of the input rather than finishing on some minimal prefix of the input that parses successfully.

```
words = { word+ ~ EOI }
```

The `EOI` rule is provided by `pest` and represents "end of input." This solves our problem for now, but it will break things later when we try to parse entire sentences. For now we'll leave the `words` rule as is and forego feeding it this kind of invalid input.

## Enclosed

Now we're going to build on the `words` rule by creating a rule, `enclosed`, that simply wraps a `words` in some kind of delimiter such as commas or parentheses.

```
enclosed = ${
    "(" ~ words ~ ")" |
    ", " ~ words ~ ","
}
```

Note that this rule begins with `$`, which indicates that we don't want `~` to implicitly gobble up whitespace. We want this because `(foo bar)` is valid, but `( foo bar )` is not.

To test this rule we're going generate more complex inputs from strategies that we'll define. First, we'll create a strategy that generates a string that can be parsed by the `enclosed` rule. To do this we need a way of generating matching delimiters. In your `tests` module, create the following type.

```
#[derive(Debug, PartialEq, Clone)]
enum EnclosedDelimiter {
    OpenParen,
    CloseParen,
    OpenComma,
    CloseComma,
}
```

Next we'll create a method that converts the various enum variants into `&str`.

```
impl EnclosedDelimiter {
    fn to_str(&self) -> &str {
        match self {
            EnclosedDelimiter::OpenParen => "(",
            EnclosedDelimiter::CloseParen => ")",
            EnclosedDelimiter::OpenComma => ", ",
            EnclosedDelimiter::CloseComma => ","
        }
    }
}
```

Now create a function called `opening_delimiter`. This function will only generate valid opening delimiters.

```
fn opening_delimiter() -> impl Strategy<Value = EnclosedDelimiter> {
    prop_oneof![
        Just(EnclosedDelimiter::OpenParen),
        Just(EnclosedDelimiter::OpenComma)
    ].boxed()
}
```

Now create a function called `valid_enclosed`. This function will return a strategy that only generates strings that the `enclosed` rule can successfully parse by matching the randomly chosen valid opening delimiter with its matching closing delimiter. This is an example of when the `prop_flat_map` method comes in handy.

```
fn valid_enclosed() -> impl Strategy<Value = String> {
    let words_strat = proptest::string::string_regex("[a-z]+( [a-z]+
    (opening_delimiter(), words_strat).prop_flat_map(|(open, words)|
        let close = match open {
            EnclosedDelimiter::OpenParen => Just(EnclosedDelimiter::(
            EnclosedDelimiter::OpenComma => Just(EnclosedDelimiter::(
            _ => unreachable!()
        };
        (Just(open), close, Just(words))
    }).prop_map(|(open, close, words)| {
        let mut enclosed = String::new();
        enclosed.push_str(open.to_str());
        enclosed.push_str(words.as_str());
        enclosed.push_str(close.to_str());
        enclosed
    })
}
```

This strategy begins with an opening delimiter and a valid sequence of words, then uses `prop_flat_map` to select the valid closing delimiter. Once the delimiters and words have been generated they are combined into a `String`. The `_ => unreachable!()` line is included because `match` must be exhaustive and we know that `opening_delimiter` can't generate either of the closing delimiters.

Now write the `parses_valid_enclosed` test and make sure it works.

```
#[test]
fn parses_valid_enclosed(enc in valid_enclosed()) {
    let parsed = SentenceParser::parse(Rule::enclosed, enc.as_str())
    prop_assert!(parsed.is_ok());
}
```

Uh oh, this test fails! The minimal failing input in this case is `(a a)`. The reason may not be obvious unless you're familiar with `pest`, but it has to do with how whitespace is handled. Remember that the `enclosed` rule began with `$`. A rule that starts with `@` or `$` is called "atomic" and will not implicitly consume whitespace between tokens, and this property cascades to sub-rules within the current rule. In

this case, `enclosed` references the `words` rule, which changes how `words` matches whitespace in the context of the `enclosed` rule. All we need to do is put a `!` in front of the `words` rule to prevent this cascade. The new rules should look like this.

```
word = { ASCII_ALPHA+ }
words = !{ word+ }
enclosed = ${
    "(" ~ words ~ ")" |
    ", " ~ words ~ ","
}
WHITESPACE = _{" "}
```

The tests should all pass now.

The next task is to generate invalid strings for the `enclosed` rule. Now the opening delimiter may be any of the delimiters, and the closing delimiter will be explicitly chosen not to match the opening delimiter.

First we'll implement the `Arbitrary` trait so that we can generate randomly selected instances of `EnclosedDelimiter`.

```
impl Arbitrary for EnclosedDelimiter {
    type Parameters = ();
    type Strategy = BoxedStrategy<Self>;
    fn arbitrary_with(_: Self::Parameters) -> Self::Strategy {
        prop_oneof![
            Just(EnclosedDelimiter::OpenParen),
            Just(EnclosedDelimiter::CloseParen),
            Just(EnclosedDelimiter::OpenComma),
            Just(EnclosedDelimiter::CloseComma),
        ].boxed()
    }
}
```

Now create the function that intentionally mismatches the delimiters.

```
fn invalid_enclosed() -> impl Strategy<Value = String> {
    let words_strat = proptest::string::string_regex("[a-z]+( [a-z]+
    (any::<EnclosedDelimiter>(), words_strat).prop_flat_map(|(open, 
        let close = match open {
            EnclosedDelimiter::OpenParen => prop_oneof![
                Just(EnclosedDelimiter::OpenParen),
                Just(EnclosedDelimiter::OpenComma),
                Just(EnclosedDelimiter::CloseComma)
            ].boxed(),
            EnclosedDelimiter::CloseParen => any::<EnclosedDelimiter
            EnclosedDelimiter::OpenComma => prop_oneof![
                Just(EnclosedDelimiter::OpenParen),
                Just(EnclosedDelimiter::CloseParen),
                // Just(EnclosedDelimiter::OpenComma),
            ].boxed(),
            EnclosedDelimiter::CloseComma => any::<EnclosedDelimiter
        };
        (Just(open), close, Just(words))
    }).prop_map(|(open, close, words)| {
        let mut enc = String::new();
        enc.push_str(open.to_str());
        enc.push_str(words.as_str());
        enc.push_str(close.to_str());
        enc
    })
}
```

You'll notice that one of the lines is commented out. If you uncomment this line you'll see that the test fails because it successfully parses what is supposed to be invalid input. The string representation of the `OpenComma` variant ends with a space, so the parser sees `foo,` as `foo`, followed by `CloseComma`, followed by ``. This means that an `OpenComma` / `OpenComma` pair can accidentally be parsed as `OpenComma` / `CloseComma` and no error will be generated. This error will be addressed by a later rule so we'll skip testing it in this rule. This is what the test looks like.

```
#[test]
```

```
fn rejects_mismatched_enclosed_delimiters(enc in invalid_enclosed())
    let parsed = SentenceParser::parse(Rule::enclosed, enc.as_str())
    prop_assert!(parsed.is_err());
}
```

We can make another test that simply leaves off the closing delimiter.

```
fn missing_closing_delimiter() -> impl Strategy<Value = String> {
    (any::<EnclosedDelimiter>(), words()).prop_map(|(open, ws)| {
        let mut enc = String::new();
        enc.push_str(open.to_str());
        enc.push_str(ws.as_str());
        enc
    })
}

#[test]
fn rejects_missing_closing_delimiter(enc in missing_closing_delimite
    let parsed = SentenceParser::parse(Rule::enclosed, enc.as_str())
    prop_assert!(parsed.is_err());
}
```

Run the tests again and verify that they all still pass.

## Chunk

The `words` and `enclosed` rules are similar in the sense that they both represent sequences of words. To encode this similarity we'll make a `chunk` rule that matches either `words` or `enclosed` so that we can represent a sequence of words, some of which may be enclosed in delimiters, as a sequence of `chunk`s.

```
chunk = ${ words | enclosed }
```

Fortunately we've already written strategies for `words` and `enclosed` so we can skip straight to writing tests for this rule.

```
#[test]
```

```
fn parses_valid_words_chunk(ws in words()) {
    let parsed = SentenceParser::parse(Rule::chunk, ws.as_str());
    prop_assert!(parsed.is_ok());
}

#[test]
fn parses_valid_enclosed_chunk(enc in valid_enclosed()) {
    let parsed = SentenceParser::parse(Rule::chunk, enc.as_str());
    prop_assert!(parsed.is_ok());
}

#[test]
fn rejects_mismatched_delim_chunk(enc in invalid_enclosed()) {
    let parsed = SentenceParser::parse(Rule::chunk, enc.as_str());
    prop_assert!(parsed.is_err());
}

#[test]
fn rejects_missing_delim_chunk(enc in missing_closing_delimiter()) {
    let parsed = SentenceParser::parse(Rule::chunk, enc.as_str());
    prop_assert!(parsed.is_err());
}
```

Run the tests to make sure they all pass.

## Punctuation

Sentences end with punctuation, so next we'll create a rule that matches punctuation.

```
punctuation = { "." | "!" | "?" }
```

This is a relatively simple rule, so the strategies are also simple. A valid punctuation character is either `.`, `!`, or `?`, and anything else is an invalid punctuation character.

```
fn punctuation() -> impl Strategy<Value = String> {
    prop_oneof![
        Just(String::from(".")),
```

```
            Just(String::from("?")),
            Just(String::from("!")),
        ]
    }

    fn invalid_punctuation() -> impl Strategy<Value = String> {
        proptest::string::string_regex("[^\\.\\?!]").unwrap()
    }

    #[test]
    fn parses_valid_punctuation(s in punctuation()) {
        let parsed = SentenceParser::parse(Rule::punctuation, s.as_str()
        prop_assert!(parsed.is_ok());
    }

    #[test]
    fn rejects_invalid_punctuation(s in invalid_punctuation()) {
        let parsed = SentenceParser::parse(Rule::punctuation, s.as_str()
        prop_assert!(parsed.is_err());
    }
```

These tests should pass without any trouble.

## Sentence

We're almost done! It's finally time to put it all together into a `sentence` rule.

```
sentence = ${ SOI ~ chunk+ ~ punctuation ~ EOI }
```

The `SOI` and `EOI` in this rule make sure that the entire input is parsed so that inputs like `a b c.def` aren't valid sentences.

We'll begin with the most basic type of valid sentence: one or more `chunk`s followed by a valid `punctuation`. First let's write a strategy that produces a sequence of `chunk`s joined by spaces.

```
    fn chunks() -> impl Strategy<Value = String> {
        proptest::collection::vec(chunk(), 1..10).prop_map(|cs| {
```

```
        cs.join(" ")
    })
}
```

Now use this `chunks` strategy to make a `valid_sentence` strategy.

```
fn valid_sentence() -> impl Strategy<Value = String> {
    (chunks(), punctuation()).prop_map(|(cs, p)| {
        let mut sentence = String::new();
        sentence.push_str(cs.as_str());
        sentence.push_str(p.as_str());
        sentence
    })
}
```

Now write a test that attempts to parse a valid sentence.

```
#[test]
fn parses_valid_sentence(s in valid_sentence()) {
    let parsed = SentenceParser::parse(Rule::sentence, s.as_str());
    prop_assert!(parsed.is_ok());
}
```

Run the test to make sure it…wait a minute, this test fails! Notice that the minimal failing input, `"(a) a."`, is a valid sentence (by our definition). This means that we must have misinformed the parser how to parse a sentence. After a little bit of investigation you'll remember that our `sentence` rule is atomic (it starts with `$`).

```
sentence = ${ SOI ~ chunk+ ~ punctuation ~ EOI }
```

As a reminder, this means that whitespace between rules is not consumed. When our parser tries to parse `"(a) a."` what it really sees is `<chunk><whitespace><chunk><punctuation>`. That `<whitespace>` doesn't match the `chunk+` piece of the `sentence` rule, so parsing fails. To fix this we'll need to handle whitespace manually where there's repetition.

```
words = ${ word ~ (" " ~ word)* }
sentence = ${ SOI ~ chunk ~ (" " ~ chunk)* ~ punctuation ~ EOI }
```

Now the test should pass.

There's a couple of other tests we can write, all of which should still pass with the current rules.

```
#[test]
fn rejects_missing_punctuation(s in chunks()) {
    let parsed = SentenceParser::parse(Rule::sentence, s.as_str());
    prop_assert!(parsed.is_err());
}

#[test]
fn rejects_trailing_characters(s in valid_sentence(), t in "[\\sa-zA
    let input = [s.as_str(), t.as_str()].join(" ");
    let parsed = SentenceParser::parse(Rule::sentence, input.as_str(
    prop_assert!(parsed.is_err());
}

#[test]
fn rejects_missing_space_between_chunks(
    w in words(),
    enc in valid_enclosed(),
    p in punctuation()
) {
    let input = [w, enc, p].join("");
    let parsed = SentenceParser::parse(Rule::sentence, input.as_str(
    prop_assert!(parsed.is_err());
}
```

## Conclusion

At this point you should have a passing understanding of how to write property-based tests. PBT isn't always the answer, but the simple act of thinking about the abstract

properties of your code can help you better understand it. The code used in this tutorial can be found on GitHub.

---

P.S. - You can follow me on BlueSky at @z-mitchell.bsky.social for Rust, Nix, and lukewarm takes.

P.P.S. - If you notice that something could be more accessible, please reach out and I'll do my best to fix it!