Procedural Macros - Expand and Implement

In Todays Talk

YOU ...

1. ...learn about three gems in the Rust Ecosystem.

In Todays Talk

YOU ...

1. ...learn about two gems in the Rust Ecosystem.
2. ...learn about `(Procedural) Macros` and how they make code **accessible** and **expressive**.

In Todays Talk

YOU ...

1. ...learn about two gems in the Rust Ecosystem.
2. ...learn about `(Procedural) Macros` and how they make code **accessible** and **expressive**.
3. ...write your first two Procedural Macros showing the concept of modern Macro Dev.
4. ...investigate how to parse in a more modular way.
5. ...get a lot of resources to dig deeper.

Who am I: Tim Janus!

- Studied Computer Science at TU Dortmund University.
- Started Learning Rust in 2021.
- Helping Clients as Rust Expert since 2023.
  - Performance and Security-critical Software Solutions.
  - Supporting Software Development Teams with Rust.
  - Instructor for Rust Workshops.
- The last 7 months: Worked with MM and YOU on the Rust Port for RediSearch.
- Meetup organizer, regular conference attendee, and broadly interested.

Chapter 1 - Gems in the Ecosystem

- Running Example `MyPoint`
- `Serde`
- `Clap`
- Compare Proc Macros with C Preprocessor Macros and C++ Templates.

Running Example - MyPoint

```rust
// missing often used derives: Hash, Eq, Ord
#[derive(Debug, Default, Clone, Copy, PartialEq, PartialOrd)]
pub struct MyPoint {
    x: f64,
    y: f64,
}
```

Before we dig deeper, explore the code...

Running Example - MyPoint

```rust
// missing often used derives: Hash, Eq, Ord
#[derive(Debug, Default, Clone, Copy, PartialEq, PartialOrd)]
pub struct MyPoint {
    x: f64,
    y: f64,
}
```

Before we dig deeper, explore the code...

... and open questions:

- Why are Hash, Eq and Ord Special?
- When not to derive?

Running Example - MyPoint

Why are Hash, Eq and Ord Special?

Many datatypes cannot easily derive them:

- `NaN` problem with floating point numbers

When not to derive?

Code Sizes matters

Gem No. 1 - Serde

By David Tolnay and many Contributors.

`Ser/De` - Easily add Serialization and Deserialization.

- Dervies Serialization and Deserialization Code.
- High Performance, no boilerplate.
- Many Formatters, e.g. json, protobuf, cbor, etc.

Gem No. 1 - Serde

By David Tolnay and many Contributors.

`Ser/De` - Easily add Serialization and Deserialization.

- Dervies Serialization and Deserialization Code.
- High Performance, no boilerplate.
- Many Formatters, e.g. json, protobuf, cbor, etc.

```
use serde::{Deserialize, Serialize};

#[derive(Debug, Default, Clone, Copy, PartialEq, PartialOrd, Serialize,
Deserialize)]
pub struct MyPoint {
    #[serde(rename = "x")]
    value_x: f64,
    #[serde(rename = "y")]
    value_y: f64,
}

fn main() {
    let point = MyPoint { value_x: 1.0, value_y: 2.0 };
    let serialized = serde_json::to_string(&point).unwrap();
    println!("MyPoint JSON: {}", serialized);

    let json = r#"{"x":42.0,"y":42.0}"#;
    let deserialized: MyPoint = serde_json::from_str(&json).unwrap();
    println!("MyPoint in-memory: {:?}", deserialized);
}
```

Gem No. 2 - Clap

By Kevin B. Knapp and many Contributors.

- `Clap` - Expressive Command Line Parsing without Boilerplate.
- Builder API.
- Derive Macro API.
- User gets Polished CLI experience.
- Flexibility to port CLIs to Clap.

Gem No. 2 - Clap

By Kevin B. Knapp and many Contributors.

- `Clap` - Expressive Command Line Parsing without Boilerplate.

```
Type 'help' for more information

> help
Available commands:
  add <x> <y>  - Add a new point with x and y coordinates
  list         - List all stored points
  quit         - Exit the application
  help         - Show this help message
> help add

Usage: mypoint add <X> <Y>

Arguments:
  <X>  X coordinate
  <Y>  Y coordinate

Options:
  -h, --help  Print help
```

Gem No. 2 - Clap

By Kevin B. Knapp and many Contributors.

- `Clap` - Expressive Command Line Parsing without Boilerplate.

```
#[derive(Parser)]
#[command(name = "mypoint")]
#[command(about = "A CLI for managing MyPoint objects")]
struct Cli {
    #[command(subcommand)]
    command: Commands,
}

#[derive(Subcommand)]
enum Commands {
    /// Add a new point with x and y coordinates
    Add {
        /// X coordinate
        x: f64,
        /// Y coordinate
        y: f64,
    },
    /// List all points
    List,
    /// Quit the application
    Quit,
}
```

Gem No. 2 - Clap

By Kevin B. Knapp and many Contributors.

- `Clap` - Expressive Command Line Parsing without Boilerplate.

Let's explore the code...

Compare C, C++ and Rust!

Category	C (Preprocessor Macros)	C++ (Templates)	Rust (Proc Macros)
When they run	Before compilation (preprocessor)	During compilation (template instantiation)	During compilation (syntax transformation)
What they see	Raw text / tokens	Types, templates, concepts	Token stream / AST-like structured syntax
Safety & Ergonomics	No type safety, brittle, unclear errors	Type-checked but complex errors	Type-checked after expansion, good tooling/errors

Chapter 2 - (Procedural) Macros, How?

1. Distinguish from Declarative Macros
2. What is a parser and an AST?
3. Definition: The three types of Macros.
4. Examples: The three types of Macros

Declarative Macros

Procedural Macros aren't declarative Macros (Pattern Matching).

```
let v = vec![1, 4, 7];
println!("{:?}", v);
```

- They are not distungishable for a caller from Procedural Function-like Macros.

Declarative Macros (Pattern Matching)

How are declarative Macros defined?

```
#[macro_export]
macro_rules! vec {
    () => (
        $crate::vec::Vec::new()
    );
    ($elem:expr; $n:expr) => (
        $crate::vec::from_elem($elem, $n)
    );
    ($($x:expr),+ $(,)?) => (
        <[_]>::into_vec(
            // Using the intrinsic produces a dramatic improvement in stack
usage for
            // unoptimized programs using this code path to construct large
Vecs.
            $crate::boxed::box_new([$($x),+])
        )
    );
}
```

- They are powerful, e.g. you can do a Lisp Parser
- They can help a lot with boilerplate!

What are Procedural Macros

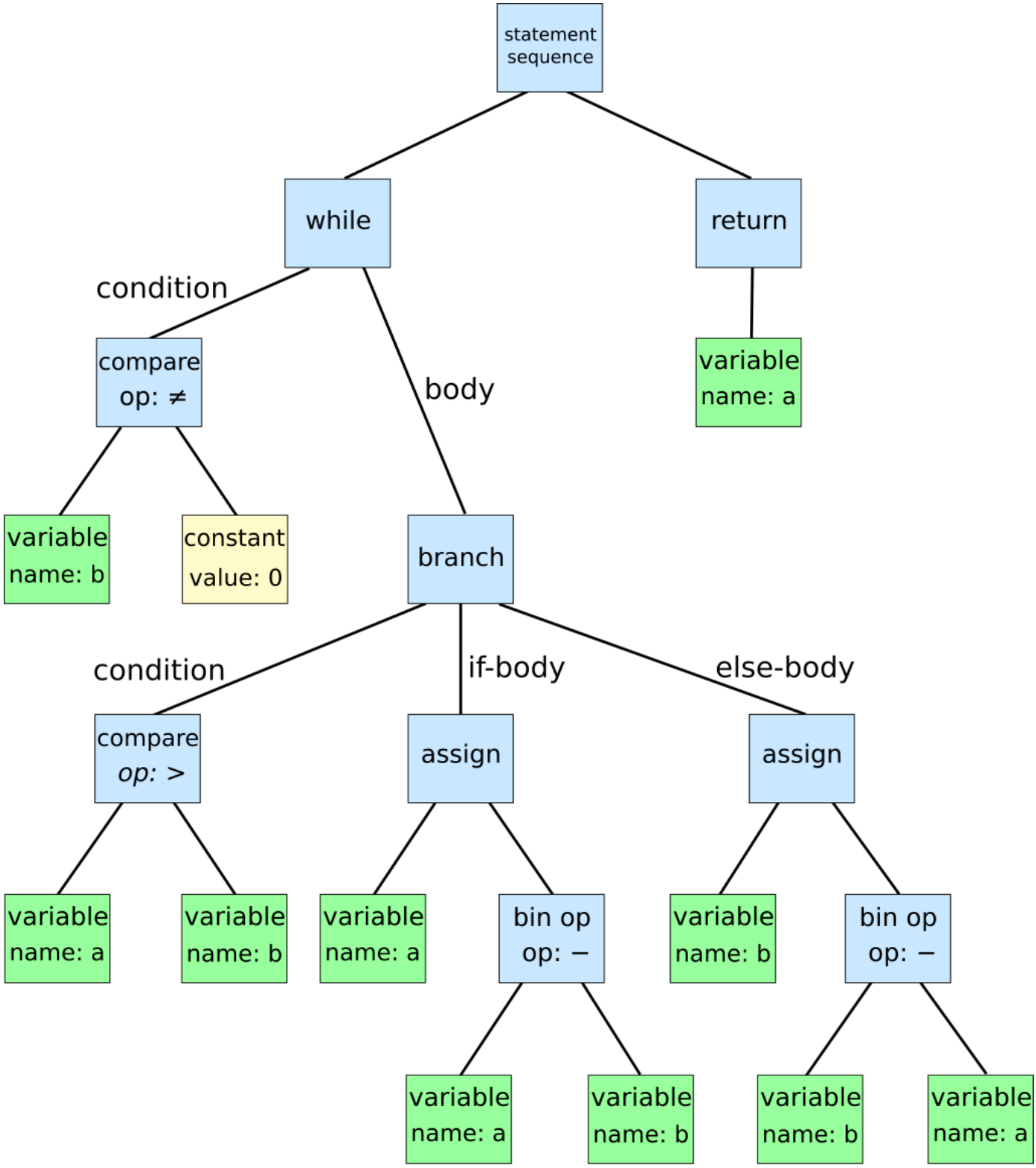
They are Parsers and Generators.

What's a Parser and how does it work on Rust Code?

Generates an AST (Abstract Syntax Tree) from Code or tokens.

```
while b != 0 {
    if a > b {
        a = a-b;
    } else {
        b = b-a;
    }
}
```

What is an AST?



Types of Procedural Macros

1. Function-like

Definition:

```
#[proc_macro]
pub fn foo(attr: TokenStream) -> TokenStream { ... }
```

Invocation:

```
foo!(Bar Baz Foo)
```

Types of Procedural Macros

2. Derive Macros

Definition:

```rust
#[proc_macro(derive(Bar))]
pub fn bar(body: TokenStream) -> TokenStream { ... }
```

Invocation:

```rust
#[derive(Bar)]
struct S;
```

Types of Procedural Macros

3. Attribute Macros

Defintion:

```rust
#[proc_macro_attribute]
pub fn baz(attr: TokenStream, item: TokenStream) -> TokenStream { ... }
```

Invocation:

```rust
#[baz(qux, quux)]
fn some_item() {}
```

Example - 1. Function-like Macros

Generates Rust-Code from the Macro Input. Looks like declarative Macro. Embeds JSON as DSL.

```
let john = json!({
    "name": "John Doe",
    "age": 43,
    "phones": [
        "+44 1234567",
        "+44 2345678"
    ]
});
```

Expands to:

```
let john = ::serde_json::Value::Object({
    let mut object = ::serde_json::Map::new();
    let _ = object
        .insert(("name").into(), ::serde_json::to_value(&"John Doe").unwrap());
    let _ = object.insert(("age").into(),
::serde_json::to_value(&43).unwrap());
    let _ = object
        .insert(
            ("phones").into(),
            ::serde_json::Value::Array(
                <[_]>::into_vec(
                    ::alloc::boxed::box_new([
                        ::serde_json::to_value(&"+44 1234567").unwrap(),
                        ::serde_json::to_value(&"+44 2345678").unwrap(),
                    ]),
                ),
            ),
        );
    object
});
```

Example - 2. Derive Macros

Generates Rust-Code from the definition of a type, e.g `struct` or `enum`.

```
#[derive(Debug)]
pub struct MyPoint {
    pub x: f64,
    pub y: f64,
}
```

Expands to

```
#[automatically_derived]
impl ::core::fmt::Debug for MyPoint {
    #[inline]
    fn fmt(&self, f: &mut ::core::fmt::Formatter) -> ::core::fmt::Result {
        ::core::fmt::Formatter::debug_struct_field2_finish(
            f,
            "MyPoint",
            "x",
            &self.x,
            "y",
            &&self.y,
        )
    }
}
```

Example - 3. Attribute Macros

```rust
#[tokio::main]
async fn main() {
    println!("hello world");
}
```

Expands to

```rust
fn main() {
    let body = async {
        {
            ::std::io::_print(format_args!("hello world\n"));
        };
    };
    #[allow(
        clippy::expect_used,
        clippy::diverging_sub_expression,
        clippy::needless_return,
        clippy::unwrap_in_result
    )]
    {
        return tokio::runtime::Builder::new_multi_thread()
            .enable_all()
            .build()
            .expect("Failed building the Runtime")
            .block_on(body);
    }
}
```

Chapter 3 - Drive your own Procedural Macros

- Tooling
- Easiest Macro: Greetings
- A derived Constructor
- Derive Validation Rules

Handy Tools for the Job

- Syn - is a parsing library for parsing a stream of Rust tokens
- Quote - provides the quote! macro for turning Rust syntax tree data structures into tokens of source code. (Introduces a a `proc_macro2` module, and you have to convert between `proc_macro2::TokenStream` and `proc_macro::TokenStream`)

`cargo expand` expands macros in source code, showing us what is really written.

Easiest Macro

```rust
#[derive(Debug, Greetings)]
pub struct MyPoint {
    pub x: f64,
    pub y: f64,
}

fn main() {
    let tmp = MyPoint { x: 1.0, y: 2.0 };
    tmp.greet();
}
```

Output:

```
Hello from MyPoint!
```

Greetings Implementation

```
#[proc_macro_derive(Greetings)]
pub fn derive_greetings(input: proc_macro::TokenStream) ->
proc_macro::TokenStream {
    let ast = parse_macro_input!(input as syn::DeriveInput);

    let name = &ast.ident;

    quote! {
        #[automatically_derived]
        impl #name {
            pub fn greet(&self) {
                println!("Hello from {}!", stringify!(#name))
            }
        }
    }.into()
}
```

We have to use a specialied crate, **let's look into the code...**

A Constructor

```
#[derive(Debug, Ctor)]
pub struct MyPoint {
    pub x: f64,
    pub y: f64,
}

fn main() {
    let tmp = MyPoint::ctor(1.0, 2.0);
}
```

Quote Iterators

For context:

```
  match &ast.data {
         syn::Data::Struct(data_struct) => {
     // ...
```

We generate iterators for later use:

```
 let params_in_sig = data_struct.fields.iter().map(|f| {
     let name = &f.ident;
     let ty = &f.ty;
     quote!(#name: #ty)
 });
```

Quote Variable Interpolation

Why are those Iterators useful?

Variable Interpolation with `#var`

- `#(#var)*` - no separators.
- `#(#var),*` - character, comma, before the asterisk is used as separator.
- `#(struct #var;)*` - Repetition can contain other tokens too.

Composite Quote for Constructor

Why are those Iterators useful?

Variable Interpolation with `#var`

- `#(#var)*` - no separators.
- `#(#var),*` - character, comma, before the asterisk is used as separator.
- `#(struct #var;)*` - Repetition can contain other tokens too.

```
quote::quote! {
    #[automatically_derived]
    impl #type_name {
        pub fn ctor(#(#params_in_sig),*) -> Self {
            Self { #(#ctor_assignments),* }
        }
    }
}
```

Error handling - Basics

We use a `Result<TokenStream, syn::Error>` in the parsing code:

```
#[proc_macro_derive(Ctor)]
pub fn derive(input: proc_macro::TokenStream) -> proc_macro::TokenStream {
    let ast = parse_macro_input!(input as syn::DeriveInput);
    match ctor::derive_with_error_checks(&ast) {
        Ok(tokens) => tokens.into(),
        Err(err) => err.to_compile_error().into(),
    }
}
```

- The called code may use `syn::Error::new_spanned(token, msg)`.
- The token holds information on the error location.

Error handling - Example

```
fn check_right_type_struct(data_struct: &DataStruct) -> Result<(), syn::Error>
{
    match data_struct.fields {
        syn::Fields::Named(_) => { /* noop */}
        syn::Fields::Unnamed(_) => return Err(syn::Error::new_spanned(
            data_struct.struct_token,
            "Ctor cannot be derived for tuple structs, yet",
        )),
        syn::Fields::Unit =>
        return Err(syn::Error::new_spanned(
            data_struct.struct_token,
            "Ctor cannot be derived for unit structs, use Default instead",
        )),
    }
    Ok(())
}
```

Well, let's force them in the code...

The Use-Case: A distillation column

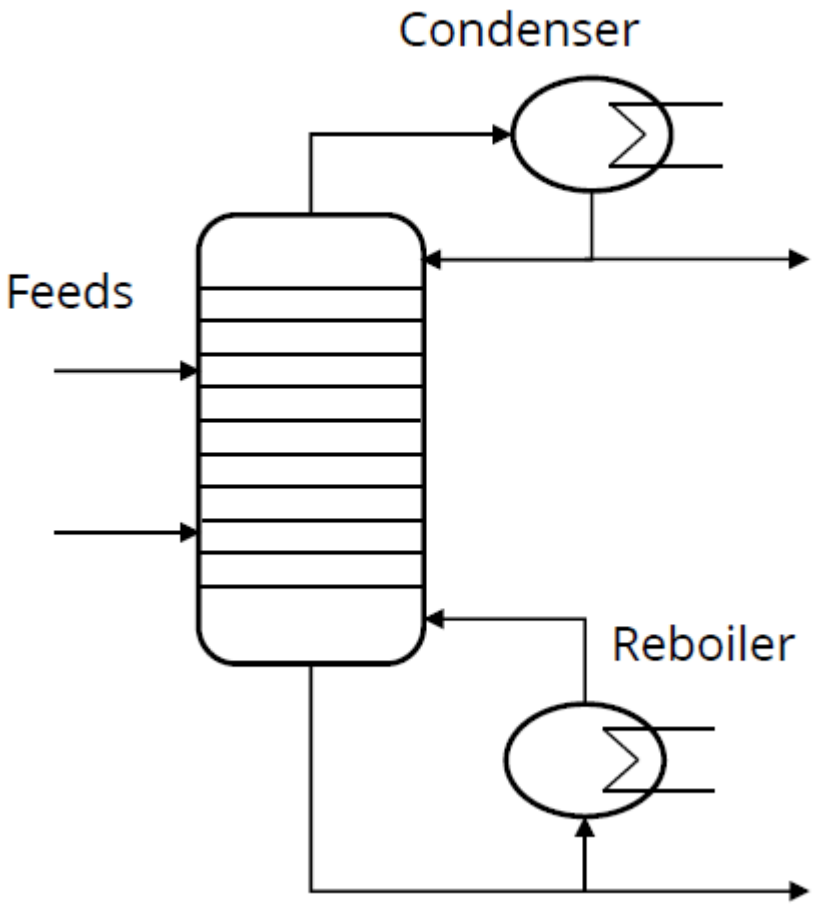
- EngCon - My old learning project for Procedural Macros

```rust
#[derive(Debug, Clone, Default, Copy, PartialEq, Validatable)]
pub struct DistillationColumn {
    #[validate_value(x >= 3)]
    pub trays: i32,

    #[validate_value(x < trays, x >= 1)]
    pub feed_place: i32,

    #[validate_value(x > 0.0)]
    pub reflux_ratio: f32,

    #[validate_value(x > 0.0, x < 1.0)]
    pub distiliate_to_feed_ratio: f32,
}
```



The Parser Trait

```rust
pub trait Parse: Sized {
    // Required method
    fn parse(input: ParseStream<'_>) -> Result<Self>;
}
```

Parsing interface implemented by all types that can be parsed in a default way from a token stream.

The Parser Trait Impl

```
pub trait Parse: Sized {
    // Required method
    fn parse(input: ParseStream<'_>) -> Result<Self>;
}


impl Parse for ValidationRule {
    fn parse(input: syn::parse::ParseStream) -> syn::Result<Self> {
        let mut candidate = ValidationRule {
            left: input.parse()?,
            cmp_op: input.parse()?,
            rigth: input.parse()?,
            right_is_field_on_self: false,
        };
        // ...
        if !matches!(
            candidate.cmp_op,
            syn::BinOp::Ge(_) | syn::BinOp::Gt(_) | syn::BinOp::Le(_) |
syn::BinOp::Lt(_)
        ) {
            // operator must be `<`, `<=`, `>=` or `>`
            return Err(syn::Error::new_spanned(
                candidate.cmp_op,
                "Only <, <=, >= and > are supported operators",
            ));
        }
        // ...
        Ok((candidate))
    }
}
```

Many Parser Traits Impls

- impl Parse for BinOp
- impl Parse for CapturedParam
- impl Parse for Expr
- impl Parse for FnArg
- impl Parse for ForeignItem
- impl Parse for GenericArgument
- impl Parse for GenericParam
- impl Parse for ImplItem
- impl Parse for Item
- impl Parse for Lit
- impl Parse for Member
- impl Parse for Meta
- impl Parse for PointerMutability
- impl Parse for RangeLimits
- impl Parse for ReturnType
- ...

Several hundred types

Further Sources

- A more depth Intro - Ciara gave a talk and it's already on YouTube.
- David Tolnays Workshop - A project-based workshop to learn Procedural Macros.

Learning begins the moment you stop watching and start building.

Thank - You!

It was a pleasure working with all of you! I l've learned a lot and truly enjoyed contributing to this project.

Let's stay in touch - and best of success for what comes next!

All that said

I'm happy to answer your questions...