RUST WROCŁAW
MEETUP #13

# SHORT STORIES ABOUT PROCEDURAL MACROS

# ABOUT ME

They call me Wojtek

11 months of professional experience with Rust in ANIXE

BSc Thesis project in Rust (compiler yay!)

Maintainer of test-case crate
github.com/frondeus/test-case

@frondeus on twitter

github.com/frondeus

# TWO STORIES

## ABOUT PROCEDURAL MACROS

- Test cases - how to generate functions based on attributes
- Derive Validate trait - how to validate input (json request).

STORY 1

# TEST CASES

```
#[test_case(None,    None    => 0 :: "treats none as 0")]
#[test_case(Some(2), Some(3) => 5)]
#[test_case(Some(2 + 3), Some(4) => 2 + 3 + 4)]
fn fancy_addition(x: Option<i8>, y: Option<i8>) -> i8 {
    x.unwrap_or(0) + y.unwrap_or(0)
}

#[test_case( 2,  4 :: "when both operands are possitive")]
#[test_case( 4,  2 :: "when operands are swapped")]
#[test_case(-2, -4 :: "when both operands are negative")]
fn multiplication_tests(x: i8, y: i8) {
    let actual = x * y;
```

# TEST CASES

## JUST LIKE IN N-UNIT OR J-UNIT

```
using NUnit.Framework;

namespace Foo.Tests
{
    [TestFixture]
    public class FooTests
    {
        [TestCase(50)]
        [TestCase(10)]
        [TestCase(-10)]
        public void When_Foo_Has_Bar(int foo)
        {
            Assert.That(foo == 10);
        }
    }
}
```

- One test function
- Many calls with different values in arguments
- Possibility to assert returning value of the function

# TEST CASES

## IN RUST

```rust
#[cfg(test)]
mod tests {
  fn when_foo_has_bar(foo: i32) {
    assert_eq!(10, foo);
  }

  #[test]
  fn tests() {
    when_foo_has_bar(50);
    when_foo_has_bar(10);
    when_foo_has_bar(-10);
  }
}
```

- Not quite yet
- Only one output in test runner
- All test cases are running sequentially (slower)
- When first test case fails, you don’t know if second and third are passing
- Don’t know which case failed

# TEST CASES

## IN RUST

```
running 1 test
test tests::tests ... FAILED

failures:

---- tests::tests stdout ----
thread 'tests::tests' panicked at 'assertion failed: `(left == right)`
  left: `10`,
 right: `50`', src/lib.rs:4:4
note: run with `RUST_BACKTRACE=1` environment variable to display a backtrace.


failures:
    tests::tests

test result: FAILED. 0 passed; 1 failed; 0 ignored; 0 measured; 0 filtered out

error: test failed, to rerun pass '--lib'
```

# TEST CASES

## IN RUST

```rust
#[cfg(test)]
mod tests {
  fn when_foo_has_bar(foo: i32) {
    assert_eq!(10, foo);
  }

  #[test]
  fn when_foo_has_bar_50() {
    when_foo_has_bar(50);
  }

  #[test]
  fn when_foo_has_bar_10() {
    when_foo_has_bar(10);
  }

  #[test]
  fn when_foo_has_bar_minus_10() {
    when_foo_has_bar(-10);
  }
}
```

- Better but still clumsy
- Parallel
- Faster
- You know which test case fails
- But a lot of boilerplate

# TEST CASES

## IN RUST

```
running 3 tests
test tests::when_foo_has_bar_10 ... ok
test tests::when_foo_has_bar_50 ... FAILED
test tests::when_foo_has_bar_minus_10 ... FAILED

failures:

---- tests::when_foo_has_bar_50 stdout ----
thread 'tests::when_foo_has_bar_50' panicked at 'assertion failed: `(left == right)`
  left: `10`,
 right: `50`', src/lib.rs:4:4
note: run with `RUST_BACKTRACE=1` environment variable to display a backtrace.

---- tests::when_foo_has_bar_minus_10 stdout ----
thread 'tests::when_foo_has_bar_minus_10' panicked at 'assertion failed: `(left == right)`
  left: `10`,
 right: `-10`', src/lib.rs:4:4


failures:
    tests::when_foo_has_bar_50
    tests::when_foo_has_bar_minus_10

test result: FAILED. 1 passed; 2 failed; 0 ignored; 0 measured; 0 filtered out
```

# LET'S USE MACRO TO REDUCE BOILERPLATE!

# TEST_CASE MACRO

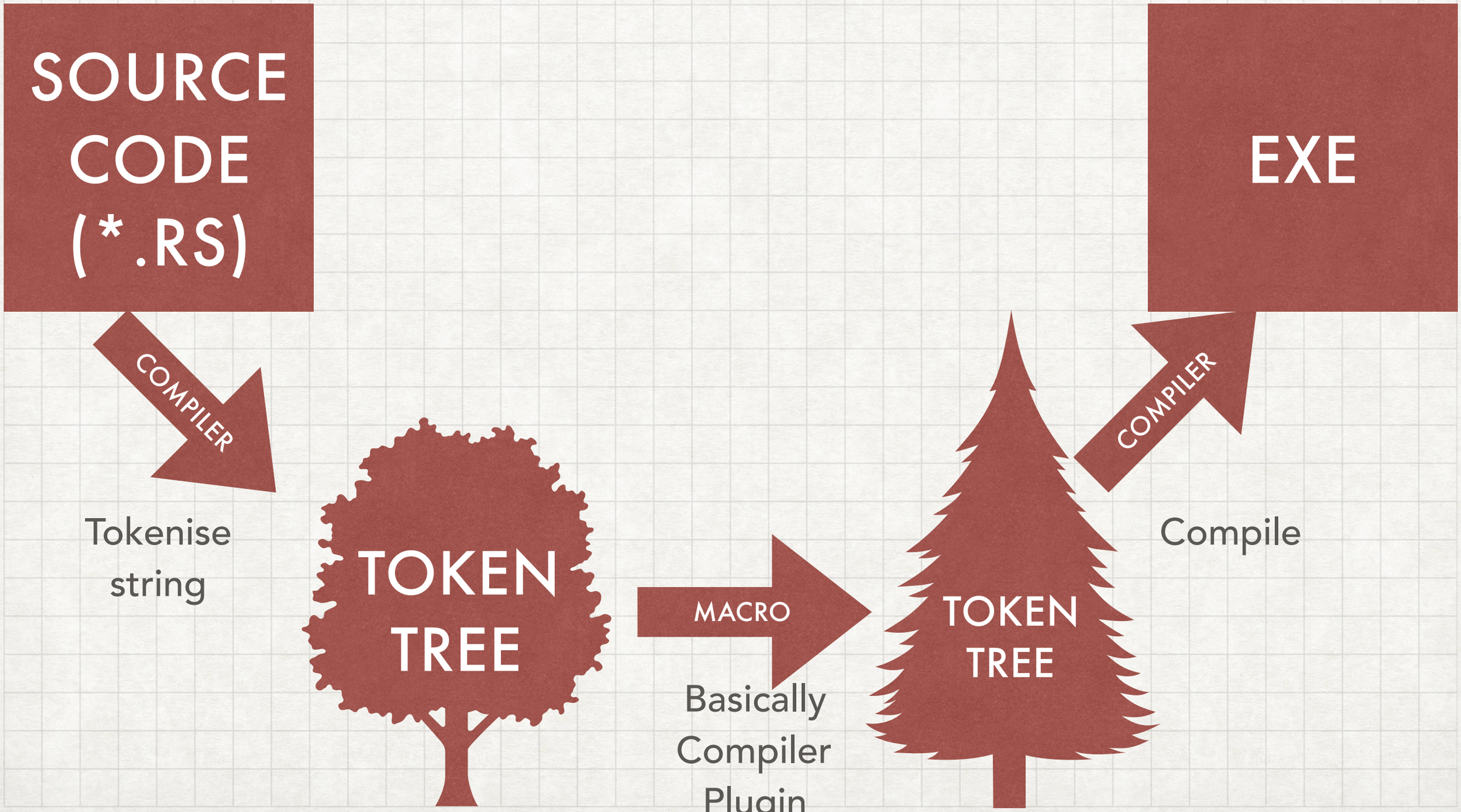
## MOCKUP WHAT WE WANT

```rust
#[cfg(test)]
mod tests {
    use test_case::test_case;

    #[test_case(50)]
    #[test_case(10)]
    #[test_case(-10 ; "minus 10")]
    fn when_foo_has_bar(foo: i32) {
        assert_eq!(10, foo);
    }

    #[test_case(50 => 60)]
    #[test_case(10 => 5)]
    #[test_case(-10 => -10)]
    fn when_foo_has_baz(foo: i32) -> i32 {
        foo + 10
    }
}
```

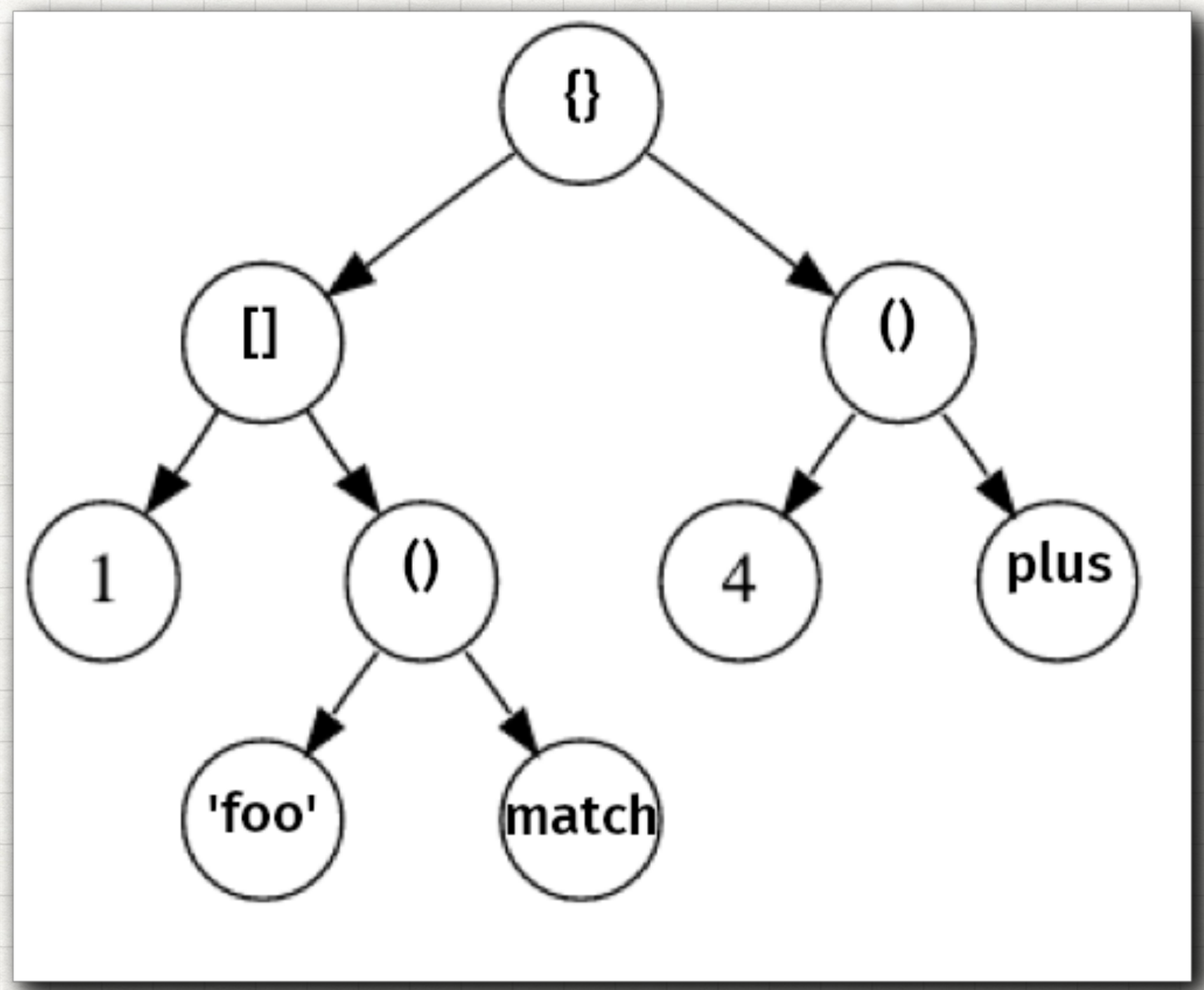
- Test case as macro attribute
- Possibility to assert returning value
- Possibility to rename test case
- Fast n Furious (zero cost abstraction)

# RUST PROCEDURAL MACROS

## HOW DO THEY WORK? DO THEY CHANGE THINGS? LET'S FIND OUT



# WHAT IS TOKEN TREE

# WHAT IS TOKEN TREE

## IN DOCUMENTATION

[-] A single token or a delimited sequence of token trees (e.g., `[1, (), ..]`).

### Variants

`Group(Group)`

[-] A token stream surrounded by bracket delimiters.

`Ident(Ident)`

[-] An identifier.

`Punct(Punct)`

[-] A single punctuation character (`+`, `,`, `$`, etc.).

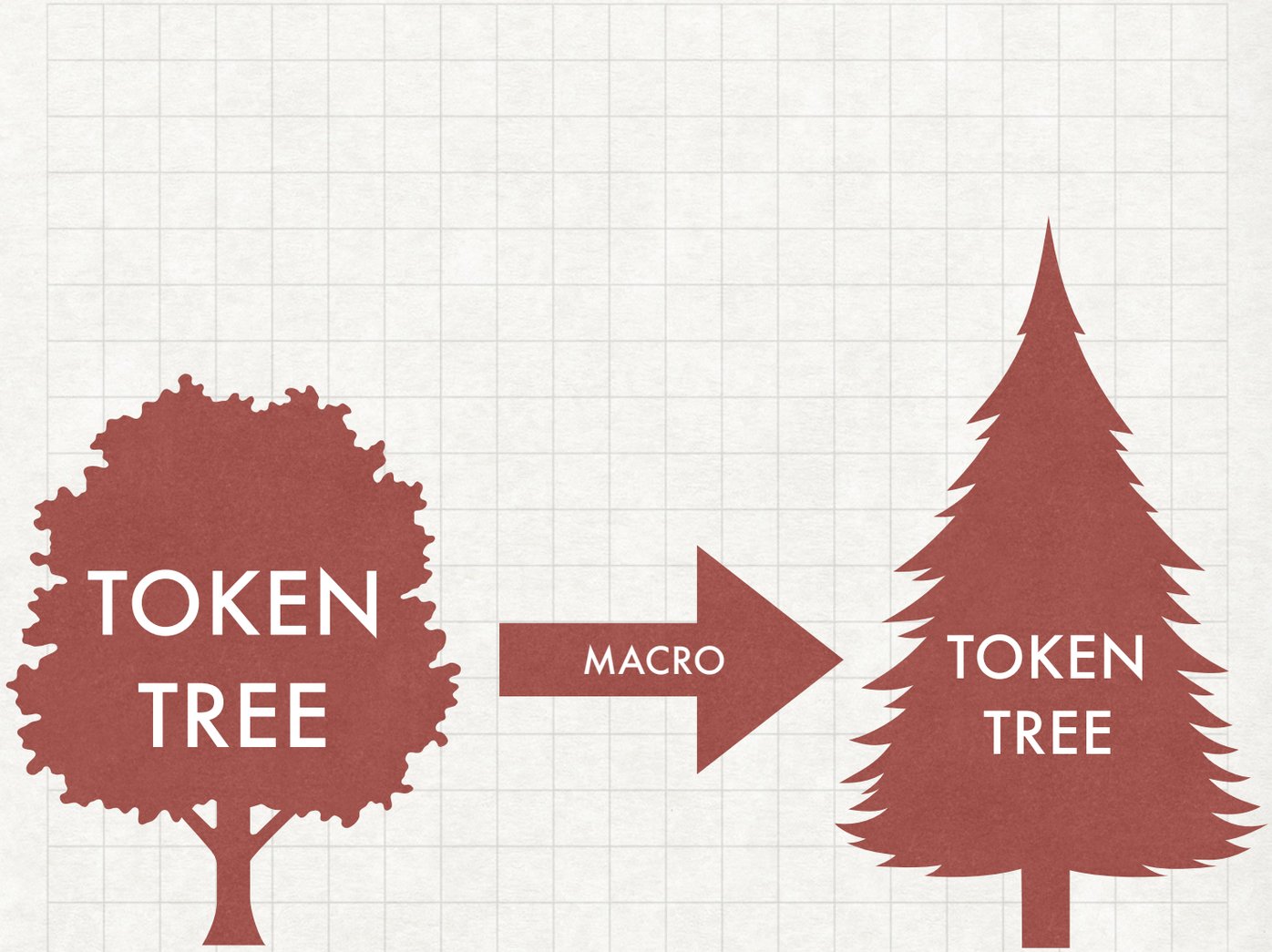
`Literal(Literal)`

[-] A literal character (`'a'`), string (`"hello"`), number (`2.3`), etc.

# WHAT MACRO DOES

## AND DOES IT VERY WELL

- Reads token tree from compiler (PARSE)
- Analyses it
- Transforms
- Writes back as a new token tree (WRITE)



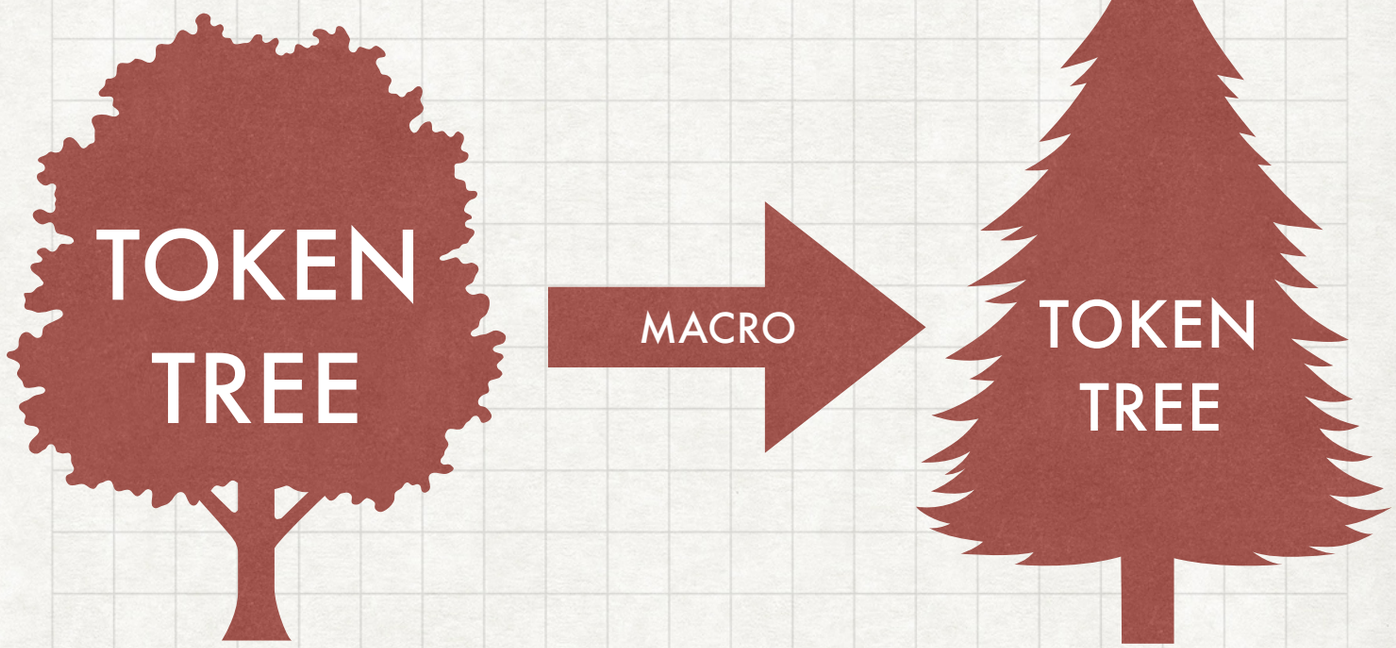
# WHAT MACRO DOES

## WHAT TOOLS WE WANT TO USE

- Reads token tree from compiler (PARSE)
- Analyses it
- Transforms
- Writes back as a new token tree (RENDER)

SYN

QUOTE



# CARGO.TOML

```toml
[lib]
doctest = false
proc-macro = true
path = "src/lib.rs"

[dependencies]
syn = { version = "1.0", features = ["full", "extra-traits"] }
quote = "1.0"
proc-macro2 = { version = "1.0", features = [] }
lazy_static = "1.4.0"
```

# TEST_CASE MACRO

## ATTRIBUTE

Arguments expressions delimited by comma

```
#[test_case(50)]
#[test_case(10)]
#[test_case(-10 ; "minus 10")]
f  when_foo_has_ba  foo: i32) {
    assert_eq!(10, foo)
}

#[test_case(50 => 60)]
```

Function (another token tree), signature and body

(optional) `;` token and literal string with test case description

(optional) `=>` token and expected result expression

LIBRARY FOR PARSING

# SYN V1.0

```rust
use syn::{Expr, LitStr};

pub struct TestCase {
    test_case_name: String,
    args: Vec<Expr>,
    expected: Option<Expr>,
    case_desc: Option<LitStr>,
}
```

- Logical structure of the attribute
- No information about function
- One function may have more than

  one test case
- Expr - parsed expression
- LitStr - literal string eg. “foo”

```rust
use syn::{Expr, LitStr};

pub struct TestCase {
    test_case_name: String,
    args: Vec<Expr>,
    expected: Option<Expr>,
    case_desc: Option<LitStr>,
}
```

# SYN V1.0

# HOW TO PARSE ATTRIBUTE

```rust
fn parse(input: ParseStream) -> Result<Self, Error> {
    let mut args = vec![];
    loop {
        let exp: Expr = input.parse()?;
        args.push(exp);
        if !input.peek(Token![,]) {
            break;
        }
        let _comma: Token![,] = input.parse()?;
    }

    let arrow: Option<Token![=>]> = input.parse()?;
    let expected = if arrow.is_some() {
        let expr: Expr = input.parse()?;
        Some(expr)
    } else {
        None
    };

    let semicolon: Option<Token![;]> = input.parse()?;
    let case_desc = if semicolon.is_some() {
        let desc: LitStr = input.parse()?;
        Some(desc)
    } else {
        None
    };

    Ok(Self {
        test_case_name,
        args,
        expected,
        case_desc,
    })
}
```

- How to parse Token Stream into structure - by implementing one Trait!
- ParseStream::parse(&self)
- ParseStream::peek(&self) -> bool
- Or Option<Token[;]>
- Token![] macro

```rust
fn parse(input: ParseStream) -> Result<Self, Error> {
    let mut args = vec![];
    loop {
        let exp: Expr = input.parse()?;
        args.push(exp);
        if !input.peek(Token![,]) {
            break;
        }
        let _comma: Token![,] = input.parse()?;
    }

    let arrow: Option<Token![=>]> = input.parse()?;
    let expected = if arrow.is_some() {
        let expr: Expr = input.parse()?;
        Some(expr)
    } else {
        None
    };

    let semicolon: Option<Token![;]> = input.parse()?;
    let case_desc = if semicolon.is_some() {
        let desc: LitStr = input.parse()?;
        Some(desc)
    } else {
        None
    };

    Ok(Self {
        test_case_name,
        args,
        expected,
        case_desc,
    })
}
```

# MACRO FUNCTION

```rust
#[proc_macro_attribute]
pub fn test_case(args: TokenStream, input: TokenStream) -> TokenStream {
    let test_case = parse_macro_input!(args as TestCase);
    let mut item = parse_macro_input!(input as ItemFn);
   ...
}
```

# NOW, HOW TO RENDER?

# QUOTE V1.0

```rust
fn render_test_cases(test_cases: &[TestCase], item: ItemFn) -> TokenStream {
    let mut rendered_test_cases = vec![];

    for test_case in test_cases {
        rendered_test_cases.push(test_case.render(item.clone()));
    }

    let mod_name = item.sig.ident.clone();

    let output = quote! {
        mod #mod_name {
            #[allow(unused_imports)]
            use super::*;

            #[allow(unused_attributes)]
            #item

            #(#rendered_test_cases)*
        }
    };

    output.into()
}
```

# QUOTE V1.0

# HOW TO RENDER OUTPUT

```rust
//impl TestCase
pub fn render(&self, item: ItemFn) -> TokenStream2 {
        let item_name = item.sig.ident.clone();
        let arg_values = self.args.iter();
        let test_case_name = self.test_case_name();
        let inconclusive = self
            .case_desc
            .as_ref()
            .map(|cd| cd.value().to_lowercase().contains("inconclusive"))
            .unwrap_or_default();

        let expected: Expr = match &self.expected {
            Some(e) => parse_quote! {
                assert_eq!(#e, _result)
            },
            None => parse_quote! {()},
        };

        let mut attrs = vec![];
        if inconclusive {
            attrs.push(parse_quote! { #[ignore] });
        }
        attrs.append(&mut item.attrs.clone());

        quote! {
            #[test]
            #(#attrs)*
            fn #test_case_name() {
                let _result = #item_name(#(#arg_values),*);
                #expected
            }
        }
    }
```

- quote! macro to generate new token stream
- parse_quote! To generate new stream and automatically parse it by syn crate
- #( #vars )* syntax to render each item in collection
- #( #vars ),* to join with comma

```rust
//impl TestCase
pub fn render(&self, item: ItemFn) -> TokenStream2 {
        let item_name = item.sig.ident.clone();
        let arg_values = self.args.iter();
        let test_case_name = self.test_case_name();
        let inconclusive = self
            .case_desc
            .as_ref()
            .map(|cd| cd.value().to_lowercase().contains("inconclusive"))
            .unwrap_or_default();

        let expected: Expr = match &self.expected {
            Some(e) => parse_quote! {
                assert_eq!(#e, _result)
            },
            None => parse_quote! {()},
        };

        let mut attrs = vec![];
        if inconclusive {
            attrs.push(parse_quote! { #[ignore] });
        }
        attrs.append(&mut item.attrs.clone());

        quote! {
            #[test]
            #(#attrs)*
            fn #test_case_name() {
                let _result = #item_name(#(#arg_values),*);
                #expected
            }
        }
    }
```

STORY 2

# DERIVE VALIDATE

```rust
#[derive(ValidateInput)]
struct C {
    #[add_validation_with(validate_email)]
    pub email: String, // <- self.a.validate_input("email")?;
                       // <- validate_room_code(&self.a, "email")?;
    pub a: String      // <- self.b.validate_input("a")?;
}

fn validate_email(email: &str, field: &str) -> Result<(), Error> {
    // Custom code here

    Ok(())
}
```

# VALIDATE INPUT

```rust
#[derive(Serialize)]
struct UserData {
    pub email: String,
    pub age: u32
}

...
fn http_post_handler(body: &str) -> Result<(), Error> {
  let data: UserData = serde_json::from_str(body)?;
  data.validate()?;
  ...
}
```

# VALIDATE TRAIT

## OUR LITTLE MECHANISM WHICH WE WANT DERIVE

```rust
enum NoError;

trait Validate {
  type Error;
  fn validate_field(&self, _field: &str) -> Result<(), Self::Error> { Ok(()) }
  fn validate_fields(&self) -> Result<(), Self::Error> { Ok (()) }
}

macro_rules! default_validate {
  () => ();
  ($($t: ty),+) => ($( impl Validate for $t { type Error = NoError; } )+)
}

default_validate!{u8, u32, u64, bool, f64, char, usize, i32, i64, ... }
```

# VALIDATE TRAIT

## IMPLEMENTATION

```rust
use crate::DateRange;
use crate::Error;

impl Validate for DateRange {
  type Error = Error;

  fn validate_field(&self, field: &str) -> Result<(), Self::Error> {
    if self.start > self.end {
      return Err(Error::InvalidDatesInUserInput {
        from: self.start.clone(),
        to: self.end.clone(),
        field: field.to_string()
      });
    }
    Ok(())
  }
}
```

# USER DATA

## IMP OUR TRAIT

```rust
#[derive(Serialize])
struct UserData {
  pub email: String,
  pub age: u32
}

impl Validate for UserData {
  type Error = Error;

  fn validate_fields(&self) -> Result<(), Self::Error> {
    self.email.validate_field("email")?; //check forbidden chars
    self.age.validate_field("age")?;

    validate_email(&self.email, "email")?; //check if has proper structure
    validate_age(&self.age, "age")?; // check if user has more than 16 y.
    Ok(());
  }
}
```

# WHAT WE WANT?

# DERIVE MACROS!

```rust
#[derive(Serialize, Validate)]
struct UserData {
  #[validate_field_with(validate_email)]
  pub email: String,

  #[validate_field_with(validate_age)]
  pub age: u32
}
```

# DERIVE MACRO

## FOR VALIDATE TRAIT

- Each field has to be validated
- Each field has to implement Validate trait (or use explicit attribute to skip it)
- Extra functions with validations are optional, But necessary

```rust
#[derive(Serialize, Validate)]
struct UserData {
  #[validate_field_with(validate_email)]
  pub email: String,

  #[validate_field_with(validate_age)]
  pub age: u32
}
```

# DERIVE MACRO

## MAIN FUNCTION

```rust
#[proc_macro_derive(Validate, attributes(validate_field_with, skip_validation))]
pub fn validate_fn(input: TokenStream) -> TokenStream {
  let input = parse_macro_input!(input as DeriveInput);

  let expanded = quote! {
    ...
  };

  TokenStream::from(expanded)
}
```

```rust
#[proc_macro_attribute]
pub fn test_case(args: TokenStream, input: TokenStream) -> TokenStream {
    let test_case = parse_macro_input!(args as TestCase);
    let mut item = parse_macro_input!(input as ItemFn);
   ...
}
```

# RETRIEVE VALIDATIONS

```
if let syn::Data::Struct(DataStruct { fields: Fields::Named(fields), .. }) = input.data {
  for field in fields.named {
    let ident = field.ident;
    let ident_name = ident.as_ref().map(|i| format!("{}", i));
    let mut skip = false;

    for attr in field.attrs {
      if attr.path == parse_quote!(skip_validation) {
        skip = true;
        continue;
      }

      if attr.path == parse_quote!(validate_field_with) {
        let name = attr.tts;
        use_format_field = true;
        validations.push(quote! {
          #name(&self.#ident, &format_field(field, #ident_name))?;
        });
      }
    }

    if skip == false {
      use_format_field = true;
      validations.push(quote! {
        Validate::validate_field(&self.#ident, &format_field(field, #ident_name))?;
      });
    }
  }
}
```

# RENDER DERIVE MACRO

```rust
let expanded = quote! {
  impl #impl_generics Validate for #name #ty_generics #where_clause {
    type Error = Error; //For simplicity of presentation I assume there is Error type

    fn validate_field(&self, field: &str) -> Result<(), Error> {
      let field = Some(field);
      #(#validations)*
      Ok(())
    }

    fn validate_fields(&self) -> Result<(), Error> {
      let field: Option<&str> = None;
      #(#validate_inputs_validations)*
      Ok(())
    }
  }
};

TokenStream::from(expanded)
```

# QUESTIONS?

# THANK YOU!