IT CAN'T BE THAT HARD: ELM SERIALIZERS AND FUZZ TESTING FOR FREE

or: what I learned writing a JSON Schema to Elm code generator in Elixir

WAIT - WHO ARE YOU?

- Peter Urbak

WAIT - WHO ARE YOU?

- Peter Urbak
- Functional polyglot programmer

WAIT - WHO ARE YOU?

- Peter Urbak
- Functional polyglot programmer
- Senior software developer at OTH.io
  - Platform for remote patient monitoring

MOTIVATION

Scratching that programming language theory itch

THE PROBLEM

Working with JSON in Elm

```
/* circle.json */
{
  "radius": "4.2",
  "center": {
    "x": "2.0",
    "y": "1.0"
  },
  "color": "red"
}
```

Elm Type Definitions

```
-- Circle.elm
type alias Circle =
    { radius : Float
    , center : Point
    , color : Maybe Color
    }

type alias Point =
    { x : Float
    , y : Float
    }

type Color = Red | Yellow | Green | Blue
```

Elm Type Definitions

```elm
-- Circle.elm
type alias Circle =
    { radius : Float
    , center : Point
    , color : Maybe Color
    }

type alias Point =
    { x : Float
    , y : Float
    }

type Color = Red | Yellow | Green | Blue
```

Elm Type Definitions

```
-- Circle.elm
type alias Circle =
    { radius : Float
    , center : Point
    , color : Maybe Color
    }

type alias Point =
    { x : Float
    , y : Float
    }

type Color = Red | Yellow | Green | Blue
```

Elm JSON Decoders

```elm
circleDecoder : Decoder Circle
circleDecoder =
    Decode.succeed Circle
        |> required "radius" Decode.float
        |> required "center" pointDecoder
        |> optional "color" (Decode.nullable colorDecoder) Nothing

pointDecoder : Decoder Point
pointDecoder =
    Decode.succeed Point
        |> required "x" Decode.float
        |> required "y" Decode.float

colorDecoder : Decoder Color
colorDecoder =
    Decode.string |> Decode.andThen (parseColor >> Decode.fromResult)

parseColor : String -> Result String Color
parseColor color =
    case color of
        "red" -> Ok Red
        "yellow" -> Ok Yellow
        "green" -> Ok Green
        "blue" -> Ok Blue
        _ -> Err <| "Unknown color type: " ++ color
```

Elm JSON Decoders

```elm
circleDecoder : Decoder Circle
circleDecoder =
    Decode.succeed Circle
        |> required "radius" Decode.float
        |> required "center" pointDecoder
        |> optional "color" (Decode.nullable colorDecoder) Nothing

pointDecoder : Decoder Point
pointDecoder =
    Decode.succeed Point
        |> required "x" Decode.float
        |> required "y" Decode.float

colorDecoder : Decoder Color
colorDecoder =
    Decode.string |> Decode.andThen (parseColor >> Decode.fromResult)

parseColor : String -> Result String Color
parseColor color =
    case color of
        "red" -> Ok Red
        "yellow" -> Ok Yellow
        "green" -> Ok Green
        "blue" -> Ok Blue
        _ -> Err <| "Unknown color type: " ++ color
```

Elm JSON Decoders

```elm
circleDecoder : Decoder Circle
circleDecoder =
    Decode.succeed Circle
        |> required "radius" Decode.float
        |> required "center" pointDecoder
        |> optional "color" (Decode.nullable colorDecoder) Nothing

pointDecoder : Decoder Point
pointDecoder =
    Decode.succeed Point
        |> required "x" Decode.float
        |> required "y" Decode.float

colorDecoder : Decoder Color
colorDecoder =
    Decode.string |> Decode.andThen (parseColor >> Decode.fromResult)

parseColor : String -> Result String Color
parseColor color =
    case color of
        "red" -> Ok Red
        "yellow" -> Ok Yellow
        "green" -> Ok Green
        "blue" -> Ok Blue
        _ -> Err <| "Unknown color type: " ++ color
```

Elm JSON Encoders

```elm
encodeCircle : Circle -> Value
encodeCircle circle =
    []  |> required "radius" circle.radius Encode.float
        |> required "center" circle.center encodePoint
        |> optional "color" circle.color encodeColor
        |> Encode.object

encodePoint : Point -> Value
encodePoint point =
    []  |> required "x" point.x Encode.float
        |> required "y" point.y Encode.float
        |> Encode.object

encodeColor : Color -> Value
encodeColor color =
    color |> colorToString |> Encode.string

colorToString : Color -> String
colorToString color =
    case color of
        Red -> "red"
        Yellow -> "yellow"
        Green -> "green"
        Blue -> "blue"
```

Elm JSON Encoders

```elm
encodeCircle : Circle -> Value
encodeCircle circle =
    [] |> required "radius" circle.radius Encode.float
       |> required "center" circle.center encodePoint
       |> optional "color" circle.color encodeColor
       |> Encode.object

encodePoint : Point -> Value
encodePoint point =
    [] |> required "x" point.x Encode.float
       |> required "y" point.y Encode.float
       |> Encode.object

encodeColor : Color -> Value
encodeColor color =
    color |> colorToString |> Encode.string

colorToString : Color -> String
colorToString color =
    case color of
        Red -> "red"
        Yellow -> "yellow"
        Green -> "green"
        Blue -> "blue"
```

Elm JSON Encoders

```elm
encodeCircle : Circle -> Value
encodeCircle circle =
    [] |> required "radius" circle.radius Encode.float
       |> required "center" circle.center encodePoint
       |> optional "color" circle.color encodeColor
       |> Encode.object

encodePoint : Point -> Value
encodePoint point =
    [] |> required "x" point.x Encode.float
       |> required "y" point.y Encode.float
       |> Encode.object

encodeColor : Color -> Value
encodeColor color =
    color |> colorToString |> Encode.string

colorToString : Color -> String
colorToString color =
    case color of
        Red -> "red"
        Yellow -> "yellow"
        Green -> "green"
        Blue -> "blue"
```

Working with JSON in Elm

Working with JSON in Elm

- Decoder/encoder learning curve.

Working with JSON in Elm

- Decoder/encoder learning curve.
- Can feel tedious.

Working with JSON in Elm

- Decoder/encoder learning curve.
- Can feel tedious.
- Already have JSON Schema descriptions.

Circle JSON Schema

```
{
  "$schema": "http://json-schema.org/draft-07/schema#",
  "$id": "https://dragonwasrobot.com/schemas/circle.json",
  "title": "Circle",
  "description": "Schema for a nice circular shape",
  "type": "object",
  "properties": {
    "radius": {
      "type": "number"
    },
    "center": {
      "$ref": "https://dragonwasrobot.com/schemas/definitions.json#point"
    },
    "color": {
      "$ref": "https://dragonwasrobot.com/schemas/definitions.json#color"
    }
  },
  "required": ["radius", "center"]
}
```

Circle JSON Schema

```
{
  "$schema": "http://json-schema.org/draft-07/schema#",
  "$id": "https://dragonwasrobot.com/schemas/circle.json",
  "title": "Circle",
  "description": "Schema for a nice circular shape",
  "type": "object",
  "properties": {
    "radius": {
      "type": "number"
    },
    "center": {
      "$ref": "https://dragonwasrobot.com/schemas/definitions.json#point"
    },
    "color": {
      "$ref": "https://dragonwasrobot.com/schemas/definitions.json#color"
    }
  },
  "required": ["radius", "center"]
}
```

Definitions JSON Schema

```
{
  "$schema": "http://json-schema.org/draft-07/schema#",
  "$id": "https://dragonwasrobot.com/schemas/definitions.json",
  "title": "Definitions",
  "definitions": {
    "point": {
      "$id": "#point",
      "type": "object",
      "properties": {
        "x": { "type": "number" },
        "y": { "type": "number" }
      },
      "required": [ "x", "y" ]
    },
    "color": {
      "$id": "#color",
      "type": "string",
      "enum": [ "red", "yellow", "green", "blue" ]
    }
  }
}
```

Definitions JSON Schema

```
{
  "$schema": "http://json-schema.org/draft-07/schema#",
  "$id": "https://dragonwasrobot.com/schemas/definitions.json",
  "title": "Definitions",
  "definitions": {
    "point": {
      "$id": "#point",
      "type": "object",
      "properties": {
        "x": { "type": "number" },
        "y": { "type": "number" }
      },
      "required": [ "x", "y" ]
    },
    "color": {
      "$id": "#color",
      "type": "string",
      "enum": [ "red", "yellow", "green", "blue" ]
    }
  }
}
```

Definitions JSON Schema

```
{
  "$schema": "http://json-schema.org/draft-07/schema#",
  "$id": "https://dragonwasrobot.com/schemas/definitions.json",
  "title": "Definitions",
  "definitions": {
    "point": {
      "$id": "#point",
      "type": "object",
      "properties": {
        "x": { "type": "number" },
        "y": { "type": "number" }
      },
      "required": [ "x", "y" ]
    },
    "color": {
      "$id": "#color",
      "type": "string",
      "enum": [ "red", "yellow", "green", "blue" ]
    }
  }
}
```

THE SOLUTION

Surely, it can't be that hard to take the JSON schema files we already have that describe our APIs and generate Elm types and serializers from them?

THE PLAN

```
json_data
|> describe_as_json_schema()
|> parse_json_schema_files_into_maps()
|> transform_maps_into_abstract_syntax_tree()
|> transform_ast_into_intermediate_representation()
|> print_ir_as_elm_files()
# optimize implementation
```

THE IMPLEMENTATION

- Parser library (`json_schema`)
- Code generator (`json-schema-to-elm`)

json_schema

Parse JSON Schema files into Elixir maps

```
json_schema_file_paths
    |> Enum.map(fn file_path ->
        file_path
        |> File.read!()
        |> Jason.decode!()
    end)
```

Transform maps into Abstract Syntax Tree (AST)

```
iex> URI.parse("http://dragonwasrobot.com/schemas/definitions.json#point")
%URI{
  scheme: "https",
  port: 443,
  host: "dragonwasrobot.com",
  path: "/schemas/definitions.json",
  fragment: "point",
  query: nil
}

iex> URI.parse("#/definitions/point")
%URI{
  scheme: nil,
  port: nil,
  host: nil,
  path: nil,
  fragment: "/definitions/point",
  query: nil
}
```

Transform maps into Abstract Syntax Tree (AST)

```
iex> URI.parse("http://dragonwasrobot.com/schemas/definitions.json#point")
%URI{
  scheme: "https",
  port: 443,
  host: "dragonwasrobot.com",
  path: "/schemas/definitions.json",
  fragment: "point",
  query: nil
}

iex> URI.parse("#/definitions/point")
%URI{
  scheme: nil,
  port: nil,
  host: nil,
  path: nil,
  fragment: "/definitions/point",
  query: nil
}
```

Transform maps into Abstract Syntax Tree (AST)

```elixir
defmodule JsonSchema.Types.ObjectType do

  alias JsonSchema.Types
  use TypedStruct

  typedstruct do
    field :name, String.t() | :anonymous, enforce: true
    field :description, String.t(), default: nil
    field :path, URI.t(), enforce: true
    field :properties, %{required(String.t()) => URI.t()}, enforce: true
    field :required, [String.t()], default: []
  end
end

%ObjectType{
     name: "circle",
     description: "A nice circular shape",
     path: URI.parse("#"),
     required: ["radius", "center"],
     properties: %{
       "radius" => URI.parse("#/properties/radius"),
       "center" => URI.parse("https://dragonwasrobot.com/definitions.json#point"),
       "color" => URI.parse("https://dragonwasrobot.com/definitions.json#color")
     }
}
```

Transform maps into Abstract Syntax Tree (AST)

```elixir
defmodule JsonSchema.Types.ObjectType do

  alias JsonSchema.Types
  use TypedStruct

  typedstruct do
    field :name, String.t() | :anonymous, enforce: true
    field :description, String.t(), default: nil
    field :path, URI.t(), enforce: true
    field :properties, %{required(String.t()) => URI.t()}, enforce: true
    field :required, [String.t()], default: []
  end
end

%ObjectType{
     name: "circle",
     description: "A nice circular shape",
     path: URI.parse("#"),
     required: ["radius", "center"],
     properties: %{
       "radius" => URI.parse("#/properties/radius"),
       "center" => URI.parse("https://dragonwasrobot.com/definitions.json#point"),
       "color" => URI.parse("https://dragonwasrobot.com/definitions.json#color")
     }
}
```

Transform maps into Abstract Syntax Tree (AST)

```elixir
defmodule JsonSchema.Types.ObjectType do

  alias JsonSchema.Types
  use TypedStruct

  typedstruct do
    field :name, String.t() | :anonymous, enforce: true
    field :description, String.t(), default: nil
    field :path, URI.t(), enforce: true
    field :properties, %{required(String.t()) => URI.t()}, enforce: true
    field :required, [String.t()], default: []
  end
end

%ObjectType{
     name: "circle",
     description: "A nice circular shape",
     path: URI.parse("#"),
     required: ["radius", "center"],
     properties: %{
       "radius" => URI.parse("#/properties/radius"),
       "center" => URI.parse("https://dragonwasrobot.com/definitions.json#point"),
       "color" => URI.parse("https://dragonwasrobot.com/definitions.json#color")
     }
}
```

Transform maps into Abstract Syntax Tree (AST)

```elixir
defmodule JsonSchema.Types.SchemaDefinition do

  @type typeDefinition ::
      ArrayType.t()
      | EnumType.t()
      | ObjectType.t()
      | PrimitiveType.t()
      | TypeReference.t()
      ...

  use TypedStruct

  typedstruct do
    field :file_path, Path.t(), enforce: true
    field :id, URI.t(), enforce: true
    field :title, String.t(), enforce: true
    field :description, String.t(), enforce: nil
    field :types, %{URI.t() => typeDefinition}, enforce: true
  end
end
```

Transform maps into Abstract Syntax Tree (AST)

```elixir
defmodule JsonSchema.Types.SchemaDefinition do

  @type typeDefinition ::
      ArrayType.t()
      | EnumType.t()
      | ObjectType.t()
      | PrimitiveType.t()
      | TypeReference.t()
      ...

  use TypedStruct

  typedstruct do
    field :file_path, Path.t(), enforce: true
    field :id, URI.t(), enforce: true
    field :title, String.t(), enforce: true
    field :description, String.t(), enforce: nil
    field :types, %{URI.t() => typeDefinition}, enforce: true
  end
end
```

The final `json_schema` API

```elixir
@type abstractSyntaxTree :: %{URI.t() => SchemaDefinition.t()}

@spec parse_schema_files([Path.t()]) :: {:ok, abstractSyntaxTree()} | {:error, ParserError.t()}
def parse_schema_files(schema_paths)

@spec resolve_type(URI.t(), SchemaDefinition.t(), abstractSyntaxTree())
    :: {:ok, {typeDefinition(), SchemaDefinition.t()}}
     | {:error, ParserError.t()}
def resolve_type(identifier, schema_def, schema_ast)
```

The final `json_schema` API

```elixir
@type abstractSyntaxTree :: %{URI.t() => SchemaDefinition.t()}

@spec parse_schema_files([Path.t()]) :: {:ok, abstractSyntaxTree()} | {:error, ParserError.t()}
def parse_schema_files(schema_paths)

@spec resolve_type(URI.t(), SchemaDefinition.t(), abstractSyntaxTree())
    :: {:ok, {typeDefinition(), SchemaDefinition.t()}}
     | {:error, ParserError.t()}
def resolve_type(identifier, schema_def, schema_ast)
```

The final `json_schema` API

```elixir
@type abstractSyntaxTree :: %{URI.t() => SchemaDefinition.t()}

@spec parse_schema_files([Path.t()]) :: {:ok, abstractSyntaxTree()} | {:error, ParserError.t()}
def parse_schema_files(schema_paths)

@spec resolve_type(URI.t(), SchemaDefinition.t(), abstractSyntaxTree())
    :: {:ok, {typeDefinition(), SchemaDefinition.t()}}
     | {:error, ParserError.t()}
def resolve_type(identifier, schema_def, schema_ast)
```

json-schema-to-elm

Transform AST into Intermediate Representation (IR)

```elixir
# elm_types.ex
@type type_definition :: {:product, product_type} | {:sum, sum_type}

@type product_type :: %{
    name: String.t(),
    fields: [%{name: String.t(), type: String.t()}]
}

# elm_decoders.ex
@type decoder_definition :: {:product, product_decoder} | {:sum, sum_decoder}

@type product_decoder :: %{
    name: String.t(),
    type: String.t(),
    clauses: [%{option: option, property_name: String.t(), decoder_name: String.t()}]
}

@type option :: :required | :optional
```

Transform AST into Intermediate Representation (IR)

```elixir
# elm_types.ex
@type type_definition :: {:product, product_type} | {:sum, sum_type}

@type product_type :: %{
    name: String.t(),
    fields: [%{name: String.t(), type: String.t()}]
}

# elm_decoders.ex
@type decoder_definition :: {:product, product_decoder} | {:sum, sum_decoder}

@type product_decoder :: %{
    name: String.t(),
    type: String.t(),
    clauses: [%{option: option, property_name: String.t(), decoder_name: String.t()}]
}

@type option :: :required | :optional
```

Print IR to Elm files

```
# product_type.elm.eex
type alias <%= product_type.name %> =
    { <%= Enum.map_join(fields, "\n    , ", fn field -> "#{field.name} : #{field.type}" end) %>
    }

# product_decoder.elm.eex
<%= product_decoder.name %> : Decoder <%= product_decoder.type %>
<%= product_decoder.name %> =
    Decode.succeed <%= product_decoder.type %><%#
      %><%= for clause <- product_decoder.clauses do %>
        |> <%= clause.option %> "<%= clause.property_name %>" <%= clause.decoder_name %><%#
      %><% end %>
```

Print IR to Elm files

```
# product_type.elm.eex
type alias <%= product_type.name %> =
    { <%= Enum.map_join(fields, "\n    , ", fn field -> "#{field.name} : #{field.type}" end) %>
    }

# product_decoder.elm.eex
<%= product_decoder.name %> : Decoder <%= product_decoder.type %>
<%= product_decoder.name %> =
    Decode.succeed <%= product_decoder.type %><%#
      %><%= for clause <- product_decoder.clauses do %>
        |> <%= clause.option %> "<%= clause.property_name %>" <%= clause.decoder_name %><%#
      %><% end %>
```

Print IR to Elm files

```elm
-- Circle.elm
type alias Circle =
    { radius : Float
    , center : Point
    , color : Maybe Color
    }

circleDecoder : Decoder Circle
circleDecoder =
    Decode.succeed Circle
        |> required "radius" Decode.float
        |> required "center" pointDecoder
        |> optional "color" (Decode.nullable colorDecoder) Nothing
```

OPTIMIZATIONS

Generate Elm property/fuzz tests from IRs

```elm
-- Property: value |> encodeValue |> decodeValue == value

encodeDecodeCircleTest : Test
encodeDecodeCircleTest =
    fuzz Circle "can encode and decode circle object" <|
        \circle ->
            circle |> Circle.encodeCircle
                   |> Decode.decodeValue Circle.decodeCircle
                   |> Expect.equal (Ok circle)

circleFuzzer : Fuzzer Circle
circleFuzzer =
    Fuzz.map3 Circle pointFuzzer (Fuzz.maybe colorFuzzer) Fuzz.float

pointFuzzer : Fuzzer Point
pointFuzzer =
    Fuzz.map2 Point Fuzz.float Fuzz.float

colorFuzzer : Fuzzer Color
colorFuzzer =
    [ Red, Yellow, Green, Blue ] |> List.map Fuzz.constant |> Fuzz.oneOf
```

Generate Elm property/fuzz tests from IRs

```elm
-- Property: value |> encodeValue |> decodeValue == value

encodeDecodeCircleTest : Test
encodeDecodeCircleTest =
    fuzz Circle "can encode and decode circle object" <|
        \circle ->
            circle |> Circle.encodeCircle
                   |> Decode.decodeValue Circle.decodeCircle
                   |> Expect.equal (Ok circle)

circleFuzzer : Fuzzer Circle
circleFuzzer =
    Fuzz.map3 Circle pointFuzzer (Fuzz.maybe colorFuzzer) Fuzz.float

pointFuzzer : Fuzzer Point
pointFuzzer =
    Fuzz.map2 Point Fuzz.float Fuzz.float

colorFuzzer : Fuzzer Color
colorFuzzer =
    [ Red, Yellow, Green, Blue ] |> List.map Fuzz.constant |> Fuzz.oneOf
```

Generate Elm property/fuzz tests from IRs

```elm
-- Property: value |> encodeValue |> decodeValue == value

encodeDecodeCircleTest : Test
encodeDecodeCircleTest =
    fuzz Circle "can encode and decode circle object" <|
        \circle ->
            circle |> Circle.encodeCircle
                   |> Decode.decodeValue Circle.decodeCircle
                   |> Expect.equal (Ok circle)

circleFuzzer : Fuzzer Circle
circleFuzzer =
    Fuzz.map3 Circle pointFuzzer (Fuzz.maybe colorFuzzer) Fuzz.float

pointFuzzer : Fuzzer Point
pointFuzzer =
    Fuzz.map2 Point Fuzz.float Fuzz.float

colorFuzzer : Fuzzer Color
colorFuzzer =
    [ Red, Yellow, Green, Blue ] |> List.map Fuzz.constant |> Fuzz.oneOf
```

Generate Elm property/fuzz tests from IRs

```elm
-- Property: value |> encodeValue |> decodeValue == value

encodeDecodeCircleTest : Test
encodeDecodeCircleTest =
    fuzz Circle "can encode and decode circle object" <|
        \circle ->
            circle |> Circle.encodeCircle
                   |> Decode.decodeValue Circle.decodeCircle
                   |> Expect.equal (Ok circle)

circleFuzzer : Fuzzer Circle
circleFuzzer =
    Fuzz.map3 Circle pointFuzzer (Fuzz.maybe colorFuzzer) Fuzz.float

pointFuzzer : Fuzzer Point
pointFuzzer =
    Fuzz.map2 Point Fuzz.float Fuzz.float

colorFuzzer : Fuzzer Color
colorFuzzer =
    [ Red, Yellow, Green, Blue ] |> List.map Fuzz.constant |> Fuzz.oneOf
```

Generate helper modules

```elm
module Data.Encode exposing (optional, required)

import Json.Encode as Encode exposing (Value)

required :
    String
    -> a
    -> (a -> Value)
    -> List ( String, Value )
    -> List ( String, Value )
required key value encode properties =
    properties ++ [ ( key, encode value ) ]

optional :
    String
    -> Maybe a
    -> (a -> Value)
    -> List ( String, Value )
    -> List ( String, Value )
optional key maybe encode properties =
    maybe
        |> Maybe.map (\value -> properties ++ [ ( key, encode value ) ])
        |> Maybe.withDefault properties
```

Generate scaffolding files

```
output/
|-- .tool-versions
|-- package.json
|-- elm.json
|-- src/
    |-- Circle.elm
    |-- Definitions.elm
    |-- Encode.elm
    |-- Decode.elm
|-- tests/
    |-- CircleTests.elm
    |-- DefinitionsTests.elm
```

Present Elm-style error messages

```
--- UNKNOWN NODE TYPE ------------------------------------------- circle.json

The value of "type" at '#/properties/description' did not match a known node type

    "type": "strink"
            ^^^^^^^^

Was expecting one of the following types

    ["null", "boolean", "object", "array", "number", "integer", "string"]

Hint: See the specification section 6.25. "Validation keywords - type"
   <http://json-schema.org/latest/json-schema-validation.html#rfc.section.6.25>



--- UNRESOLVED REFERENCE ---------------------------------------- circle.json

The following reference at `#/properties/color` could not be resolved

"$ref": "https://www.dragonwarobot.com/schemas/definitions.json#kolor"
        ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^

Hint: See the specification section 9. "Base URI and dereferencing"
   <http://json-schema.org/latest/json-schema-core.html#rfc.section.9>
```

LESSONS LEARNED

Parser library (`json_schema`):

LESSONS LEARNED

Parser library (`json_schema`):

1. Code as input data.

LESSONS LEARNED

Parser library (`json_schema`):

1. Code as input data.
2. Reading Specifications/RFCs:
    - JSON Schema core + validation and URI.

LESSONS LEARNED

Parser library (`json_schema`):

1. Code as input data.
2. Reading Specifications/RFCs:
    - JSON Schema core + validation and URI.
3. URIs as scheme for identifying/resolving resources.

LESSONS LEARNED

Parser library (`json_schema`):

1. Code as input data.
2. Reading Specifications/RFCs:
    - JSON Schema core + validation and URI.
3. URIs as scheme for identifying/resolving resources.
4. The subtle art of helpful error messages.

LESSONS LEARNED

Parser library (`json_schema`):

1. Code as input data.
2. Reading Specifications/RFCs:
    - JSON Schema core + validation and URI.
3. URIs as scheme for identifying/resolving resources.
4. The subtle art of helpful error messages.
5. Writing API and docs for a general purpose library.

LESSONS LEARNED

Code generator (`json-schema-to-elm`):

LESSONS LEARNED

Code generator (`json-schema-to-elm`):

1. Code as output data.

LESSONS LEARNED

Code generator (`json-schema-to-elm`):

1. Code as output data.
2. Analyzing the syntax of a programming language.

LESSONS LEARNED

Code generator (`json-schema-to-elm`):

1. Code as output data.
2. Analyzing the syntax of a programming language.
3. The subtle art of code generation.

LESSONS LEARNED

Code generator (`json-schema-to-elm`):

1. Code as output data.
2. Analyzing the syntax of a programming language.
3. The subtle art of code generation.
4. Optimizations and developer friendliness:
    - Fuzz tests, helper modules, scaffolding files, etc.

QUESTIONS

- **Name:** Peter Urbak
- **Company:** OTH.io
- **Github:** `dragonwasrobot`
- **Website:** `dragonwasrobot.com`
- **LinkedIn:** `peterurbak`