Macros in Elixir: Responsible Code Generation

Lizzie Paquette

Brex

B2B financial Products
- Card & Cash

Systems Team
- Libraries & Frameworks

Overview

- Goals
- Basics
- Examples
  - Struct Definition: bdefstruct
  - Schema Definition: DataSchema
  - Macro Definition: Meta Macro
- Review
- Questions

Goals

Use Macros to:

- Reduce boilerplate
- Enforce best practices
- Improve reliability

While keeping code:

- Readable
- Maintainable
- Idiomatic

Goals

Use Macros to:

- Reduce boilerplate
- Enforce best practices
- Improve reliability

While keeping code:

- Readable
- Maintainable
- Idiomatic

Basics

What is a Macro?

- A macro is a transformation from AST → AST that is evaluated (expanded) at compile time.
- “Code that writes code”

Basics

Macros are all around us

- unless
- if
- defstruct
- |>

Basics

Macros are all around us

- unless
- if
- defstruct
- |>
- ...

Basics

What is an AST?

- Abstract Syntax Tree
- Elixir -> AST -> Bytecode
- First Class Value
- Structure: {operator, metadata, argument list}
  - Type: Macro.t()

Basics

What is an AST?

```
2 + 3
```

```
{:+,
    [context: Elixir, import: Kernel],
    [2, 3]
}
```

Basics

What is an AST?

```
defmodule M do
  def add(x, y) do
    x + y
  end
end
```

```
{:defmodule, _meta,
[{:__aliases__, _meta, [:M]},
  [do: {:def, _meta,
      [{:add, _meta,
          [{:x, _meta, Elixir}, {:y, _meta, Elixir}]},
        [do: {:+, _meta,
            [{:x, _meta, Elixir}, {:y, _meta, Elixir}]}
        ]
      ]}
  ]
]}
```

Basics

Anatomy of a Macro

```
defmacro my_macro(_args, [do: _block] // []) do
  <code_evaluted_at_compile_time>
  quote do
    <injected_into_callers_context>
  end
end
```

Basics

Quoted Values & Macro Expansion

- Recursive
- Outside in
- Macro.expand_once

```
if not (0 == 1) do
  "Hello"
end
```

Basics

Anatomy of a Macro

```
defmacro my_macro(_args, [do: _block] // []) do
  <code_evaluted_at_compile_time>
  quote do
    <injected_into_callers_context>
  end
end
```

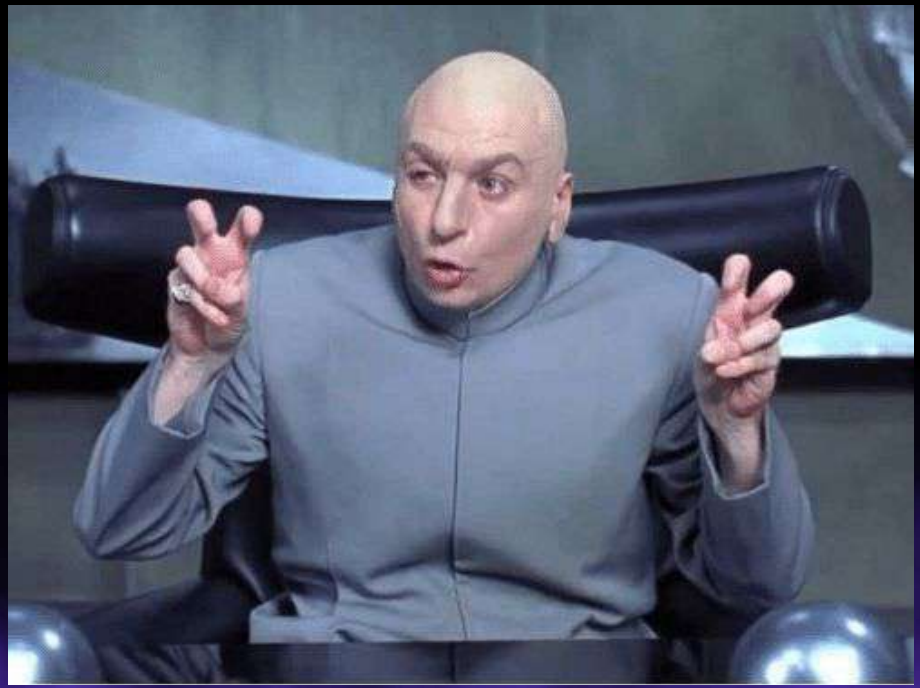
Could be anything!

Unchecked!

Basics

Quoted Values

- macros
  - AST -> AST
- quote/2
  - regular code -> AST
- unquote/1
  - AST -> regular code



Basics

Quoted Values

```
defmacro double(x) do
 quote do
   unquote(x) + unquote(x)
 end
end
```

Value	Code | AST
`x`	AST
`unquote(x)`	Code
`unquote(x) + unquote(x)`	Code

`quote do`
`  unquote(x) + unquote(x)`
`end`
	AST

Basics

Quoted Values

```
defmacro double(x) do
 quote do
   unquote(x) + unquote(x)
 end
end
```

```
>>> double(IO.inspect(4, label: "Side Effect"))
..> Side Effect: 4
..> Side Effect: 4
..> 8
```

Basics

Quoted Values

```
defmacro double(x) do
 quote do
   z = unquote(x)
   z + z
 end
end
```

```
>>> double(IO.inspect(4, label: "Side Effect"))
..> Side Effect: 4
..> 8
```

Basics

Quoted Values

```
defmacro double(x) do
 quote bind_quoted: [z: x] do
   z + z
 end
end
```

```
>>> double(IO.inspect(4, label: "Side Effect"))
..> Side Effect: 4
..> 8
```

Basics

Macro Expansion

- Recursive
- Outside in
- Macro.expand_once

```
unless 0 == 1 do
  "Hello"
end
```

Basics

Macro Expansion

- Recursive
- Outside in
- Macro.expand_once

```
case not (0 == 1) do
  x when x in [false, nil] -> nil
  _ -> "Hello"
end
```

Basics

Contexts

- Macro vs. Caller

```
defmodule MyMod do
 defmacro who_am_i() do
   quote do
     end
   end
 end
```

Macro Context

Caller's Context

```
defmodule MyCaller do

end
```

Caller's Context

Basics

Contexts

- Macro vs. Caller

```elixir
defmodule MyMod do
 defmacro who_am_i() do
   IO.inspect(__MODULE__, label: "Macro Context")

   quote do
     IO.inspect(__MODULE__, label: "Caller Context")


   end
 end
end
```

```elixir
defmodule MyCaller do
 require MyMod
 MyMod.who_am_i()

end
```

```
...> Macro Context: MyMod
...> Caller Context: MyCaller
```

Basics

Contexts

- Macro vs. Caller

```
defmodule MyMod do
 defmacro who_am_i() do
   IO.inspect(__MODULE__, label: "Macro Context")

   quote do
     IO.inspect(__MODULE__, label: "Caller Context")
     IO.inspect(unquote(__MODULE__), label:
       "Value from Macro Context")
   end
 end
end
```

```
defmodule MyCaller do
 require MyMod
 MyMod.who_am_i()

end
```

```
...> Macro Context: MyMod
...> Caller Context: MyCaller
...> Value from Macro Context: MyMod
```

Basics

Contexts

- Macro vs. Caller

```
defmodule MyMod do
 defmacro who_am_i() do
   IO.inspect(__MODULE__, label: "Macro Context")
   IO.inspect(__CALLER__.module, label: "Caller Env")
   quote do
     IO.inspect(__MODULE__, label: "Caller Context")
     IO.inspect(unquote(__MODULE__), label:
       "Value from Macro Context")
   end
 end
end
```

[Macro.Env](#)

A struct that holds compile time environment information.

The current environment can be accessed at any time as `__ENV__/0`. Inside macros, the caller environment can be accessed as `__CALLER__/0`.

```
>
...> Caller Env: MyCaller
```

Basics

Hygiene

- Hygiene - variables, imports, aliases defined in a macro do not leak into the caller's own definitions.
- var!

```
defmacro set(val) do
 quote do
   v = unquote(val)
 end
end

>>> v = 2
>>> set(3)
>>> v
..> 2
```

Basics

Hygiene

- Hygiene - variables, imports, aliases defined in a macro do not leak into the caller's own definitions.
- var!

```
defmacro override(val) do
 quote do
   var!(v) = unquote(val)
 end
end
```

```
>>> v = 2
>>> override(3)
>>> v
..> 3
```

Basics

Special Cases

- Kernel.SpecialForms

```elixir
use MyModule
```

```elixir
require MyModule
MyModule.__using__()
```

```elixir
defmodule MyModule do
  defmacro __using__(args, opts) do
    …
  end
end
```

Goals

Use Macros to:

- Reduce boilerplate
- Enforce best practices
- Improve performance/reliability

While keeping code:

- Readable
- Maintainable
- Idiomatic

Example: Struct Definition

```
defmodule User do
 defstruct [:id, :account, :first_name, :last_name, :email, :password, :status]
end
```

Example: Struct Definition

```
defmodule User do
 @enforce_keys [:id, :first_name, :last_name, :account, :password]

 defstruct [ :id, :account, :first_name, :last_name, :email, {:password, "1234"}, {:status,
:active}]
end
```

Example: Struct Definition

```elixir
defmodule User do
@typedoc """
   The password is 1234 by default.
   The status is a required value can be
:active or :disabled.
   The email must be a valid gmail address.
 """
  @type t :: %{
   required(:__struct__) => atom(),
   required(:id) => integer(),
   required(:account) => String.t,
   required(:first_name) => String.t,
   required(:last_name) => String.t,
   optional(:email) => String.t | nil,
   optional(:password) => String.t,
   optional(:status) => atom()
  }
```

```elixir
@enforce_keys [:id, :first_name, :last_name,
:account, :password]

 defstruct [ :id, :account, :first_name,
:last_name,:email, {:password, "1234"}, {:status,
:active}]
end
```

Example: Struct Definition

```elixir
defmodule User do
@typedoc """
   The password is 1234 by default.
   The status is a required value can be
:active or :disabled.
   The email must be a valid gmail address.
 """
  @type t :: %{
   required(:__struct__) => atom(),
   required(:id) => integer(),
   required(:account) => String.t,
   required(:first_name) => String.t,
   required(:last_name) => String.t,
   optional(:email) => String.t | nil,
   optional(:password) => String.t,
   optional(:status) => atom()
  }
```

```elixir
@enforce_keys [:id, :first_name, :last_name,
:account, :password]

 defstruct [ :id, :account, :first_name,
:last_name,:email, {:password, "1234"}, {:status,
:active}]
end
```

Example: Struct Definition

How can we make this better?

- Reduce Boilerplate
  - Reference each field only once
- Best Practices
  - Make fields required by default
  - If optional have to explicitly state a default value
- Improve Reliability
  - Validate at compile time

Example: Struct Definition

Do we need a Macro?

- Can this be done with behaviors or protocols? ✖
- Can this be done with higher order functions? ✖
- Generally: Can functions can produce a struct definition? ✖

We need a macro.

Example: Struct Definition

Final Result

- DSL
  - domain-specific language
- Field attributes

```
bdefstruct do
   field :id, integer
   field :account, Account.t()
   field :first_name, String.t()
   field :last_name, String.t()
   @fielddoc "must be a valid gmail address"
   field :email, String.t(), nil
   field :password, String.t, "1234"
   @fielddoc "values can be :active | :disabled"
   field :status, atom(), :active
 end
```

Example: Struct Definition

Behind the Scenes

- Parsing
- Codegen
- Injection

```
defmodule Brex.Struct do
 defmacro bdefstruct([], do: block) do
   fields = Parser.parse_body(block, parsing_env)

   type_spec_ast = {:%, [], [{:__MODULE__, [], Elixir},
        {:%{}, [], fields |> Enum.map(fn x -> {x.name, x.type} end)}]}

   required_fields = fields |> Enum.filter(&Map.has_key?(&1, :required)) |>
Enum.map(fn %{name: name} -> name end)

   struct_ast_fields = fields |> Enum.map(fn
       %{name: name, required: nil} -> name
       %{name: name, optional: default} -> {name, default}
     end)

   quote do
     @typedoc unquote(generate_doc_string(fields))
     @type t :: unquote(type_spec_ast)
     @enforce_keys unquote(required_fields)
     defstruct unquote(struct_ast_fields)
   end
 end
```

Example: Struct Definition

Behind the Scenes

- Parsing
- Codegen
- Injection

```
defmodule Brex.Struct do
 defmacro bdefstruct([], do: block) do
   fields = Parser.parse_body(block, parsing_env)

   type_spec_ast = {:%, [], [{:__MODULE__, [], Elixir},
        {:%{}, [], fields |> Enum.map(fn x -> {x.name, x.type} end)}]}

   required_fields = fields |> Enum.filter(&Map.has_key?(&1, :required)) |>
Enum.map(fn %{name: name} -> name end)

   struct_ast_fields = fields |> Enum.map(fn
       %{name: name, required: nil} -> name
       %{name: name, optional: default} -> {name, default}
     end)

   quote do
     @typedoc unquote(generate_doc_string(fields))
     @type t :: unquote(type_spec_ast)
     @enforce_keys unquote(required_fields)
     defstruct unquote(struct_ast_fields)
   end
 end
```

Example: Struct Definition

Parsing

- Tree recursion
- No macros!

```
defmodule Parser do
 def parse_body({:__block__, _, lines}, parsing_env), do: parse_lines(lines, parsing_env)
 def parse_body(nil, parsing_env), do: parse_lines([], parsing_env)
 def parse_body(line, parsing_env), do: parse_lines([line], parsing_env)

 defp parse_lines(lines, parsing_env) do ...

 defp parse_line({:@, _, [{attribute_name, _, attribute_values}]},
   %{field_attributes: attributes}) do
 defp parse_line({field_marker, _, args}, %{field_keywords: keywords}) do ...

 defp verify_options(given, expected, name) do ..

 defp process_errors({field, error}, expected, name) do ...
end
```

Example: Struct Definition

Parsing

- Static validation
  - Exceptions -> compile time errors
  - Optimal to check argument types
- Debuggability
  - Line numbers inaccurate
  - Clear error messages

```elixir
defp verify_options(given, expected, name) do
  case Optimal.validate(given, Optimal.schema(opts: expected)) do
    {:ok, opts} -> opts
    {:error, errors} -> raise ArgumentError,
       message: Enum.map(&process_errors(&1, expected, name))
  end
end

defp process_errors({field, error}, valid_options, name) do
  cond do
    String.contains?(error, "no extra keys") ->
      "Invalid #{name}: #{field} is not allowed, only #{valid_options}."
    String.contains?(error, "must be of type nil") ->
       "Invalid #{name}: #{field} takes no arguments"
    true -> "Invalid #{name}: #{field} - #{error}"
  end
end
```

Example: Struct Definition

Codegen

- Parser :: AST -> Map
- Codegen :: Map -> AST

```
defp codegen_struct(fields) do
  struct_ast_fields =
    fields
    |> Enum.map(fn
      %{name: name, required: nil} -> name
      %{name: name, optional: default} -> {name, default}
    end)

  quote do
    defstruct unquote(struct_ast_fields)
  end
end
```

Example: Struct Definition

Injection

- Keep small
- Quoted code is unchecked
  - Don’t write AST values by hand
  - Check with Macro.validate, Code.eval_quoted, Macro.to_string

```
quote do
  @typedoc unquote(doc_string(fields))
  @type t :: unquote(type_spec_ast)
  @enforce_keys unquote(required_fields)
  defstruct unquote(struct_ast_fields)
end
```

Intermediate Example: Schema Definition

Similar Issue

- 10 field schema would have 100 lines

```elixir
defmodule User.Analysis do
  use Ecto.Schema

  schema "user_analysis" do
    field :unique_token, :string
    field :is_dry_run, :boolean
    field :reason, :string
    field :count, :integer
    field :status, Status
    field :config, Config
    embeds_one :result_payload, X.Data.ResultPayload
    belongs_to :account, X.Data.Account
    timestamps(type: :utc_datetime_usec)
  end

  @fields [:unique_token, :is_dry_run, :reason, :count,
:status, :config, :result_payload, :account_id]
  @required_fields [:unique_token, :is_dry_run, :count,
:status, :config, :account_id]

  def changeset(struct, params \\ %{}) do
    struct
    |> raw_changeset(params)
    |> validate_immutable([:unique_token, :is_dry_run,
:reason, :account_id], :error)
  end

  def raw_changeset(struct, params \\ %{}) do
    struct
    |> base_partial_cast(params, @fields)
    |> validate_required(@required_fields)
    |> validate_uniqueness(unique_token: [])
    |> cast_embedded([:result_payload])
    |> cast_poly(:config)
    |> validate_params()
  end

  def validate_params(changeset) do
    changeset
  end

  defoverridable(validate_params: 1)
end
```

Intermediate Example: Schema Definition

Similar Issue

- 10 field schema would have 100 lines

Intermediate Example: Schema Definition

How can we make this better?

- Reduce Boilerplate
  - Automatically generate changeset validations
- Best Practices:
  - Make fields are immutable & required by default
- Improve performance/reliability
  - Compile time validations

Intermediate Example: Schema Definition

Success Metrics

- 50% lines of code reduction
- 100% new schemas written using this construct
- X questions asked (Not measured but felt)
    - How much of the api was unintuitive and unclear?

Intermediate Example: Schema Definition

```
data_schema(table_name: "user_analysis", id_prefix: "userana") do
  @unique
  field :unique_token, :string
  field :is_dry_run, :boolean
  @optional
  field :reason, :string
  @mutable
  field :count, :integer
  @mutable
  enum :status, do: :running | :finished | :error
  poly :config, do: X.Data.TypeAConfig | X.Data.TypeBConfig
  @optional
  @mutable
  embeds_one :result_payload, X.Data.ResultPayload
  belongs_to :account, X.Data.Account
end
```

Intermediate Example: Schema Definition

Contracts & Documentation

- Make what you're providing clear

```
defmodule Brex.DataSchema.Contract do
 @type changeset :: Ecto.Changeset.t()
 @callback changeset(struct, params :: map) :: changeset
 @callback raw_changeset(struct, params :: map) :: changeset
 @callback validate_params(changeset) :: changeset
end
```

Intermediate Example: Schema Definition

Escape Hatches

- Allow the developer to customize their experience

```elixir
def changeset(struct, params}) do
  struct
  |> raw_changeset(params)
  |> validate_immutable(
    immutable_fields)
  |> validate_params()
end
```

```elixir
def raw_changeset(struct, params) do
 basic_validations(struct, params, @fields)
end

defoverridable(validate_params: 1)
def validate_params(changeset), do: changeset
```

Intermediate Example: Schema Definition

Reflections

- Useful for developers to see what's happening
  - debugging
- Convention: underscored functions
- Macro.escape

```
@callback __fields__() :: map
def __fields__() do
  unquote(Macro.escape(parsed_body))
end

@callback __macro_arguments__() :: map
def __macro_arguments__() do
  unquote(Macro.escape(macro_arguments))
end
```

Intermediate Example: Schema Definition

Reflections

```
defp execute(mod) do
  Mix.Project.compile([mod])
  fields = mod.__fields__()
  table_name = Map.fetch!(mod.__macro_arguments__(), :table_name)
  generated_code = [ codegen_create_table(fields, table_name), codegen_unique_index(fields,
    table_name), codegen_indexes(fields, table_name)]
  :ok = Macro.validate(generated_code)
  change_contents = generated_code |> Macro.to_string() |> Code.format_string!()
  Mix.Tasks.Ecto.Gen.Migration.run(["create_" <> to_snake(mod), "--change", change_contents])
end
```

Intermediate Example: Schema Definition

Reflections

```
defp execute(mod) do
  Mix.Project.compile([mod])
  fields = mod.__fields__()
  table_name = Map.fetch!(mod.__macro_arguments__(), :table_name)
  generated_code = [ codegen_create_table(fields, table_name), codegen_unique_index(fields,
    table_name), codegen_indexes(fields, table_name)]
  :ok = Macro.validate(generated_code)
  change_contents = generated_code |> Macro.to_string() |> Code.format_string!()
  Mix.Tasks.Ecto.Gen.Migration.run(["create_" <> to_snake(mod), "--change", change_contents])
end
```

```
>>> mix brex.data_schema.gen.migration Users
```

```
data_schema(table_name: "users") do
 @unique
 field :id, :integer
 belongs_to :account, Account
 field :first_name, :string
 field :last_name, :string
 @optional
 @mutable
 field :email, :string
 @optional
 @mutable
 field :password, :string
 @mutable
 enum :status, do: :active | :disabled
end
```

```
defmodule Brex.Migrations.Users do
 use Ecto.Migration

 def change do
   create(table(:users, primary_key: false)) do
     add(:id, :integer, primary_key: true)
     add(:account_id,
      references(:accounts, type: :string), null: false)
     add(:first_name, :text, null: false)
     add(:last_name, :text, null: false)
     add(:email, :text, [])
     add(:password, :text, [])
     add(:status, :string, null: false)
     timestamps(type: :utc_datetime_usec)
   end

   create(index(:users, [:account_id]))
   create(index(:users, [:status]))
 end
end
```

DRY it out

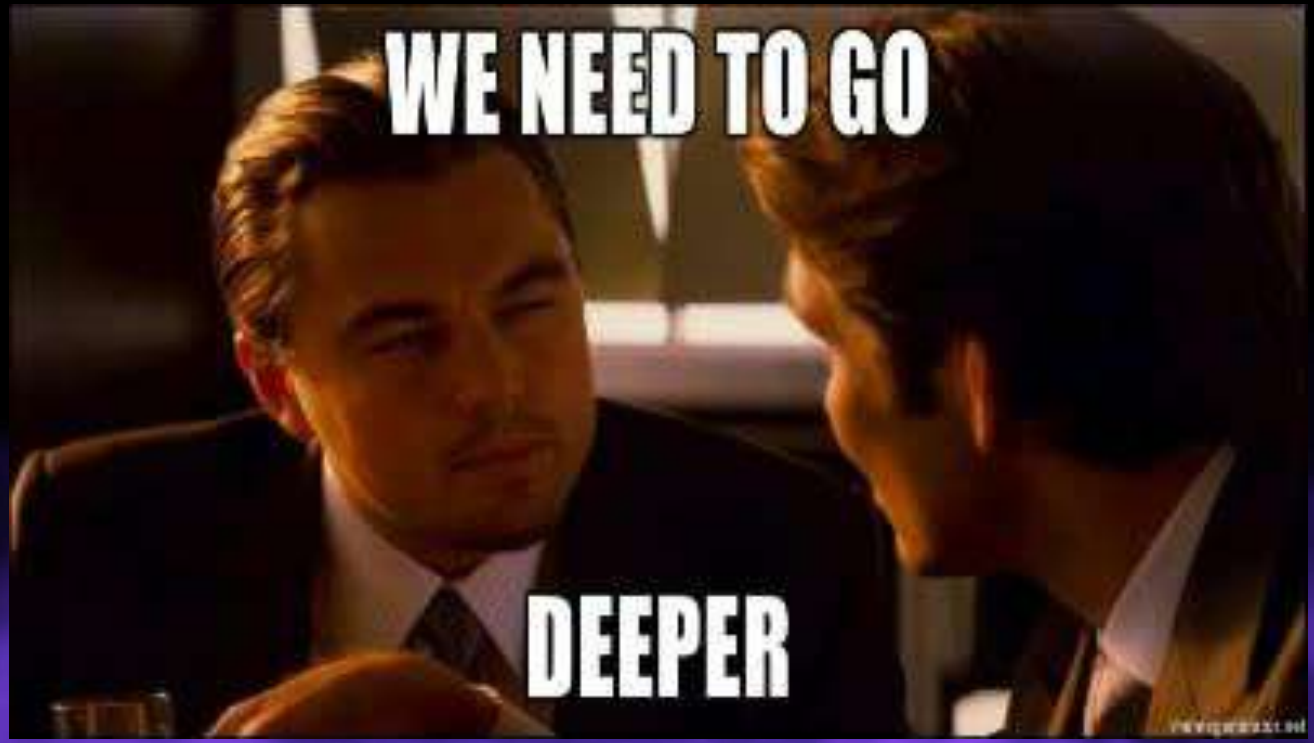
```
bdefstruct do
 field :id, integer
 field :account, Account.t()
 field :first_name, String.t()
 field :last_name, String.t()
 @fielddoc "must be a valid gmail address"
 field :email, String.t(), nil
 field :password, String.t, "1234"
 @fielddoc "values can be :active |
:disabled"
 field :status, atom(), :active
end
```

```
data_schema(table_name: "users") do
 @unique
 field :id, :integer
 belongs_to :account, Account
 field :first_name, :string
 field :last_name, :string
 @optional
 @mutable
 field :email, :string
 @optional
 @mutable
 field :password, :string
 @mutable
 enum :status, do: :active | :disabled
end
```

DRY it out

```
macro_name(macro_arguments :: Keyword.t()) do
 @field_attribute possible_arguments
 @field_attribute possible_arguments
 field_keyword :: atom, possible_arguments
end
```

DRY it out



DRY it out

We need a Meta Macro!

DRY it out

Advanced Example: Macro Definition

Meta Macro Benefits

- Now easy to define other schema like objects
- Each becomes easily extendable
    - Eg. Add attribute to specify which a field is json encodable
- Brex.Struct and Brex.DataSchema become macroless!

Advanced Example: Macro Definition

Input Contract

```
@doc "Optional hook into parser."
@callback process_fields(map) :: map

@doc "Returns the AST that will be injected."
@callback codegen(macro_arguments :: map, fields :: [map]) :: Macro.t()

@macro_name
@macro_arguments
@field_keywords
@field_attributes
```

Advanced Example: Macro Definition

```
@macro_name :bdefstruct
@macro_arguments []
@field_keywords [field: {:list, :any}]
@field_attributes [fielddoc: {:list, :string}]
```

```
@macro_name :data_schema
@macro_arguments [error_level: {:enum, [:warn,
:error]}, id_prefix: :string, table_name: :string,
embedded: :boolean, primary_key: :any, timestamps:
:boolean]

@field_keywords [belongs_to: {:list, :any},
embeds_one: {:list, :any}, embeds_many: {:list,
:any}, field: {:list, :any}, has_one: {:list,
:any}, has_many: {:list, :any}, many_to_many:
{:list, :any}, poly: {:list, :any}, enum: {:list,
:any}]

@field_attributes [mutable: nil, unique:
:any,optional: nil, fielddoc: {:list, :string}]
```

Advanced Example: Macro Definition

Output Contract

```
@callback __fields__() :: map
@callback __macro_arguments__() :: map
@callback unquote(macro_name)(opts :: Keyword.t()) :: Macro.t
```

Advanced Example: Macro Definition

Module Attributes

Elixir Getting Started: Module Attributes

Module attributes in Elixir serve three purposes:

- They serve to annotate the module, often with information to be used by the user or the VM.
- They work as constants.
- They work as a temporary module storage to be used during compilation.

Advanced Example: Macro Definition

Module Attributes

Elixir Getting Started: Module Attributes

Module attributes in Elixir serve three purposes:

- They serve to annotate the module, often with information to be used by the user or the VM.
- They work as constants.
- They work as a temporary module storage to be used during compilation.

Advanced Example: Macro Definition

Module Attributes

- To access their module attributes we must read them Brex.Struct and Brex.DataSchema before they are compiled.
- Compilation order is nondeterministic
  - Race Conditions
- Need a way to force compilation order

Module

> After a module is compiled, using many of the functions in this module will raise errors, since it is out of their scope to inspect runtime data.

Advanced Example: Macro Definition

Compile Time Hooks

- before_compile
- after_compile
- Watch out for deadlock!

```
defmodule MetaMacro do
  defmacro __using__(_opts) do
    quote do
      # unquote(__MODULE__) = MetaMacro
      @behaviour unquote(__MODULE__)
      @before_compile unquote(__MODULE__)
    end
  end
```

```
defmacro __before_compile__(env) do
 # env.module = Brex.DataSchema
 macro_name = Module.get_attribute(env.module, :macro_name)
 ...
 quote do
   defmacro unquote(macro_name)(args \\ [], do: block) do
    <body>
   end
 end
end
```

Advanced Example: Meta Macro

How do we know it works?

- “Test your generated code not your code generation.”
  - Chris McCord, Metaprogramming Expert

Revisit Goals:

Use Macros to:

- Reduce boilerplate ✓
- Enforce best practices ✓
- Improve reliability ✓

While keeping code:

- Readable ✓
- Maintainable ✓
- Idiomatic ✓

50% less code

Required & Immutable by default, auto generated validations

Compile time checks & optimizations

Thoughtful & well documented DSL

Extendable schema declaration, most code non macro

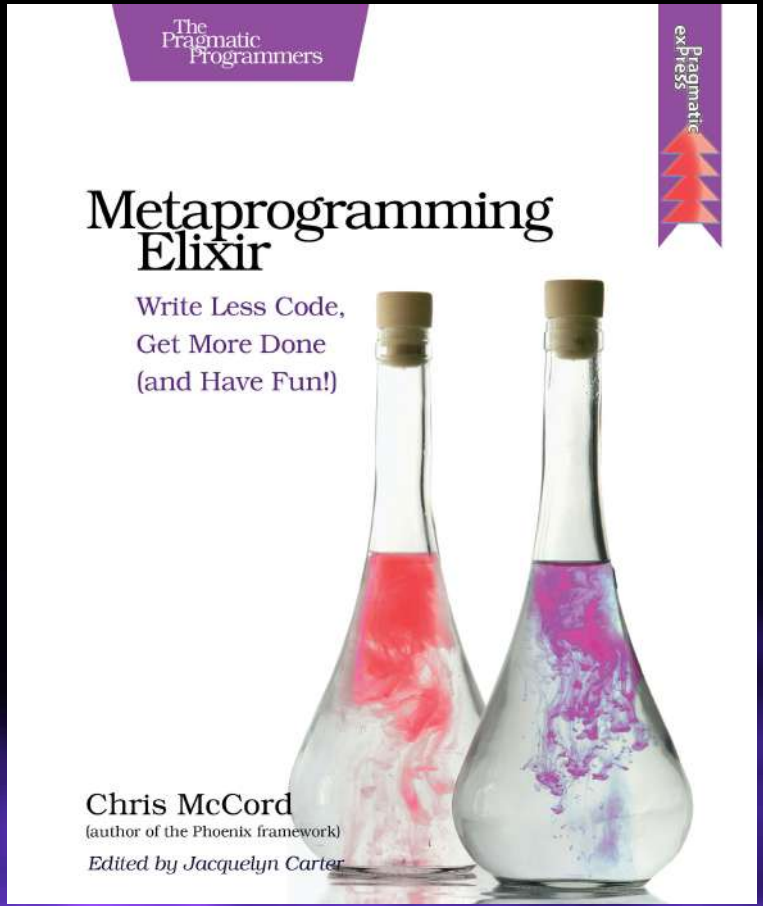
Limited DSL scope, no overloaded built ins, built on top of rather than instead of

Summary

- Basics
  - Macro Anatomy, Elixir AST, Quoted Expressions, Macro Expansion, Contexts, Hygiene, and Special Cases
- Examples
  - DSLs, Static Validations, Module Attributes, Compile Time Race Conditions, Compile Time Hooks
- Best Practices
  - Documentation, Developer Escape Hatches/ Override-ables, Reflections, Small Macros
- Pitfalls
  - Failing to use HOFs & Protocols instead, Hand Written ASTs, Low Debuggability, Huge Macros, Deadlocks, Testing code generation rather than generated code

Resources

- Syntactic Reference
  - https://hexdocs.pm/elixir/syntax-reference.html
- Kernel, Macro, Code, Module
- Optimal by Albert-IO
  - https://github.com/albert-io/optimal
- Metaprogramming Elixir
  - Chris McCord



Questions?