20 APRIL, 2019

(Re:) Writing A Custom Derive From Zero

Presented by Xidorn Quan

A typical type in Servo's style system

```rust
/// Either `<color>` or `auto`.
pub enum ColorOrAuto<C> {
    /// A `<color>
    Color(C),
    /// `auto`
    Auto,
}
```

A typical type in Servo's style system

```rust
#[derive(
    Animate,
    Clone,
    ComputeSquaredDistance,
    Copy,
    Debug,
    MallocSizeOf,
    PartialEq,
    Parse,
    SpecifiedValueInfo,
    ToAnimatedValue,
    ToAnimatedZero,
    ToComputedValue,
    ToCss,
)]
```

```rust
/// Either `<color>` or `auto`.
pub enum ColorOrAuto<C> {
    /// A `<color>
    Color(C),
    /// `auto`
    Auto,
}
```

A typical type in Servo's style system

```rust
#[derive(
    Animate,
    Clone,
    ComputeSquaredDistance,
    Copy,
    Debug,
    MallocSizeOf,
    PartialEq,
    Parse,
    SpecifiedValueInfo,
    ToAnimatedValue,
    ToAnimatedZero,
    ToComputedValue,
    ToCss,
)]
```

```rust
/// Either `<color>` or `auto`.
pub enum ColorOrAuto<C> {
    /// A `<color>
    Color(C),
    /// `auto`
    Auto,
}
```

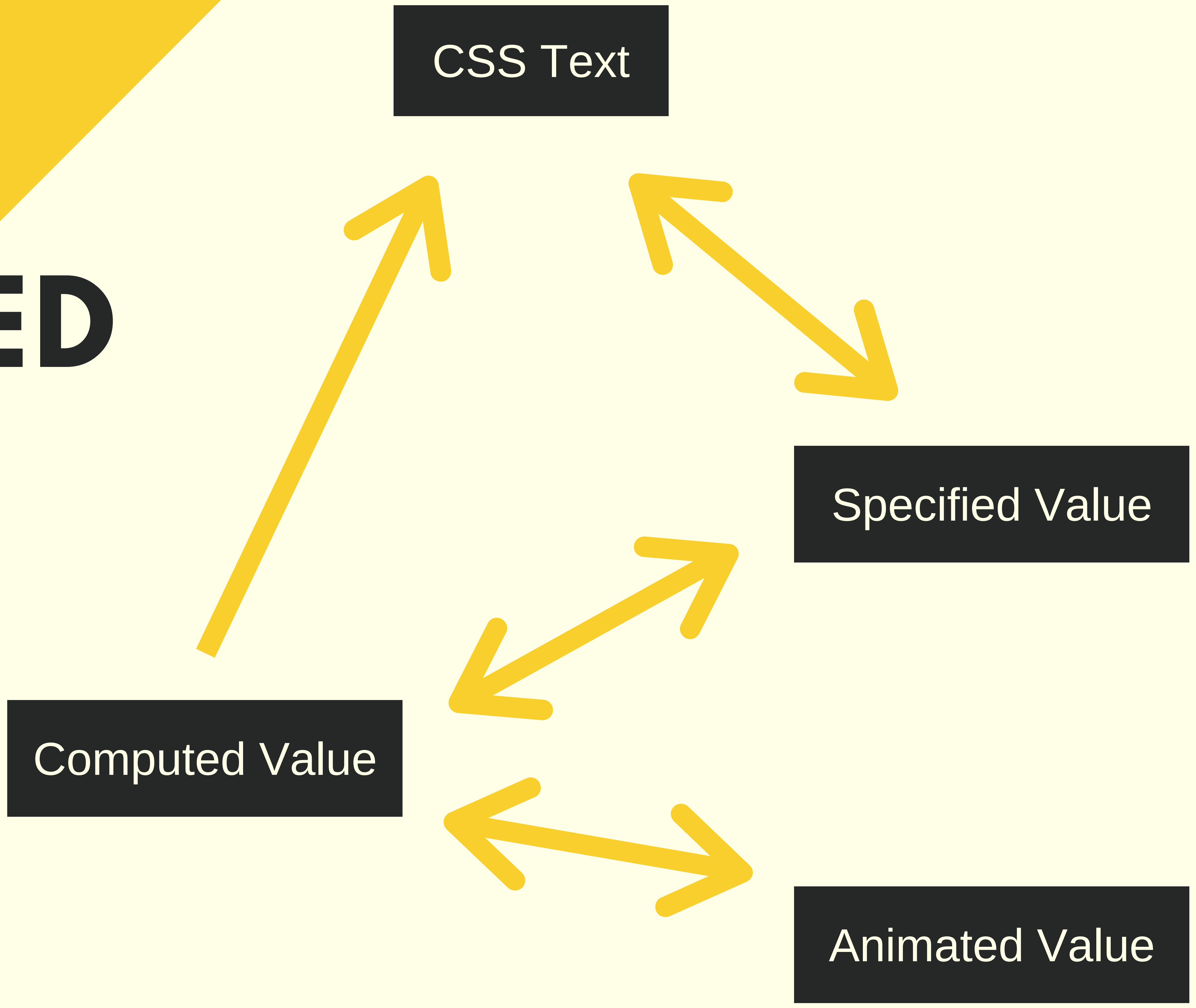
without custom derive...

```rust
use cssparser::{Parser, Token};
use parser::{Parse, ParserContext};
use std::fmt::{self, Write};
use style_traits::{CssWriter, KeywordsCollectFn, ParseError, SpecifiedValueInfo, ToCss};
use values::animated::{Animate, Procedure, ToAnimatedValue, ToAnimatedZero};
use values::computed::{Context, ToComputedValue};
use values::distance::{ComputeSquaredDistance, SquaredDistance};

impl<C: Animate> Animate for ColorOrAuto<C> {
    fn animate(&self, other: &Self, procedure: Procedure) -> Result<Self, ()> {
        match (self, other) {
            (&ColorOrAuto::Color(ref this), &ColorOrAuto::Color(ref other)) => {
                this.animate(other, procedure).map(ColorOrAuto::Color)
            }
            (&ColorOrAuto::Auto, &ColorOrAuto::Auto) => {
                Ok(ColorOrAuto::Auto)
            }
            _ => Err(())
        }
    }
}

impl<C: ComputeSquaredDistance> ComputeSquaredDistance for ColorOrAuto<C> {
    fn compute_squared_distance(&self, other: &Self) -> Result<SquaredDistance, ()> {
        match (self, other) {
            (&ColorOrAuto::Color(ref this), &ColorOrAuto::Color(ref other)) => {
                this.compute_squared_distance(other)
            }
            (&ColorOrAuto::Auto, &ColorOrAuto::Auto) => {
                Ok(SquaredDistance::from_sqrt(0.))
            }
            _ => Err(())
        }
    }
}
```

```rust
impl<C: Parse> Parse for ColorOrAuto<C> {
    fn parse<'i, 't>(
        context: &ParserContext,
        input: &mut Parser<'i, 't>,
    ) -> Result<Self, ParseError<'i>> {
        if let Ok(v) = input.try(|i| C::parse(context, i)) {
            return Ok(ColorOrAuto::Color(v));
        }
        let location = input.current_source_location();
        let ident = input.expect_ident()?;
        match_ignore_ascii_case! { &ident,
            "auto" => Ok(ColorOrAuto::Auto),
            _ => Err(location.new_unexpected_token_error(Token::Ident(ident.clone()))),
        }
    }
}

impl<C: SpecifiedValueInfo> SpecifiedValueInfo for ColorOrAuto<C> {
    const SUPPORTED_TYPES: u8 = C::SUPPORTED_TYPES;
    fn collect_completion_keywords(f: KeywordsCollectFn) {
        C::collect_completion_keywords(f);
        f(&["auto"]);
    }
}

impl<C: ToAnimatedValue> ToAnimatedValue for ColorOrAuto<C> {
    type AnimatedValue = ColorOrAuto<C::AnimatedValue>;
    fn to_animated_value(self) -> Self::AnimatedValue {
        match self {
            ColorOrAuto::Color(c) => ColorOrAuto::Color(c.to_animated_value()),
            ColorOrAuto::Auto => ColorOrAuto::Auto,
        }
    }
    fn from_animated_value(animated: Self::AnimatedValue) -> Self {
        match animated {
            ColorOrAuto::Color(c) => ColorOrAuto::Color(C::from_animated_value(c)),
            ColorOrAuto::Auto => ColorOrAuto::Auto,
        }
    }
}
```

```rust
impl<C: ToAnimatedZero> ToAnimatedZero for ColorOrAuto<C> {
    fn to_animated_zero(&self) -> Result<Self, ()> {
        match self {
            ColorOrAuto::Color(c) => c.to_animated_zero().map(ColorOrAuto::Color),
            ColorOrAuto::Auto => Ok(ColorOrAuto::Auto),
        }
    }
}

impl<C: ToComputedValue> ToComputedValue for ColorOrAuto<C> {
    type ComputedValue = ColorOrAuto<C::ComputedValue>;
    fn to_computed_value(&self, context: &Context) -> Self::ComputedValue {
        match self {
            ColorOrAuto::Color(c) => ColorOrAuto::Color(c.to_computed_value(context)),
            ColorOrAuto::Auto => ColorOrAuto::Auto,
        }
    }
    fn from_computed_value(computed: &Self::ComputedValue) -> Self {
        match computed {
            ColorOrAuto::Color(c) => ColorOrAuto::Color(C::from_computed_value(c)),
            ColorOrAuto::Auto => ColorOrAuto::Auto,
        }
    }
}

impl<C: ToCss> ToCss for ColorOrAuto<C> {
    fn to_css<W: fmt::Write>(&self, dest: &mut CssWriter<W>) -> fmt::Result {
        match self {
            ColorOrAuto::Color(c) => c.to_css(dest),
            ColorOrAuto::Auto => dest.write_str("auto"),
        }
    }
}
```

CSS IS COMPLICATED



Custom derive in Servo's style system

Conversion between data forms

Recursive computation

Simple compile-time reflection

To write a custom derive, you need

clear idea about the code it generates

```rust
pub trait ToCss {
    fn to_css<W>(&self, dest: &mut W) -> fmt::Result where W: fmt::Write;
}
```

```rust
pub trait ToCss {
    fn to_css<W>(&self, dest: &mut W) -> fmt::Result where W: fmt::Write;
}
```

Cargo.toml

```toml
[lib]
proc-macro = true
```

lib.rs

```rust
extern crate proc_macro;
use proc_macro::TokenStream;
#[proc_macro_derive(ToCss)]
pub fn derive_to_css(input: TokenStream) -> TokenStream {
    unimplemented!()
}
```

```rust
pub trait ToCss {
    fn to_css<W>(&self, dest: &mut W) -> fmt::Result where W: fmt::Write;
}
```

Cargo.toml

```toml
[lib]
proc-macro = true
```

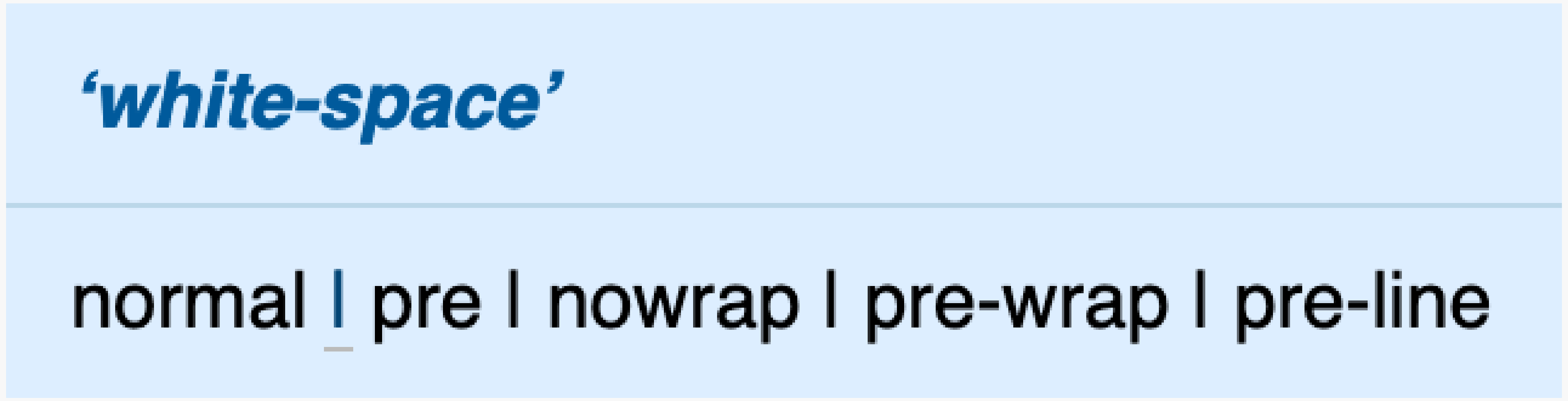
lib.rs

```rust
extern crate proc_macro;
use proc_macro::TokenStream;
#[proc_macro_derive(ToCss)]
pub fn derive_to_css(input: TokenStream) -> TokenStream {
    unimplemented!()
}
```

Required even in Rust 2018 edition

HOW TO WRITE A CUSTOM DERIVE FOR

SIMPLE ENUMS


‘white-space’

normal | pre | nowrap | pre-wrap | pre-line

```
pub enum WhiteSpace {
    Normal,
    Nowrap,
    Pre,
    PreWrap,
    PreLine,
}
```

```rust
pub enum WhiteSpace {
    Normal,
    Nowrap,
    Pre,
    PreWrap,
    PreLine,
}
```

```rust
impl ToCss for WhiteSpace {
    fn to_css<W>(&self, dest: &mut W) -> fmt::Result
    where
        W: fmt::Write,
    {
        match self {
            WhiteSpace::Normal => dest.write_str("normal"),
            WhiteSpace::Nowrap => dest.write_str("nowrap"),
            WhiteSpace::Pre => dest.write_str("pre"),
            WhiteSpace::PreWrap => dest.write_str("pre-wrap"),
            WhiteSpace::PreLine => dest.write_str("pre-line"),
        }
    }
}
```

```
impl ToCss for WhiteSpace {
    fn to_css<W>(&self, dest: &mut W) -> fmt::Result
    where
        W: fmt::Write,
    {
        match self {
            WhiteSpace::Normal => dest.write_str("normal"),
            WhiteSpace::Nowrap => dest.write_str("nowrap"),
            WhiteSpace::Pre => dest.write_str("pre"),
            WhiteSpace::PreWrap => dest.write_str("pre-wrap"),
            WhiteSpace::PreLine => dest.write_str("pre-line"),
        }
    }
}
```

Mostly type-neutral

```rust
#[recursion_limit = "128"]

use proc_macro::TokenStream;
use quote::quote;

TokenStream::from(quote! {
    impl style_traits::ToCss for /* ??? */ {
        fn to_css<W>(&self, dest: &mut W) -> std::fmt::Result
        where
            W: std::fmt::Write,
        {
            match self {
                /* ??? */
            }
        }
    }
})
```

```
#[recursion_limit = "128"]

use proc_macro::TokenStream;
use quote::quote;

TokenStream::from(quote! {
    impl style_traits::ToCss for /* ??? */ {
        fn to_css<W>(&self, dest: &mut W) -> std::fmt::Result
        where
            W: std::fmt::Write,
        {
            match self {
                /* ??? */
            }
        }
    }
})
```

For quote!()

proc_macro::TokenStream vs. proc_macro2::TokenStream

```rust
#[recursion_limit = "128"]

use proc_macro::TokenStream;
use quote::quote;

TokenStream::from(quote! {
    impl style_traits::ToCss for /* ??? */ {
        fn to_css<W>(&self, dest: &mut W) -> std::fmt::Result
        where
            W: std::fmt::Write,
        {
            match self {
                /* ??? */
            }
        }
    }
})
```

Absolute path

```
#[recursion_limit = "128"]

use proc_macro::TokenStream;
use quote::quote;

TokenStream::from(quote! {
    impl style_traits::ToCss for /* ??? */ {
        fn to_css<W>(&self, dest: &mut W) -> std::fmt::Result
        where
            W: std::fmt::Write,
        {
            match self {
                /* ??? */
            }
        }
    }
})
```

Use #var to add variable content in quote!()

```diff
TokenStream::from(quote! {
-   impl style_traits::ToCss for /* ??? */ {
+   impl style_traits::ToCss for #name {
        fn to_css<W>(&self, dest: &mut W) -> std::fmt::Result
        where
            W: std::fmt::Write,
        {
            match self {
-               /* ??? */
+               #match_body
            }
        }
    }
})
```

```rust
use heck::KebabCase;
use proc_macro::TokenStream;
use proc_macro2::TokenStream as TokenStream2;
use quote::quote;
use syn::{parse_macro_input, Data, DeriveInput, Fields};

let input = parse_macro_input!(input as DeriveInput);
let name = input.ident;
let match_body = match input.data {
    Data::Enum(data) => {
        data.variants.into_iter().flat_map(|variant| {
            match variant.fields {
                Fields::Unit => {
                    let ident = variant.ident;
                    let value = ident.to_string().to_kebab_case();
                    quote! {
                        #name::#ident => std::fmt::Write::write_str(dest, #value),
                    }
                }
                _ => panic!("unsupported variant fields"),
            }
        }).collect::<TokenStream2>()
    }
    _ => panic!("unsupported data structure"),
};
```

```rust
use heck::KebabCase;
use proc_macro::TokenStream;
use proc_macro2::TokenStream as TokenStream2;
use quote::quote;
use syn::{parse_macro_input, Data, DeriveInput, Fields};

let input = parse_macro_input!(input as DeriveInput);
let name = input.ident;
let match_body = match input.data {
    Data::Enum(data) => {
        data.variants.into_iter().flat_map(|variant| {
            match variant.fields {
                Fields::Unit => {
                    let ident = variant.ident;
                    let value = ident.to_string().to_kebab_case();
                    quote! {
                        #name::#ident => std::fmt::Write::write_str(dest, #value),
                    }
                }
                _ => panic!("unsupported variant fields"),
            }
        }).collect::<TokenStream2>()
    }
    _ => panic!("unsupported data structure"),
};
```

Use syn to parse input as DeriveInput

Generate a match branch for each variant

```rust
use heck::KebabCase;
use proc_macro::TokenStream;
use proc_macro2::TokenStream as TokenStream2;
use quote::quote;
use syn::{parse_macro_input, Data, DeriveInput, Fields};

let input = parse_macro_input!(input as DeriveInput);
let name = input.ident;
let match_body = match input.data {
    Data::Enum(data) => {
        data.variants.into_iter().flat_map(|variant| {
            match variant.fields {
                Fields::Unit => {
                    let ident = variant.ident;
                    let value = ident.to_string().to_kebab_case();
                    quote! {
                        #name::#ident => std::fmt::Write::write_str(dest, #value),
                    }
                }
                _ => panic!("unsupported variant fields"),
            }
        }).collect::<TokenStream2>()
    }
    _ => panic!("unsupported data structure"),
};
```

Qualified syntax with absolute path

```rust
use heck::KebabCase;
use proc_macro::TokenStream;
use proc_macro2::TokenStream as TokenStream2;
use quote::quote;
use syn::{parse_macro_input, Data, DeriveInput, Fields};

let input = parse_macro_input!(input as DeriveInput);
let name = input.ident;
let match_body = match input.data {
    Data::Enum(data) => {
        data.variants.into_iter().flat_map(|variant| {
            match variant.fields {
                Fields::Unit => {
                    let ident = variant.ident;
                    let value = ident.to_string().to_kebab_case();
                    quote! {
                        #name::#ident => std::fmt::Write::write_str(dest, #value),
                    }
                }
                _ => panic!("unsupported variant fields"),
            }
        }).collect::<TokenStream2>()
    }
    _ => panic!("unsupported data structure"),
};
```

Extract into a separate module

```
extern crate proc_macro;
use proc_macro::TokenStream;

mod to_css;

#[proc_macro_derive(ToCss)]
pub fn derive_to_css(input: TokenStream) -> TokenStream {
    to_css::derive(syn::parse_macro_input!(input)).into()
}
```

HOW TO WRITE A CUSTOM DERIVE FOR

ENUMS WITH FIELDS

```rust
pub enum InitialLetters {
    Normal,
    Values(f32, i32),
}
```

```rust
pub enum InitialLetters {
    Normal,
    Values(f32, i32),
}
```

```rust
match self {
    InitialLetters::Normal => dest.write_str("normal"),
    InitialLetters::Values(v1, v2) => write!(dest, "{} {}", v1, v2),
}
```

```
match self {
    InitialLetters::Normal => dest.write_str("normal"),
    InitialLetters::Values(v1, v2) => write!(dest, "{} {}", v1, v2),
}
```

```
match self {
    InitialLetters::Normal => dest.write_str("normal"),
    InitialLetters::Values(v1, v2) => {
        v1.to_css(dest)?;
        dest.write_char(' ')?;
        v2.to_css(dest)?;
        Ok(())
    }
}
```

```
Fields::Unnamed(fields) => {
    let bindings = &(0..fields.unnamed.len())
        .map(|i| Ident::new(&format!("v{}", i), Span::call_site()))
        .collect::<Vec<_>>();
    let is_first = iter::once(true).chain(iter::repeat(false));
    quote! {
        #name::#ident(#(#bindings),*) => {
            #(
                if !#is_first {
                    std::fmt::Write
                }
                style_traits::ToCss
            )*
            Ok(())
        }
    }
}
```

```
data.variants.into_iter().flat_map(|variant| {
    match variant.fields {
        Fields::Unit => {
            let ident = variant.ident;
            let value = ident.to_string().to_kebab_case();
            quote! {
                #name::#ident => std::fmt::Write::write_str(dest, #value),
            }
        }
        _ => panic!("unsupported variant fields"),
    }
}).collect::<TokenStream2>()
```

```
Fields::Unnamed(fields) => {
    let bindings = &(0..fields.unnamed.len())
        .map(|i| Ident::new(&format!("v{}", i), Span::call_site()))
        .collect::<Vec<_>>();
    let is_first = iter::once(true).chain(iter::repeat(false));
    quote! {
        #name::#ident(#(#bindings),*) => {
            #(
                if !#is_first {
                    std::fmt::Write::write_char(dest, ' ')?;
                }
                style_traits::ToCss::to_css(#bindings, dest)?;
            )*
            Ok(())
        }
    }
}
```

Generate binding variables

```rust
Fields::Unnamed(fields) => {
    let bindings = &(0..fields.unnamed.len())
        .map(|i| Ident::new(&format!("v{}", i), Span::call_site()))
        .collect::<Vec<_>>();
    let is_first = iter::once(true).chain(iter::repeat(false));
    quote! {
        #name::#ident(#(#bindings),*) => {
            #(
                if !#is_first {
                    std::fmt::Write::write_char(dest, ' ')?;
                }
                style_traits::ToCss::to_css(#bindings, dest)?;
            )*
            Ok(())
        }
    }
}
```

Ident for identifier

```
Fields::Unnamed(fields) => {
    let bindings = &(0..fields.unnamed.len())
        .map(|i| Ident::new(&format!("v{}", i), Span::call_site()))
        .collect::<Vec<_>>();
    let is_first = iter::once(true).chain(iter::repeat(false));
    quote! {
        #name::#ident(#(#bindings),*) => {
            #(
                if !#is_first {
                    std::fmt::Write::write_char(dest, ' ')?;
                }
                style_traits::ToCss::to_css(#bindings, dest)?;
            )*
            Ok(())
        }
    }
}
```

Repetition as in macro_rules!

HOW TO WRITE A CUSTOM DERIVE FOR

STRUCTS

```rust
pub struct CounterPair {
    name: CustomIdent,
    value: i32,
}
```

```rust
self.name.to_css(dest)?;
dest.write_char(' ')?;
self.value.to_css(dest)?;
Ok(())
```

```
self.name.to_css(dest)?;
dest.write_char(' ')?;
self.value.to_css(dest)?;
Ok(())
```

```
match self {
    CounterPair { name, value } => {
        name.to_css(dest)?;
        dest.write_char(' ')?;
        value.to_css(dest)?;
        Ok(())
    }
}
```

HOW TO WRITE A CUSTOM DERIVE FOR

GENERIC TYPES

```rust
pub enum ColorOrAuto<C> {
    Color(C),
    Auto,
}
```

```rust
pub enum ColorOrAuto<C> {
    Color(C),
    Auto,
}
```

```rust
impl<C> ToCss for ColorOrAuto<C>
where
    C: ToCss,
{
    fn to_css<W>(&self, dest: &mut W) -> fmt::Result
    where
        W: fmt::Write,
    {
        match self {
            ColorOrAuto::Color(c) => c.to_css(dest),
            ColorOrAuto::Auto => dest.write_str("auto"),
        }
    }
}
```

```rust
pub enum ColorOrAuto<C> {
    Color(C),
    Auto,
}
```

```rust
impl<C> ToCss for ColorOrAuto<C>
where
    C: ToCss,
{
    fn to_css<W>(&self, dest: &mut W) -> fmt::Result
    where
        W: fmt::Write,
    {
        match self {
            ColorOrAuto::Color(c) => c.to_css(dest),
            ColorOrAuto::Auto => dest.write_str("auto"),
        }
    }
}
```

```rust
if !input.generics.params.is_empty() {
    let mut where_clause = input.generics.where_clause.take();
    let predicates = &mut where_clause.get_or_insert(parse_quote!(where)).predicates;
    for param in input.generics.type_params() {
        let ident = &param.ident;
        predicates.push(parse_quote!(#ident: style_traits::ToCss));
    }
    input.generics.where_clause = where_clause;
}
let (impl_generics, ty_generics, where_clause) = input.generics.split_for_impl();
quote! {
    impl#impl_generics style_traits::ToCss for #name#ty_generics
    #where_clause
    {
        // same as before
    }
}
```

```rust
if !input.generics.params.is_empty() {
    let mut where_clause = input.generics.where_clause.take();
    let predicates = &mut where_clause.get_or_insert(parse_quote!(where)).predicates;
    for param in input.generics.type_params() {
        let ident = &param.ident;
        predicates.push(parse_quote!(#ident: style_traits::ToCss));
    }
    input.generics.where_clause = where_clause;
}
let (impl_generics, ty_generics, where_clause) = input.generics.split_for_impl();
quote! {
    impl#impl_generics style_traits::ToCss for #name#ty_generics
    #where_clause
    {
        // same as before
    }
}
```

Add trait bounds

```rust
if !input.generics.params.is_empty() {
    let mut where_clause = input.generics.where_clause.take();
    let predicates = &mut where_clause.get_or_insert(parse_quote!(where)).predicates;
    for param in input.generics.type_params() {
        let ident = &param.ident;
        predicates.push(parse_quote!(#ident: style_traits::ToCss));
    }
    input.generics.where_clause = where_clause;
}
let (impl_generics, ty_generics, where_clause) = input.generics.split_for_impl();
quote! {
    impl#impl_generics style_traits::ToCss for #name#ty_generics
    #where_clause
    {
        // same as before
    }
}
```

Split for impl

HOW TO WRITE A CUSTOM DERIVE WITH

ATTRIBUTES

```rust
#[derive(ToCss)]
pub enum TransformStyle {
    Flat,
    #[css(keyword = "preserve-3d")]
    Preserve3d,
}
```

```rust
#[derive(ToCss)]
pub enum TransformStyle {
    Flat,
    #[css(keyword = "preserve-3d")]
    Preserve3d,
}
```

```rust
#[derive(Default, FromVariant)]
#[darling(attributes(css), default)]
struct CssVariantAttrs {
    keyword: Option<String>,
}
```

```rust
#[derive(ToCss)]
pub enum TransformStyle {
    Flat,
    #[css(keyword = "preserve-3d")]
    Preserve3d,
}
```

```rust
#[derive(Default, FromVariant)]
#[darling(attributes(css), default)]
struct CssVariantAttrs {
    keyword: Option<String>,
}
```

```rust
let attrs = CssVariantAttrs::from_variant(&variant)
    .expect("failed to parse variant attributes");
```

```rust
#[derive(ToCss)]
pub enum TransformStyle {
    Flat,
    #[css(keyword = "preserve-3d")]
    Preserve3d,
}
```

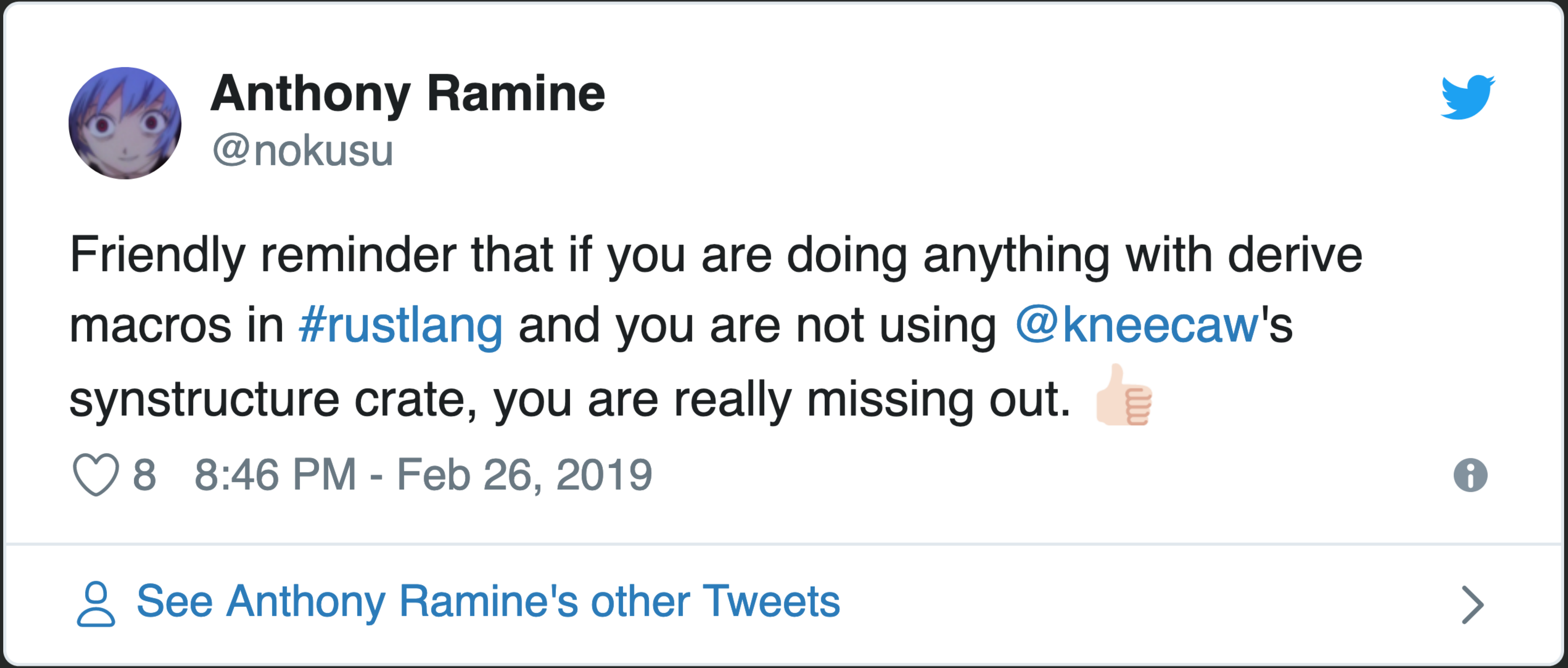
```rust
#[derive(Default, FromVariant)]
#[darling(attributes(css), default)]
struct CssVariantAttrs {
    keyword: Option<String>,
}
```

```rust
let attrs = CssVariantAttrs::from_variant(&variant)
    .expect("failed to parse variant attributes");
```

```diff
-#[proc_macro_derive(ToCss)]
+#[proc_macro_derive(ToCss, attributes(css))]
 pub fn derive_to_css(input: TokenStream) -> TokenStream {
     to_css::derive(syn::parse_macro_input!(input)).into()
 }
```

HOW TO SIMPLIFY A CUSTOM DERIVE WITH

SYNSTRUCTURE



Anthony Ramine
@nokusu

Friendly reminder that if you are doing anything with derive macros in #rustlang and you are not using @kneecaw's synstructure crate, you are really missing out. 👍🏻

8 8:46 PM - Feb 26, 2019

See Anthony Ramine's other Tweets

```rust
extern crate proc_macro;
use proc_macro::TokenStream;

mod to_css;

#[proc_macro_derive(ToCss, attributes(css))]
pub fn derive_to_css(input: TokenStream) -> TokenStream {
    to_css::derive(syn::parse_macro_input!(input)).into()
}

use heck::KebabCase;
use proc_macro2::{Span, TokenStream};
use quote::quote;
use std::iter;
use syn::{parse_quote, Data, DeriveInput, Fields, Ident};

pub fn derive(mut input: DeriveInput) -> TokenStream {
    let name = input.ident;
    let match_body = match input.data {
        Data::Struct(data) => derive_fields(&name, quote!(#name), data.fields, None),
        Data::Enum(data) => data
            .variants
            .into_iter()
            .flat_map(|variant| {
                let ident = variant.ident;
                derive_fields(&ident, quote!(#name::#ident), variant.fields)
            })
            .collect(),
        _ => panic!("unsupported data structure"),
    };

    if !input.generics.params.is_empty() {
        let mut where_clause = input.generics.where_clause.take();
        let predicates = &mut where_clause.get_or_insert(parse_quote!(where)).predicates;
        for param in input.generics.type_params() {
            let ident = &param.ident;
            predicates.push(parse_quote!(#ident: style_traits::ToCss));
        }
        input.generics.where_clause = where_clause;
    }
    let (impl_generics, ty_generics, where_clause) = input.generics.split_for_impl();
    quote! {
        impl#impl_generics style_traits::ToCss for #name#ty_generics
        #where_clause
        {
            fn to_css<W>(&self, dest: &mut W) -> std::fmt::Result
            where
                W: std::fmt::Write,
            {
                match self {
                    #match_body
                }
            }
        }
    }
}

fn derive_fields(ident: &Ident, pat: TokenStream, fields: Fields) -> TokenStream {
    match fields {
        Fields::Unit => {
            let value = ident.to_string().to_kebab_case();
            quote! {
                #pat => std::fmt::Write::write_str(dest, #value),
            }
        }
        Fields::Unnamed(fields) => {
            let bindings = (0..fields.unnamed.len())
                .map(|i| Ident::new(&format!("v{}", i), Span::call_site()))
                .collect::<Vec<_>>();
            let body = derive_fields_body(&bindings);
            quote! {
                #pat(#(#bindings),*) => { #body }
            }
        }
        Fields::Named(fields) => {
            let field_names = fields
                .named
                .into_iter()
                .map(|field| field.ident.unwrap())
                .collect::<Vec<_>>();
            let body = derive_fields_body(&field_names);
            quote! {
                #pat { #(#field_names),* } => { #body }
            }
        }
    }
}

fn derive_fields_body(bindings: &[Ident]) -> TokenStream {
    let is_first = iter::once(true).chain(iter::repeat(false));
    quote! {
        #(
            if !#is_first {
                std::fmt::Write::write_char(dest, ' ')?;
            }
            style_traits::ToCss::to_css(#bindings, dest)?;
        )*
        Ok(())
    }
}
```

With synstructure

```rust
extern crate proc_macro;
use proc_macro::TokenStream;

mod to_css;

#[proc_macro_derive(ToCss, attributes(css))]
pub fn derive_to_css(input: TokenStream) -> TokenStream {
    to_css::derive(syn::parse_macro_input!(input)).into()
}

use heck::KebabCase;
use proc_macro2::{Span, TokenStream};
use quote::quote;
use std::iter;
use syn::{parse_quote, Data, DeriveInput, Fields, Ident};

pub fn derive(mut input: DeriveInput) -> TokenStream {
    let name = input.ident;
    let match_body = match input.data {
        Data::Struct(data) => derive_fields(&name, quote!(#name), data.fields, None),
        Data::Enum(data) => data
            .variants
            .into_iter()
            .flat_map(|variant| {
                let ident = variant.ident;
                derive_fields(&ident, quote!(#name::#ident), variant.fields)
            })
            .collect(),
        _ => panic!("unsupported data structure"),
    };

    if !input.generics.params.is_empty() {
        let mut where_clause = input.generics.where_clause.take();
        let predicates = &mut where_clause.get_or_insert(parse_quote!(where)).predicates;
        for param in input.generics.type_params() {
            let ident = &param.ident;
            predicates.push(parse_quote!(#ident: style_traits::ToCss));
        }
        input.generics.where_clause = where_clause;
    }
    let (impl_generics, ty_generics, where_clause) = input.generics.split_for_impl();
    quote! {
        impl#impl_generics style_traits::ToCss for #name#ty_generics
        #where_clause
        {
            fn to_css<W>(&self, dest: &mut W) -> std::fmt::Result
            where
                W: std::fmt::Write,
            {
                match self {
                    #match_body
                }
            }
        }
    }
}

fn derive_fields(ident: &Ident, pat: TokenStream, fields: Fields) -> TokenStream {
    match fields {
        Fields::Unit => {
            let value = ident.to_string().to_kebab_case();
            quote! {
                #pat => std::fmt::Write::write_str(dest, #value),
            }
        }
        Fields::Unnamed(fields) => {
            let bindings = (0..fields.unnamed.len())
                .map(|i| Ident::new(&format!("v{}", i), Span::call_site()))
                .collect::<Vec<_>>();
            let body = derive_fields_body(&bindings);
            quote! {
                #pat(#(#bindings),*) => { #body }
            }
        }
        Fields::Named(fields) => {
            let field_names = fields
                .named
                .into_iter()
                .map(|field| field.ident.unwrap())
                .collect::<Vec<_>>();
            let body = derive_fields_body(&field_names);
            quote! {
                #pat { #(#field_names),* } => { #body }
            }
        }
    }
}

fn derive_fields_body(bindings: &[Ident]) -> TokenStream {
    let is_first = iter::once(true).chain(iter::repeat(false));
    quote! {
        #(
            if !#is_first {
                std::fmt::Write::write_char(dest, ' ')?;
            }
            style_traits::ToCss::to_css(#bindings, dest)?;
        )*
        Ok(())
    }
}
```

```rust
use heck::KebabCase;
use proc_macro2::TokenStream;
use quote::quote;
use std::iter;
use synstructure::{decl_derive, Structure};

fn derive_to_css(input: Structure) -> TokenStream {
    let body = input.each_variant(|vi| {
        let bindings = vi.bindings();
        if bindings.is_empty() {
            let value = vi.ast().ident.to_string().to_kebab_case();
            return quote! {
                std::fmt::Write::write_str(dest, #value)
            };
        }
        let is_first = iter::once(true).chain(iter::repeat(false));
        quote! {
            #(
                if !#is_first {
                    std::fmt::Write::write_char(dest, ' ')?;
                }
                style_traits::ToCss::to_css(#bindings, dest)?;
            )*
            Ok(())
        }
    });
    input.gen_impl(quote! {
        gen impl style_traits::ToCss for @Self {
            fn to_css<W>(&self, dest: &mut W) -> std::fmt::Result
            where
                W: std::fmt::Write,
            {
                match self {
                    #body
                }
            }
        }
    })
}

decl_derive!([ToCss, attributes(css)] => derive_to_css);
```

```rust
let body = input.each_variant(|vi| {
    let bindings = vi.bindings();
    if bindings.is_empty() {
        let value = vi.ast().ident.to_string().to_kebab_case();
        return quote! {
            std::fmt::Write::write_str(dest, #value)
        };
    }
    let is_first = iter::once(true).chain(iter::repeat(false));
    quote! {
        #(
            if !#is_first {
                std::fmt::Write::write_char(dest, ' ')?;
            }
            style_traits::ToCss::to_css(#bindings, dest)?;
        )*
        Ok(())
    }
});
```

```
let body = input.each_variant(|vi| {
    let bindings = vi.bindings();
    if bindings.is_empty() {
        let value = vi.ast().ident.to_string().to_kebab_case();
        return quote! {
            std::fmt::Write::write_str(dest, #value)
        };
    }
    let is_first = iter::once(true).chain(iter::repeat(false));
    quote! {
        #(
            if !#is_first {
                std::fmt::Write::write_char(dest, ' ')?;
            }
            style_traits::ToCss::to_css(#bindings, dest)?;
        )*
        Ok(())
    }
});
```

Handles bindings and enum vs. struct

```rust
let body = input.each_variant(|vi| {
    let bindings = vi.bindings();
    if bindings.is_empty() {
        let value = vi.ast().ident.to_string().to_kebab_case();
        return quote! {
            std::fmt::Write::write_str(dest, #value)
        };
    }
    let is_first = iter::once(true).chain(iter::repeat(false));
    quote! {
        #(
            if !#is_first {
                std::fmt::Write::write_char(dest, ' ')?;
            }
            style_traits::ToCss::to_css(#bindings, dest)?;
        )*
        Ok(())
    }
});
```

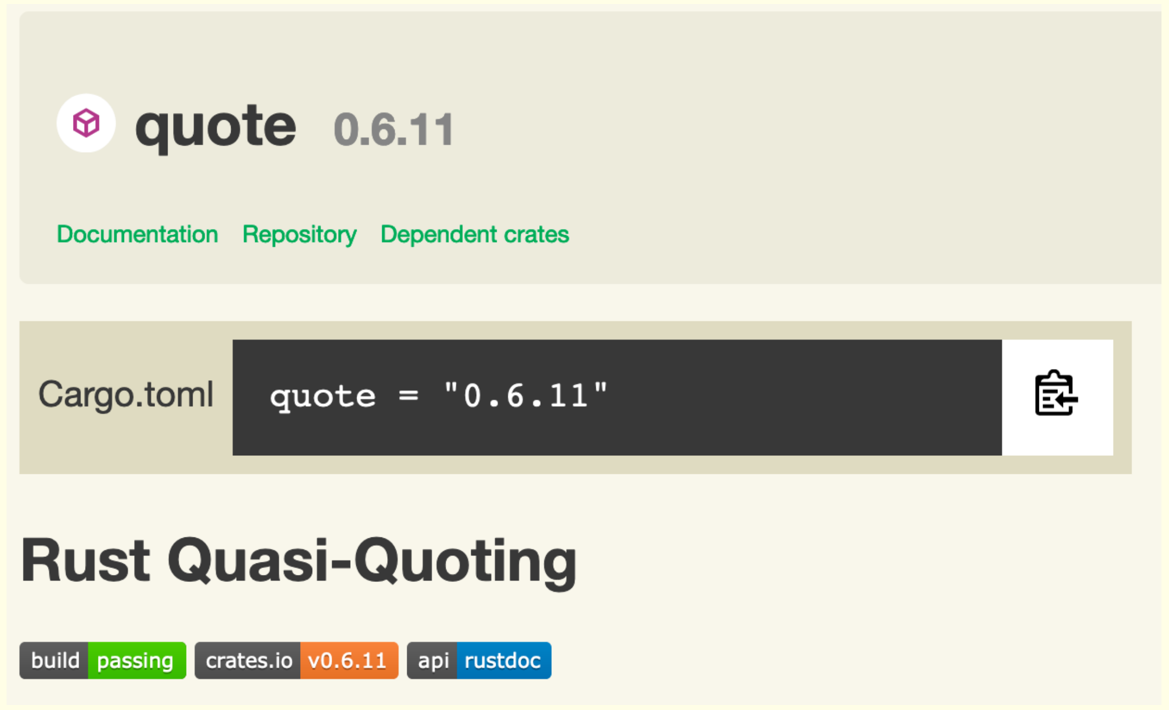
```rust
input.gen_impl(quote! {
    gen impl style_traits::ToCss for @Self {
        fn to_css<W>(&self, dest: &mut W) -> std::fmt::Result
        where
            W: std::fmt::Write,
        {
            match self {
                #body
            }
        }
    }
})
```

Handles generics

```
let body = input.each_variant(|vi| {
    let bindings = vi.bindings();
    if bindings.is_empty() {
        let value = vi.ast().ident.to_string().to_kebab_case();
        return quote! {
            std::fmt::Write::write_str(dest, #value)
        };
    }
    let is_first = iter::once(true).chain(iter::repeat(false));
    quote! {
        #(
            if !#is_first {
                std::fmt::Write::write_char(dest, ' ')?;
            }
            style_traits::ToCss::to_css(#bindings, dest)?;
        )*
        Ok(())
    }
});
```

```
input.gen_impl(quote! {
    gen impl style_traits::ToCss for @Self {
        fn to_css<W>(&self, dest: &mut W) -> std::fmt::Result
        where
            W: std::fmt::Write,
        {
            match self {
                #body
            }
        }
    }
})
```

Main crates used here



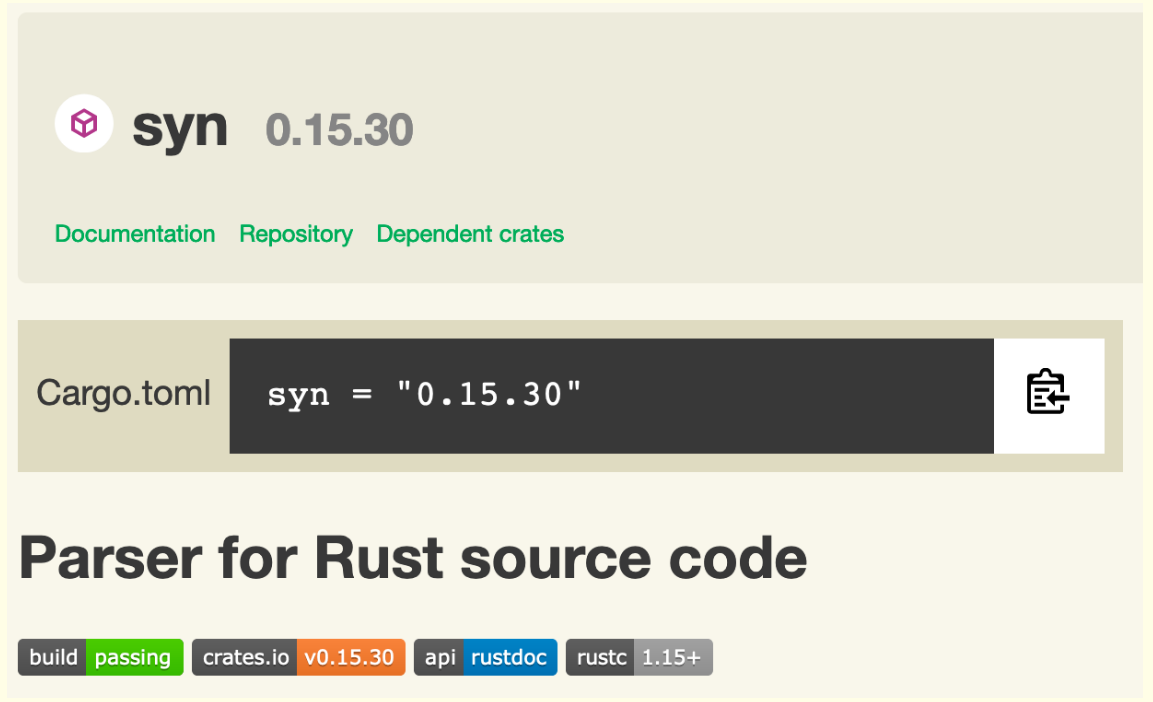
quote 0.6.11

Documentation Repository Dependent crates

Cargo.toml `quote = "0.6.11"`

Rust Quasi-Quoting

build passing crates.io v0.6.11 api rustdoc



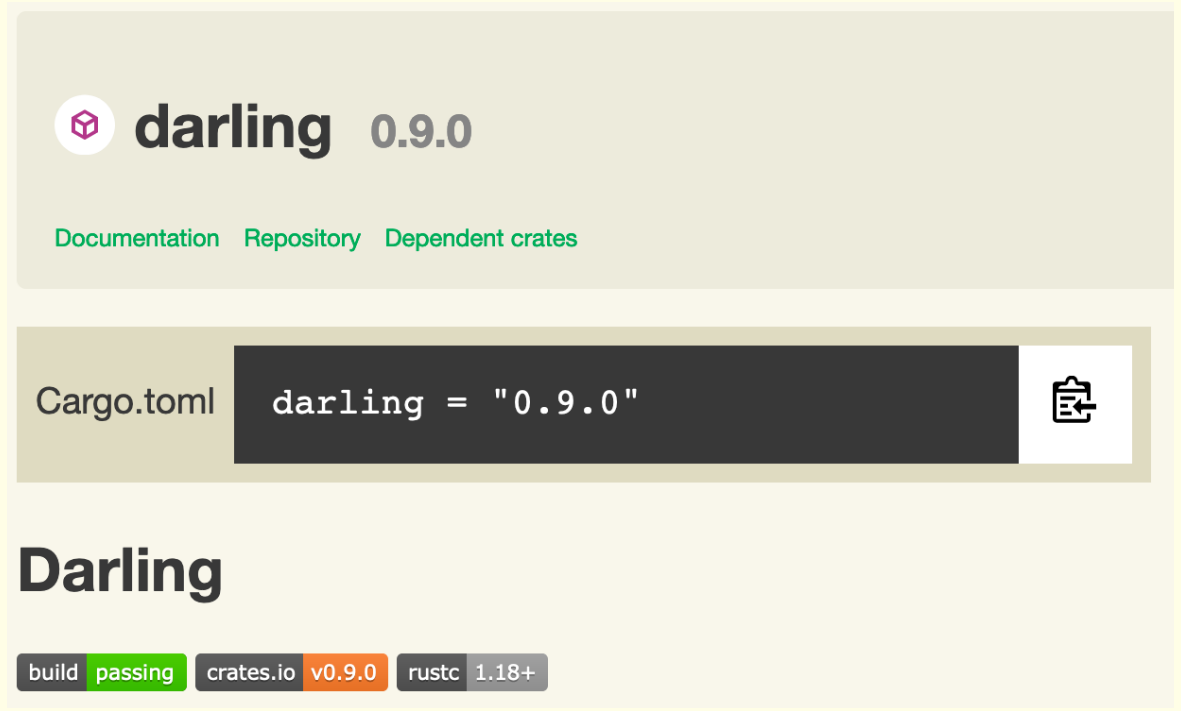
syn 0.15.30

Documentation Repository Dependent crates

Cargo.toml `syn = "0.15.30"`

Parser for Rust source code

build passing crates.io v0.15.30 api rustdoc rustc 1.15+



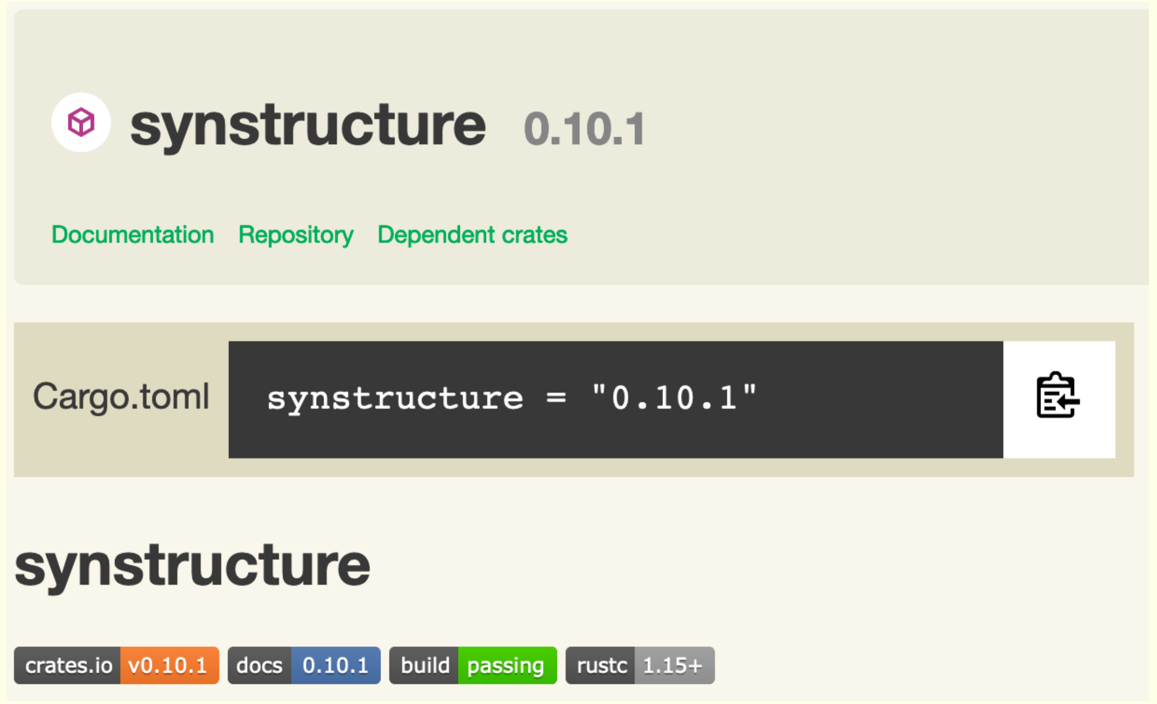
darling 0.9.0

Documentation Repository Dependent crates

Cargo.toml `darling = "0.9.0"`

Darling

build passing crates.io v0.9.0 rustc 1.18+



synstructure 0.10.1

Documentation Repository Dependent crates

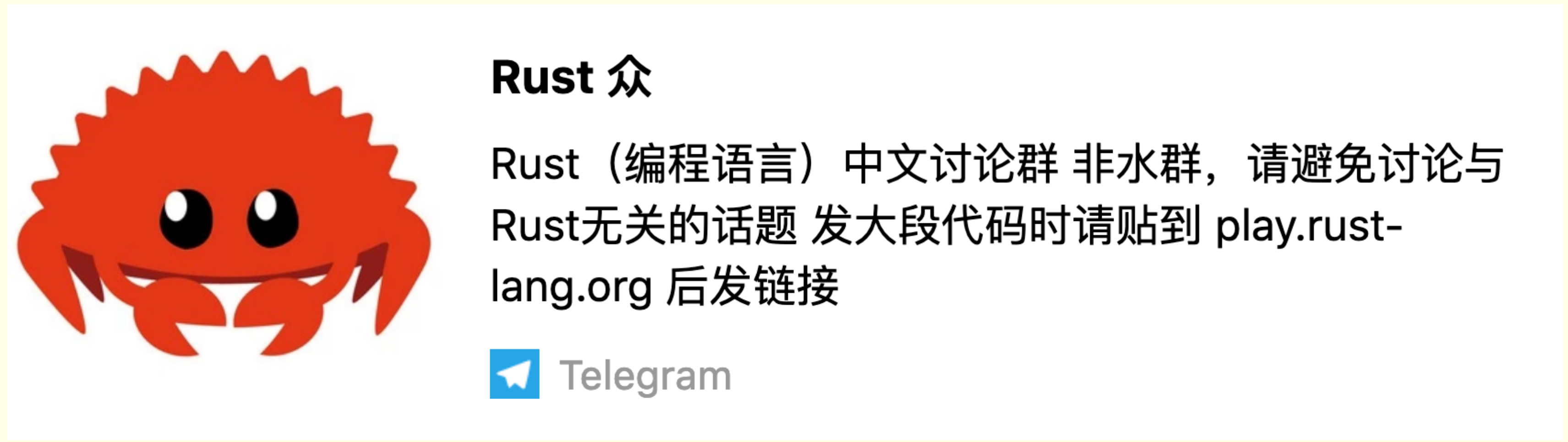
Cargo.toml `synstructure = "0.10.1"`

synstructure

crates.io v0.10.1 docs 0.10.1 build passing rustc 1.15+

Code will be available at:

https://github.com/upsuper/custom-derive-2019



Rust 众

Rust（编程语言）中文讨论群 非水群，请避免讨论与Rust无关的话题 发大段代码时请贴到 play.rust-lang.org 后发链接

Telegram

@rust_zh

https://t.me/rust_zh

Questions?

THANK YOU!