ASA·I 浅井

```
info: you are at WITS
info: j.w.w. Reed Mullanix
warning: favonia on stage
```

A typical implementation day

Exhausted after listening to all POPL talks;
no energy to implement error handling until…

A typical implementation day

Exhausted after listening to all POPL talks; no energy to implement error handling until...

```
Exception: Failure "type error".
Raised at Stdlib.failwith in file "std
lib.ml", line 29, characters 17-33
Called from <unknown> in file "./test.
ml", line 12, characters 9-17
Called from Topeval.load_lambda in fil
e "toplevel/byte/topeval.ml", line 89,
 characters 4-14
```

What constitutes a $\overset{\text{good}}{\wedge}$ diagnostic

What constitutes a $\overset{\text{good}}{\wedge}$ diagnostic

Should the program terminate now?

What constitutes a $\overset{\text{good}}{\wedge}$ diagnostic

Should the program terminate now?

How seriously should the user take it?

warning, error, or info?

What constitutes a $\overset{\text{good}}{\wedge}$ diagnostic

Should the program terminate now?

How seriously should the user take it?

warning, error, or info?

these two
are different!

What constitutes a $\overset{\text{good}}{\wedge}$ diagnostic

Should the program terminate now?

How seriously should the user take it?

these two
are different!

warning, error, or info?

A Google-able code "E0411 site:stackoverflow.com"

What constitutes a $\overset{\text{good}}{\wedge}$ diagnostic

Should the program terminate now?

How seriously should the user take it?

warning, error, or info?

these two are different!

A Google-able code "E0411 site:stackoverflow.com"

A user-perceived stack backtrace

not call backtrace for debugging! diagnostics are for users, not *you!*

What constitutes a $\overset{\text{good}}{\wedge}$ diagnostic

Should the program terminate now?

How seriously should the user take it?

these two
are different!

warning, error, or info?

A Google-able code "E0411 site:stackoverflow.com"

A user-perceived stack backtrace

not call backtrace for debugging! diagnostics are for users, not *you!*

Allowing multiple spans (locations in source files)

Structured or unstructured?

Structured or unstructured?

100% structured: `emit (TypeError ("List.rev", tm, tp))`

Structured or unstructured?

100% structured: `emit (TypeError ("List.rev", tm, tp))`

0% structured: `emit "type error: List.rev is evil"`

Structured or unstructured?

100% structured: `emit (TypeError ("List.rev", tm, tp))`

0% structured: `emit "type error: List.rev is evil"`

50% structured: `emit TypeError "List.rev is evil"`

Structured or unstructured?

100% structured:	`emit (TypeError ("List.rev", tm, tp))`
0% structured:	`emit "type error: List.rev is evil"`
50% structured:	`emit TypeError "List.rev is evil"`

Some structuredness, especially the classification, helps users identify (= Google) relevant help documents

Structured or unstructured?

100% structured:	`emit (TypeError ("List.rev", tm, tp))`
0% structured:	`emit "type error: List.rev is evil"`
50% structured:	`emit TypeError "List.rev is evil"`

- Some structuredness, especially the classification, helps users identify (= Google) relevant help documents
- However, full structuredness is challenging for very ad-hoc messages
  Think about all possible errors from parsing

Structured or unstructured?

100% structured: `emit (TypeError ("List.rev", tm, tp))`

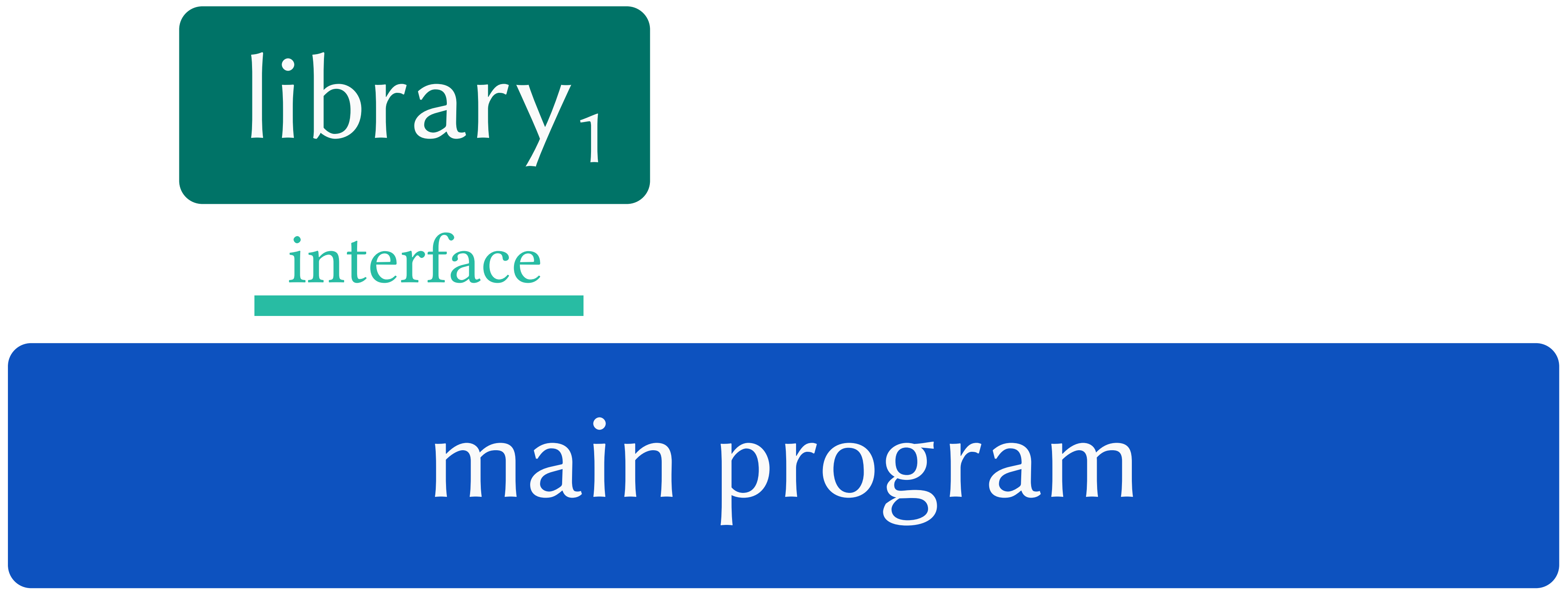
0% structured: `emit "type error: List.rev is evil"`

50% structured: `emit TypeError "List.rev is evil"`

- Some structuredness, especially the classification, helps users identify (= Google) relevant help documents
- However, full structuredness is challenging for very ad-hoc messages Think about all possible errors from parsing
- Which one? We support both the 100% and 50% style!

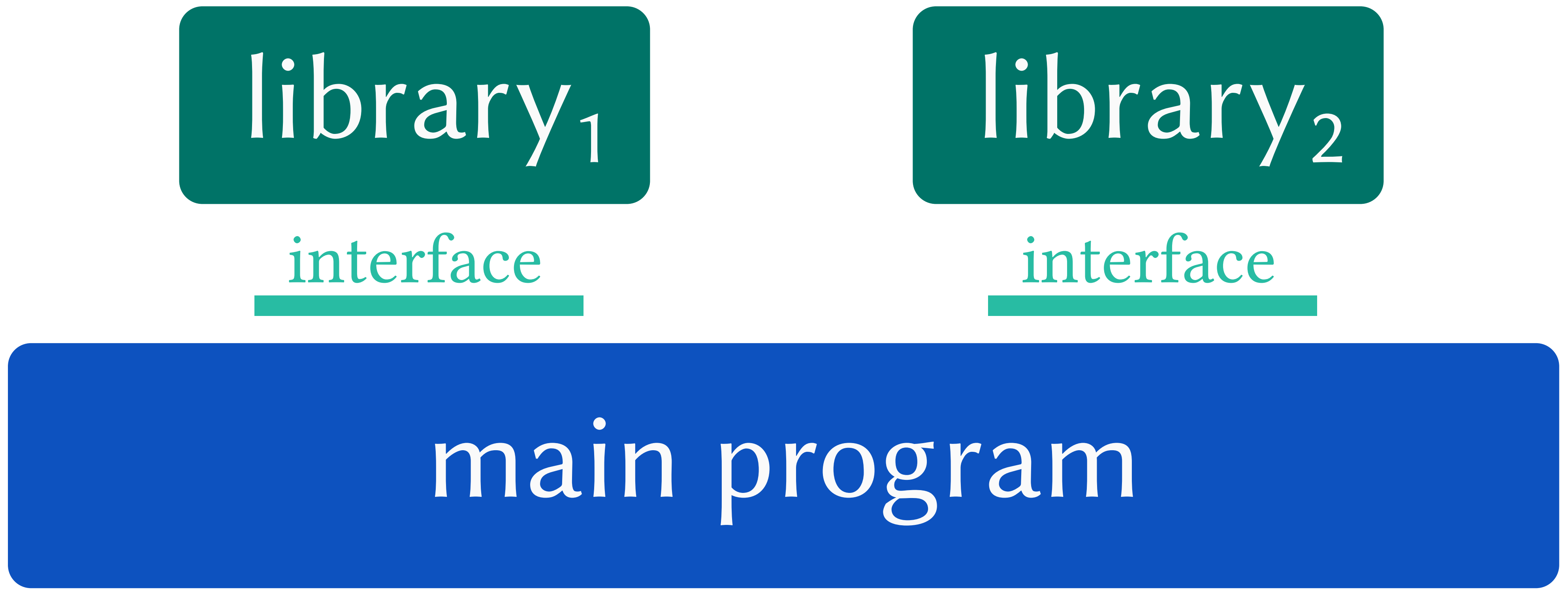
Compositionality

It should be easy to use a library that also uses asai



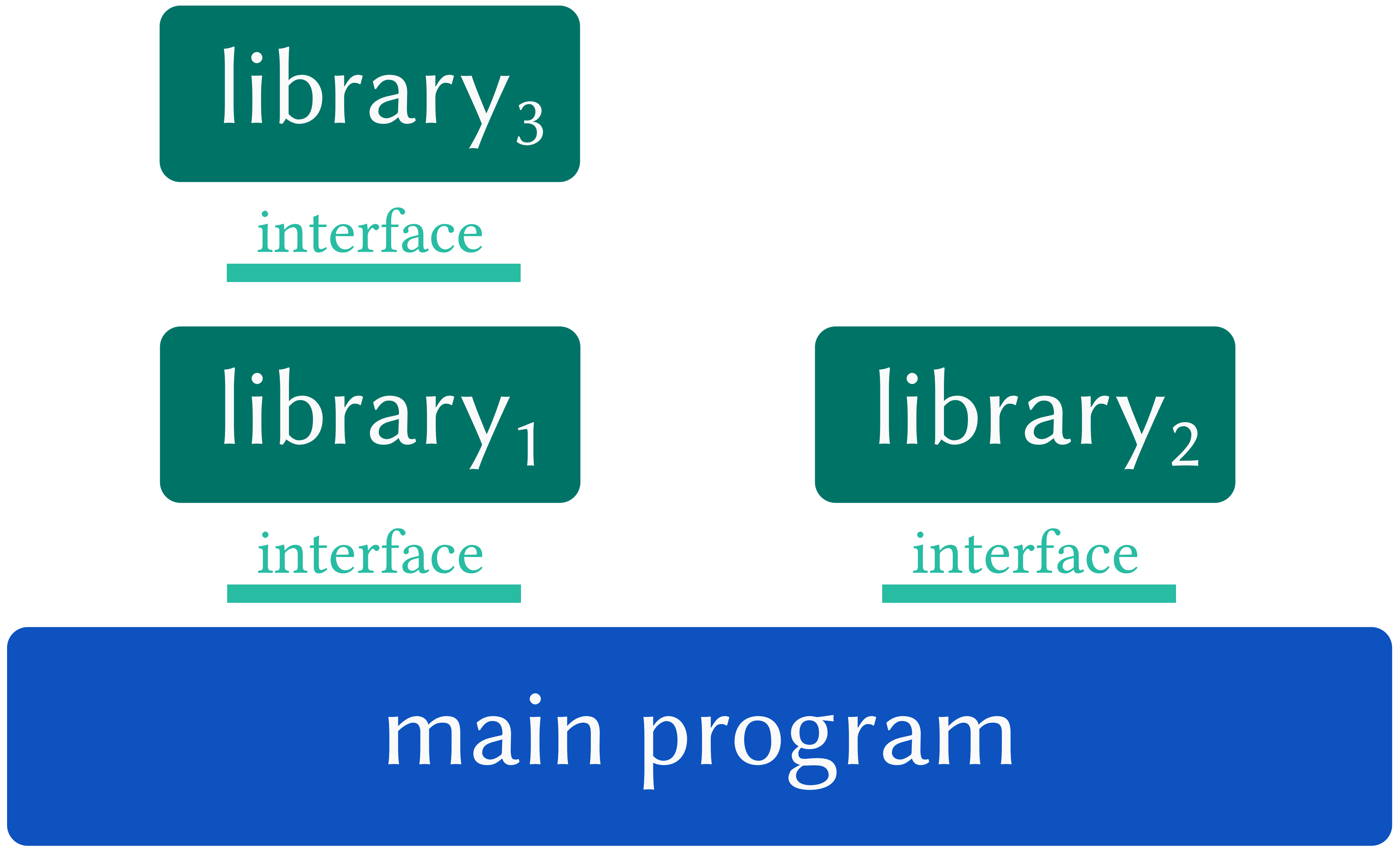
Compositionality

It should be easy to use a library that also uses asai



Compositionality

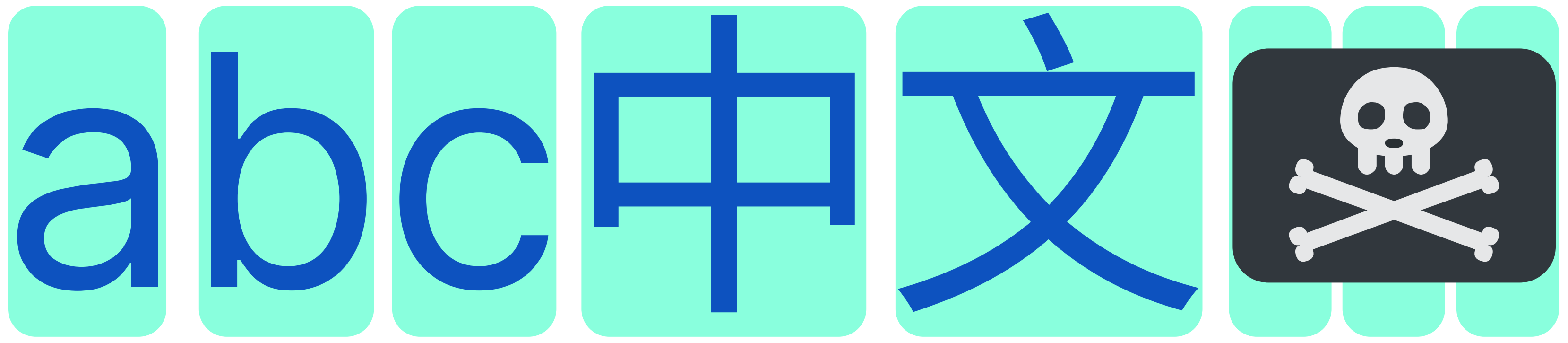
It should be easy to use a library that also uses asai



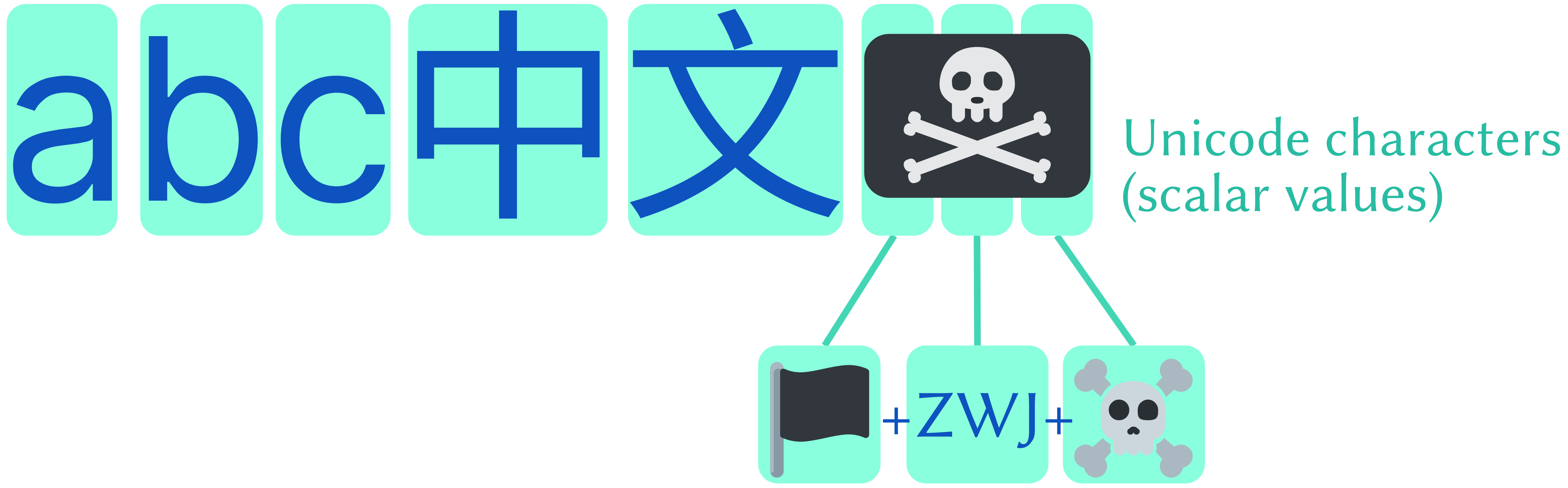
Unicode Support

abc中文☠️

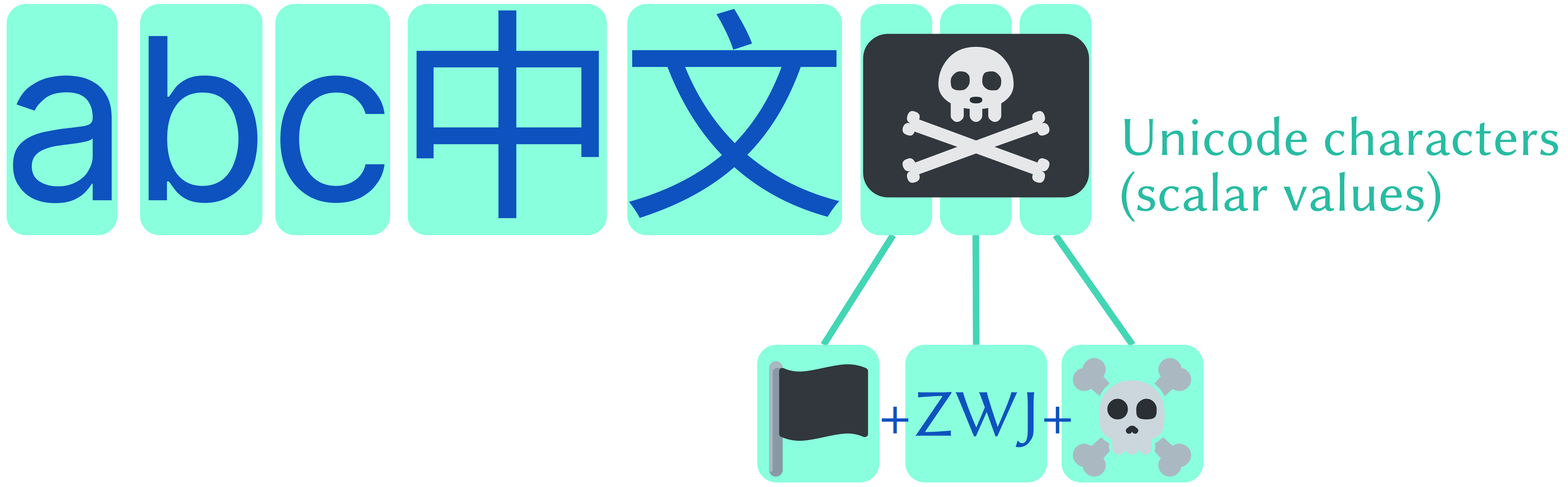
Unicode Support



Unicode Support



Unicode Support



No easy way to predict the visual widths

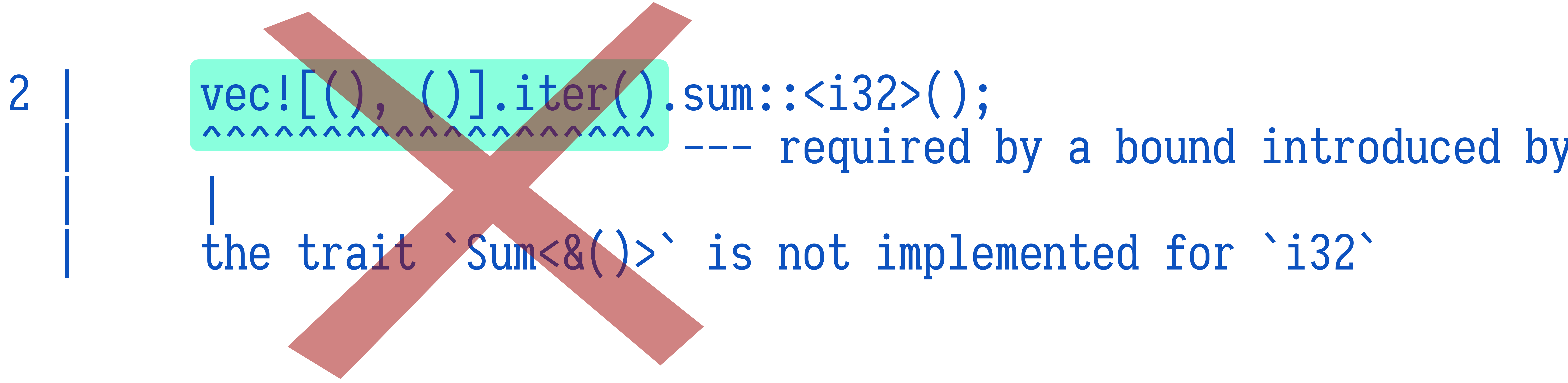
Your fonts, terminals, and maybe locales matter

Many programs use (broken) heuristics

Unicode Support

```
2 |     vec![(), ()].iter().sum::<i32>();
  |     ^^^^^^^^^^^^^^^^^^^ --- required by a bound introduced by
  |     |
  |     the trait `Sum<&()>` is not implemented for `i32`
```

Unicode Support

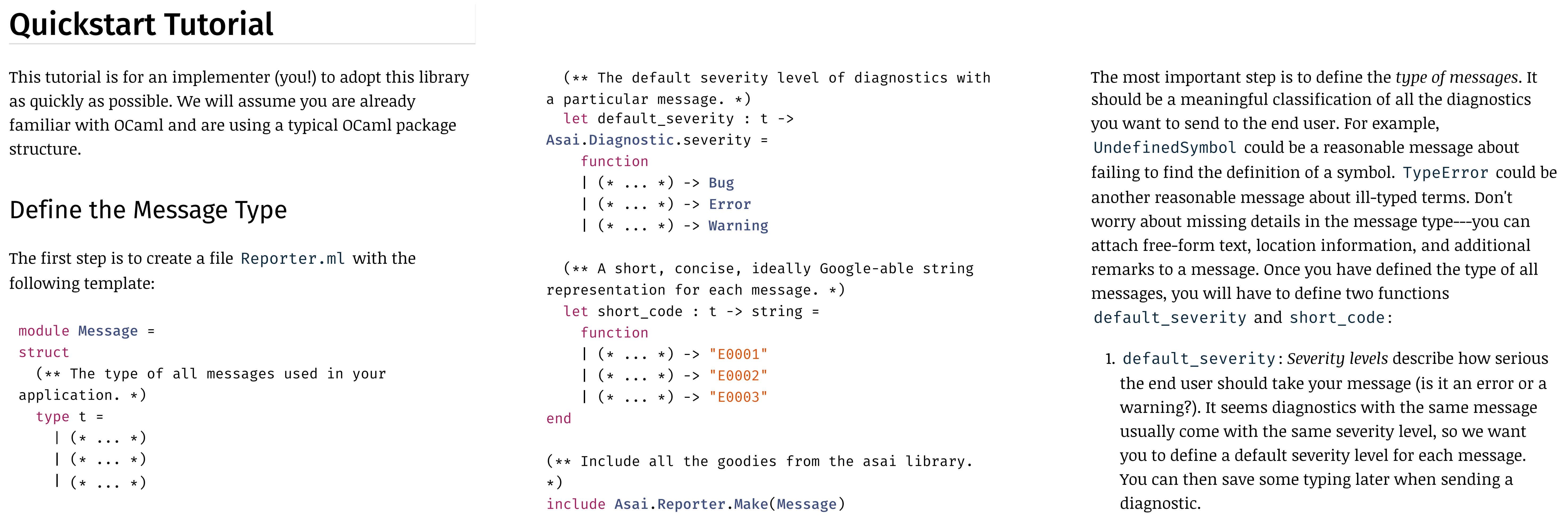


```
2 |     vec![(), ()].iter().sum::<i32>();
  |     ^^^^^^^^^^^^^^^^^^^^ --- required by a bound introduced by
  |     |
  |     the trait `Sum<&()>` is not implemented for `i32`
```

You cannot know the visual width!

If it fails for emojis, it fails. Period.

Tutorial-Oriented Design



Quickstart Tutorial

This tutorial is for an implementer (you!) to adopt this library as quickly as possible. We will assume you are already familiar with OCaml and are using a typical OCaml package structure.

Define the Message Type

The first step is to create a file `Reporter.ml` with the following template:

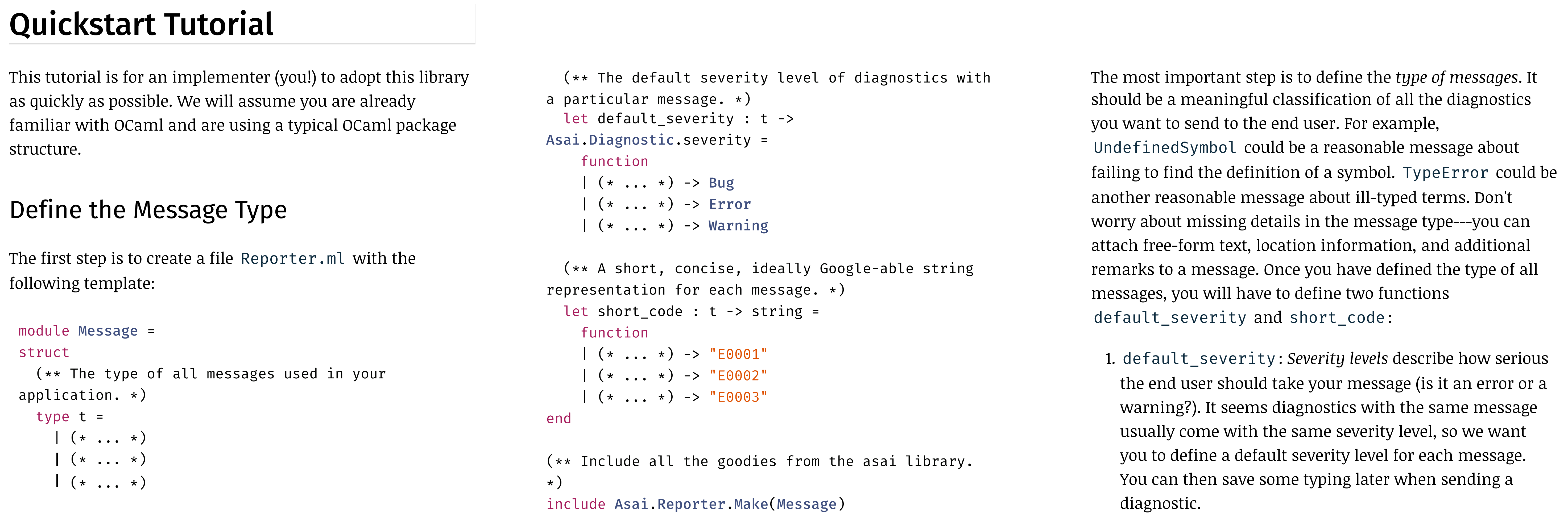
```
module Message =
struct
  (** The type of all messages used in your
application. *)
  type t =
    | (* ... *)
    | (* ... *)
    | (* ... *)

  (** The default severity level of diagnostics with
a particular message. *)
  let default_severity : t ->
Asai.Diagnostic.severity =
    function
    | (* ... *) -> Bug
    | (* ... *) -> Error
    | (* ... *) -> Warning

  (** A short, concise, ideally Google-able string
representation for each message. *)
  let short_code : t -> string =
    function
    | (* ... *) -> "E0001"
    | (* ... *) -> "E0002"
    | (* ... *) -> "E0003"
end

(** Include all the goodies from the asai library.
*)
include Asai.Reporter.Make(Message)
```

The most important step is to define the *type of messages*. It should be a meaningful classification of all the diagnostics you want to send to the end user. For example, `UndefinedSymbol` could be a reasonable message about failing to find the definition of a symbol. `TypeError` could be another reasonable message about ill-typed terms. Don't worry about missing details in the message type---you can attach free-form text, location information, and additional remarks to a message. Once you have defined the type of all messages, you will have to define two functions `default_severity` and `short_code`:

1. `default_severity`: *Severity levels* describe how serious the end user should take your message (is it an error or a warning?). It seems diagnostics with the same message usually come with the same severity level, so we want you to define a default severity level for each message. You can then save some typing later when sending a diagnostic.

https://redprl.org/asai/asai/quickstart.html

Tutorial-Oriented Design



Quickstart Tutorial

This tutorial is for an implementer (you!) to adopt this library as quickly as possible. We will assume you are already familiar with OCaml and are using a typical OCaml package structure.

Define the Message Type

The first step is to create a file `Reporter.ml` with the following template:

```
module Message =
struct
  (** The type of all messages used in your
application. *)
  type t =
    | (* ... *)
    | (* ... *)
    | (* ... *)

  (** The default severity level of diagnostics with
a particular message. *)
  let default_severity : t ->
Asai.Diagnostic.severity =
    function
    | (* ... *) -> Bug
    | (* ... *) -> Error
    | (* ... *) -> Warning

  (** A short, concise, ideally Google-able string
representation for each message. *)
  let short_code : t -> string =
    function
    | (* ... *) -> "E0001"
    | (* ... *) -> "E0002"
    | (* ... *) -> "E0003"
end

(** Include all the goodies from the asai library.
*)
include Asai.Reporter.Make(Message)
```

The most important step is to define the *type of messages*. It should be a meaningful classification of all the diagnostics you want to send to the end user. For example, `UndefinedSymbol` could be a reasonable message about failing to find the definition of a symbol. `TypeError` could be another reasonable message about ill-typed terms. Don't worry about missing details in the message type---you can attach free-form text, location information, and additional remarks to a message. Once you have defined the type of all messages, you will have to define two functions `default_severity` and `short_code`:

1. `default_severity`: *Severity levels* describe how serious the end user should take your message (is it an error or a warning?). It seems diagnostics with the same message usually come with the same severity level, so we want you to define a default severity level for each message. You can then save some typing later when sending a diagnostic.

https://redprl.org/asai/asai/quickstart.html

Write a tutorial to improve your design

rubber duck design™

Related OCaml Work

	asai	**Grace** (just released)
representaiton (type of diagnostics)	LSP-style	Rust-style

Related OCaml Work

	asai	**Grace** (just released)
representaiton (type of diagnostics)	LSP-style	Rust-style
generation	algebraic effects	

Related OCaml Work

	asai	**Grace** (just released)
representaiton (type of diagnostics)	LSP-style	Rust-style
generation	algebraic effects	
rendering	emoji-focused	Rust-inspired

Related OCaml Work

	asai	**Grace** (just released)
representaiton (type of diagnostics)	LSP-style	Rust-style
generation	algebraic effects	
rendering	emoji-focused	Rust-inspired

Current plan: bridge these two libraries

Success Stories

Success Stories

algaett: our prototype to check things combine

Success Stories

algaett: our prototype to check things combine

forester: Jon Sterling's tool to generate his website

(not a proof assistant!)

Success Stories

algaett: our prototype to check things combine

forester: Jon Sterling's tool to generate his website

(not a proof assistant!)

(WIP): Mike Shulman's type checker for HOTT

Success Stories

algaett: our prototype to check things combine

forester: Jon Sterling's tool to generate his website

(not a proof assistant!)

(WIP): Mike Shulman's type checker for HOTT

(???): *(Your next tool here)*

Success Stories

algaett: our prototype to check things combine

forester: Jon Sterling's tool to generate his website

(not a proof assistant!)

(WIP): Mike Shulman's type checker for HOTT

(???): *(Your next tool here)*

`https://ocaml.org/p/asai`