Advanced Topics in OCaml

Concepts of Programming Languages
Lecture 11

CAS CS 320

Recap
HOFs Beyond Lists

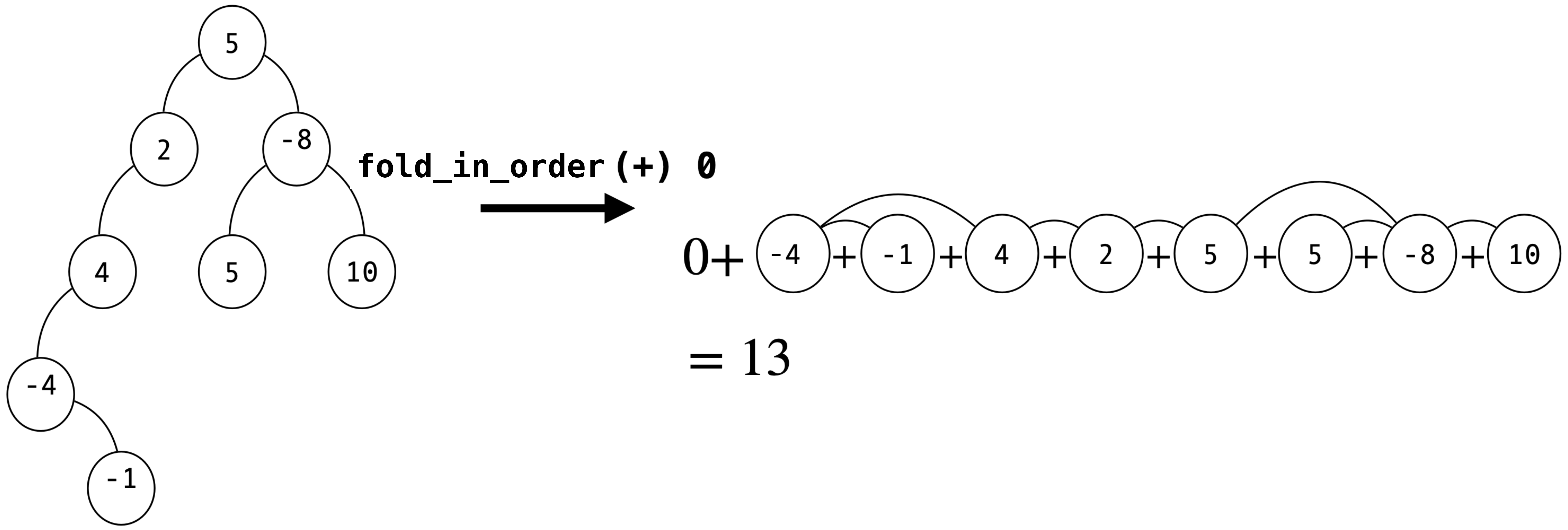
Trees

```
type 'a tree =
  | Leaf
  | Node of 'a * 'a tree * 'a tree

let map f t =
  let rec go t =
    match t with
    | Leaf -> Leaf
    | Node (x, l, r) -> Node (f x, go l, go r)
  in go t
```

Mapping over a tree maintains the structure but recursively updates values with **f**

Fold over Trees (The Picture)



Practice Problem

Implement the function

val fold_in_order :
(‘a -> ‘b -> ‘a) -> ‘a -> ‘b tree -> ‘a

so that **fold_in_order op base t n** *does* an **in-order** fold for trees

It is equivalent to "flattening" the tree into a list, and then folding that list

(This is different from what is given in the textbook)

Outline

- » Practice **typing derivation** with HOFs
- » Give a short tutorial on **ounit** a unit test framework for OCaml
- » A quick look at **modules** in OCaml (another very cool feature)

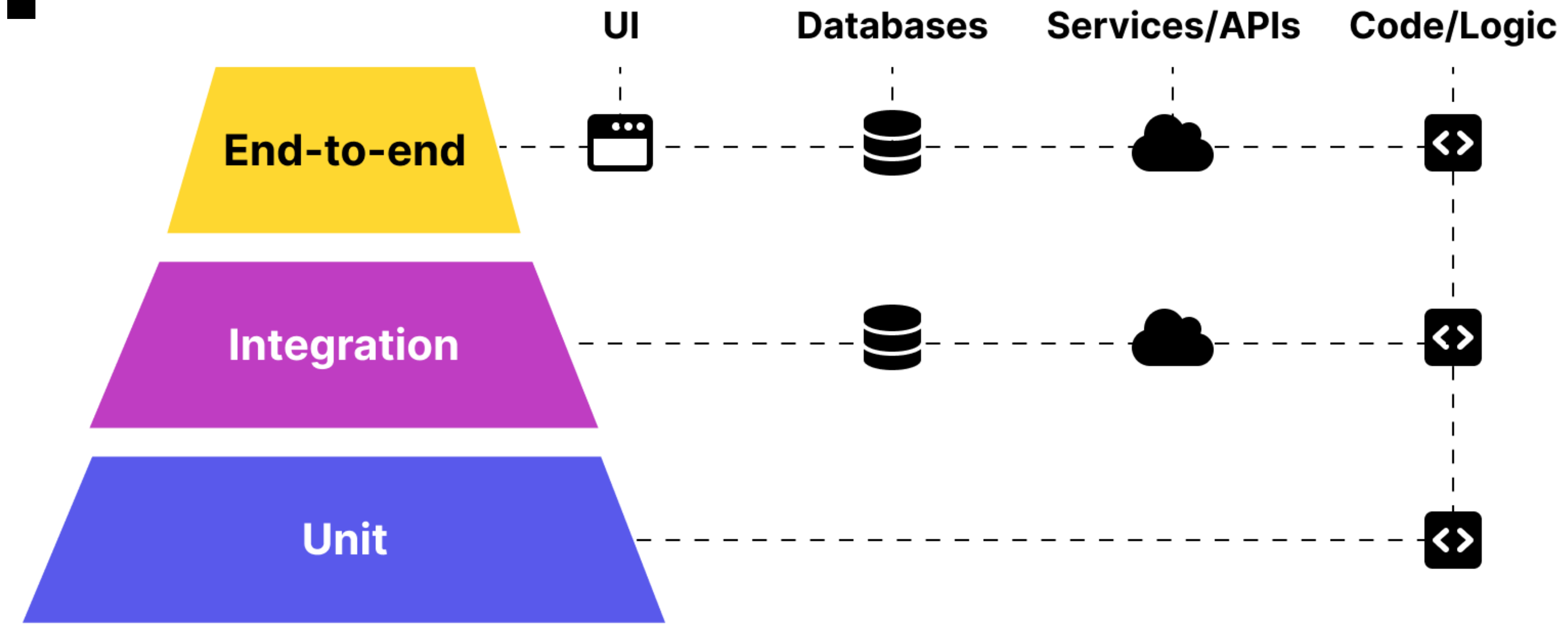
Typing Derivation for fold_right

$$\Gamma = \{\text{foldr} : (\alpha \to \beta \to \beta) \to \alpha \text{ list} \to \beta \to \beta,$$
$$\text{op} : \alpha \to \beta \to \beta,$$
$$l : \alpha \text{ list},$$
$$\text{base} : \beta\}$$
$$\Gamma' = \Gamma, x : \alpha, xs : \alpha \text{ list}$$

$$\dfrac{\dfrac{\text{VAR}}{\Gamma \vdash l : \alpha \text{ list}} \quad \dfrac{\text{VAR}}{\Gamma \vdash \text{base} : \beta} \quad \dfrac{\dfrac{\text{VAR}}{\Gamma' \vdash \text{op} : \alpha \to \beta \to \beta} \quad \dfrac{\text{VAR}}{\Gamma' \vdash x : \alpha} \quad \dfrac{\dfrac{}{\Gamma' \vdash \text{foldr} : \overset{(\alpha\to\beta\to\beta)}{\to \alpha \text{ list} \to \beta \to \beta}} \quad \dfrac{}{\Gamma' \vdash \text{op} : \alpha \to \beta \to \beta} \quad \dfrac{}{\Gamma' \vdash xs : \alpha \text{ list}} \quad \dfrac{}{\Gamma' \vdash \text{base} : \beta}}{\Gamma' \vdash \text{foldr op } \overset{xs}{\cancel{\text{base}}} \ \overset{\text{base}}{\cancel{xs}} : \beta}}{\Gamma, x : \alpha, xs : \alpha \text{ list} \vdash \text{op } x \ (\text{foldr op } \overset{xs}{\cancel{\text{base}}} \ \overset{\text{base}}{\cancel{xs}}) : \beta}}{\{\text{foldr} : (\alpha \to \beta \to \beta) \to \alpha \text{ list} \to \beta, \text{op} : \alpha \to \beta \to \beta, \text{base} : \beta, l : \alpha \text{ list}\} \vdash \\ \text{match } l \text{ with } \mid [\,] \to \text{base} \mid x :: xs \to \text{op } x \ (\text{foldr op } \overset{xs}{\cancel{\text{base}}} \ \overset{\text{base}}{\cancel{xs}}) : \beta}$$

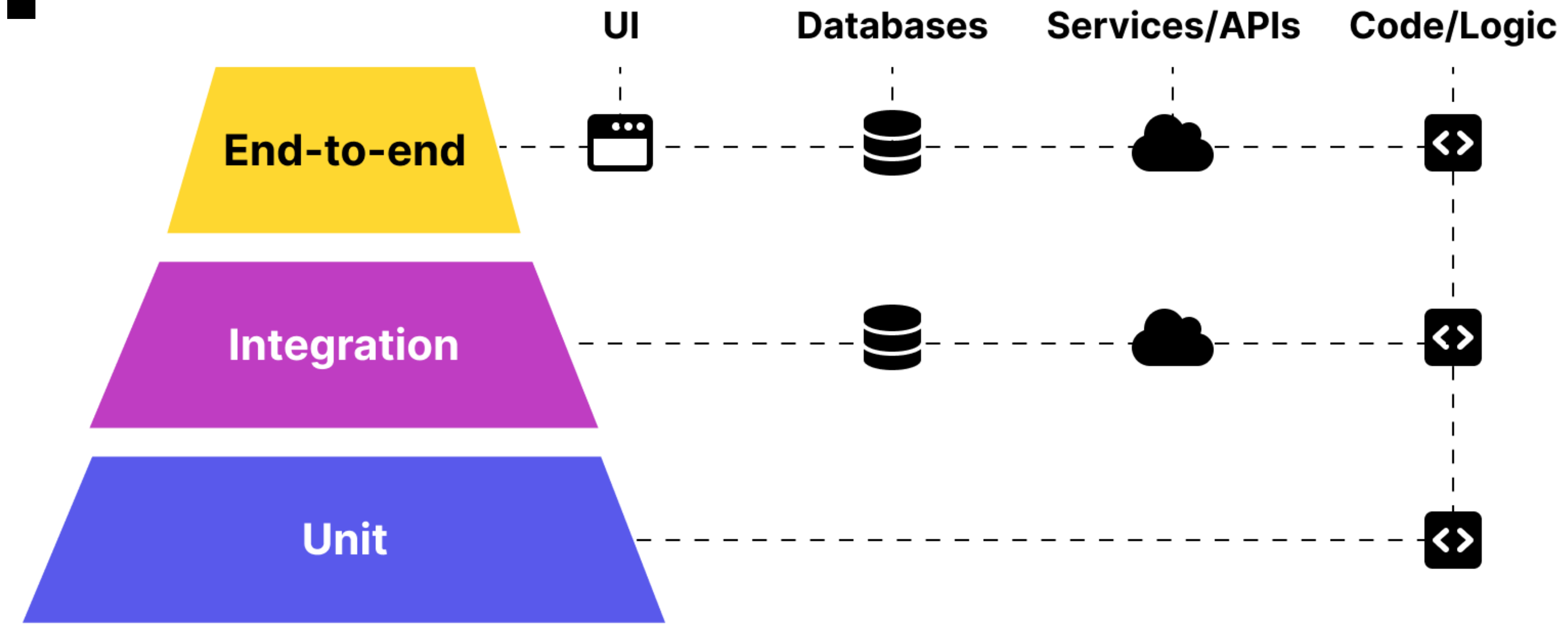
Unit Testing

High Level



https://katalon.com/resources-center/blog/unit-testing-vs-functional-testing

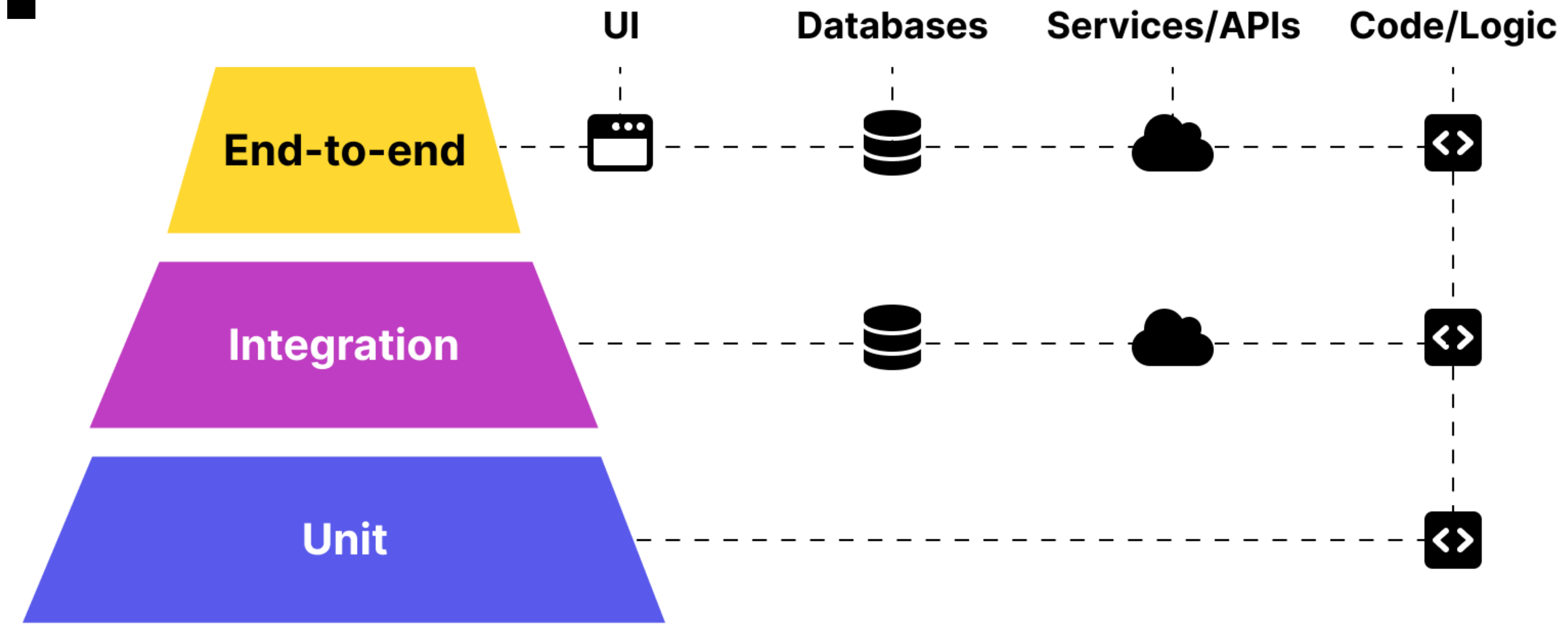
High Level



`unit tests` `verify our code on a collection of hand-chosen inputs`

https://katalon.com/resources-center/blog/unit-testing-vs-functional-testing

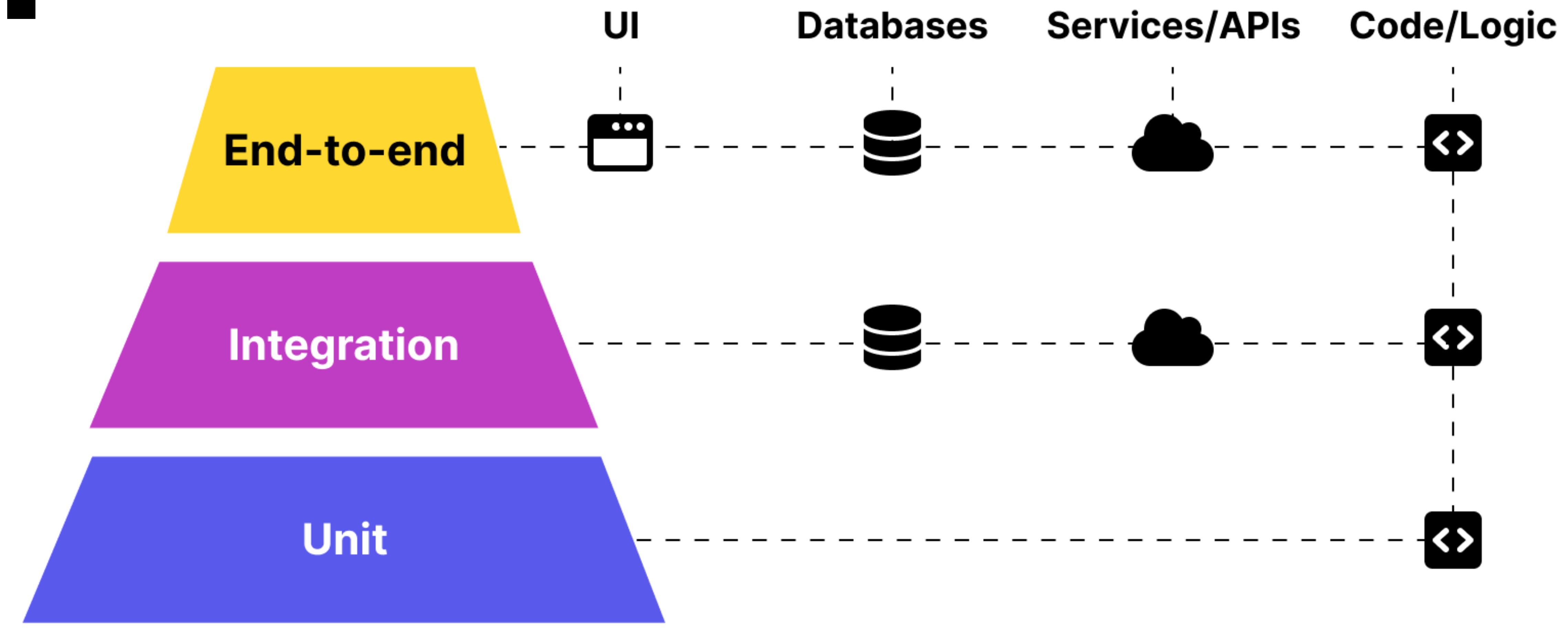
High Level



unit tests verify our code on a collection of hand–chosen inputs

It's the easiest kind of testing and the "first line of defense"

High Level



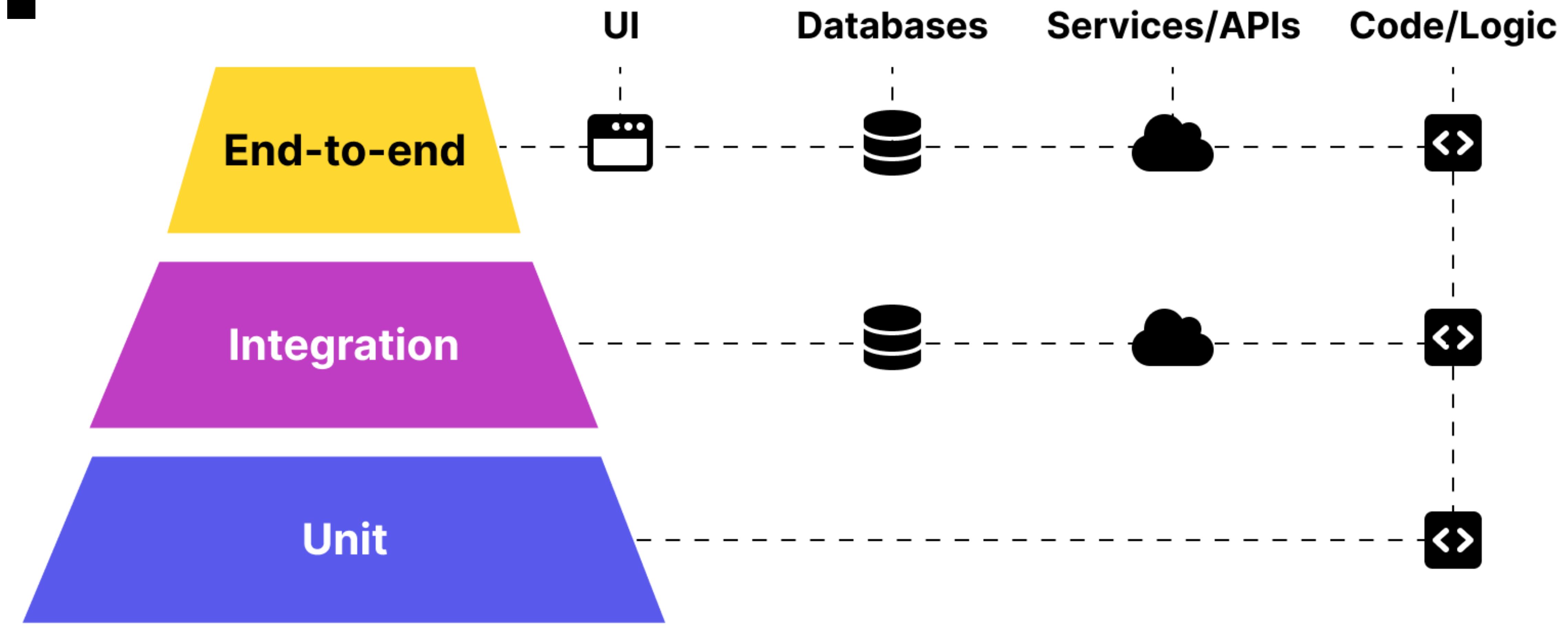
unit tests verify our code on a collection of hand-chosen inputs

It's the easiest kind of testing and the "first line of defense"

It's distinct from **fuzz testing** or **randomized testing** in which random inputs are checked

https://katalon.com/resources-center/blog/unit-testing-vs-functional-testing

High Level



unit tests verify our code on a collection of hand-chosen inputs

It's the easiest kind of testing and the "first line of defense"

It's distinct from **fuzz testing** or **randomized testing** in which random inputs are checked

It's also distinct from an area of CS called ***software verification***, which uses computers to *prove* that our programs are correct

https://katalon.com/resources-center/blog/unit-testing-vs-functional-testing

Words of Warning

Words of Warning

There are many unit testing frameworks out there. We'll use **OUnit2** which is very good, but by no means the most featured

Words of Warning

There are many unit testing frameworks out there. We'll use **OUnit2** which is very good, but by no means the most featured

This course is not about learning OUnit2, we just want to get familiar with the ideas

Words of Warning

There are many unit testing frameworks out there. We'll use **OUnit2** which is very good, but by no means the most featured

This course is not about learning OUnit2, we just want to get familiar with the ideas

We are already using OUnit2 for testing assignments and projects, so you need to know how to read and write tests in this framework

Set-up

```
(test
 (name test_PROG)
 (libraries ounit2))
```

Set-up

```
(test
 (name test_PROG)
 (libraries ounit2))
```

To use **OUnit2** for testing, we have to add it as a dependency of the tests in our **dune** project

Set-up

```
(test
 (name test_PROG)
 (libraries ounit2))
```

To use **OUnit2** for testing, we have to add it as a dependency of the tests in our **dune** project

*We will always do this for you in your projects, but it's good to know how to do for our own projects**

**I know no one is using OCaml for personal projects, but I can hope...*

Important Functions

`(>::)`	creates a labelled test
`(>:::)`	creates a labelled test suite
`assert_equal`	compares two values in a unit test
`assert_raises`	checks that an expression raises the right exception
`run_test_tt_main`	runs a test suite

Example Test Suite

```
let tests = "test suite for sum" >::: [
  "empty" >:: (fun _ -> assert_equal 0 (sum []));
  "singleton" >:: (fun _ -> assert_equal 1 (sum [1]));
  "two_elements" >:: (fun _ -> assert_equal 3 (sum [1; 2]));
]

let _ = run_test_tt_main tests
```

test/test_<PROG>.ml

Each test is given a name, the suite is given a name as well

Each test is wrapped in an anonymous function *(why?)*

Unit Testing with OUnit

Benefits:

- » Tests can be run in parallel
- » Failed tests don't block!

Downsides:

- » More code to read and write
- » Output is a bit difficult to read...

Modules

High Level

High Level

Modules attempt to capture multiple programming patterns with a single construct:

High Level

```
module Interpreter = struct
  let type_check = ...
  let eval = ...
end
```

(1)

Modules attempt to capture multiple programming patterns with a single construct:

1. **Namespaces:** a way of separate coding into logical units

High Level

Modules attempt to capture multiple programming patterns with a single construct:

1. **Namespaces:** a way of separate coding into logical units

2. **Abstraction/Encapsulation:** a way of abstracting away implementation details and organizing core functionality (e.g., of a data structure)

```
module Interpreter = struct
  let type_check = ...
  let eval = ...
end
```

(1)

```
module Stack = struct
  type 'a t = 'a list
  let push x s = x :: s
  let pop s = match s with
    | [] -> None
    | x :: xs -> Some (x, xs)
end
```

(2)

High Level

Modules attempt to capture multiple programming patterns with a single construct:

1. **Namespaces:** a way of separate coding into logical units

2. **Abstraction/Encapsulation:** a way of abstracting away implementation details and organizing core functionality (e.g., of a data structure)

3. **Code Reuse:** a way to write general code that can be instantiated in different settings

```
module Interpreter = struct
  let type_check = ...
  let eval = ...
end
```

(1)

```
module Stack = struct
  type 'a t = 'a list
  let push x s = x :: s
  let pop s = match s with
    | [] -> None
    | x :: xs -> Some (x, xs)
end
```

(2)

```
module VarSet = Set.Make(String)
module Context = Map.Make(String)
```

(3)

Structures

```
module Foo = struct
  let double (x : int) : int = x + x

  let is_whitespace (c : char) =
    List.mem c [' '; '\n'; '\t'; '\r']

  let version = 225

end
```

Structures

A **structure** is a collection of definitions used to define a **module**

```
module Foo = struct
  let double (x : int) : int = x + x

  let is_whitespace (c : char) =
    List.mem c [' '; '\n'; '\t'; '\r']

  let version = 225

end
```

Structures

A **structure** is a collection of definitions used to define a **module**

Structures are *not* first-class values, we *must* use the **module** keyword when defining a structure

```
module Foo = struct
  let double (x : int) : int = x + x

  let is_whitespace (c : char) =
    List.mem c [' '; '\n'; '\t'; '\r']

  let version = 225

end
```

Structures

A **structure** is a collection of definitions used to define a **module**

Structures are *not* first-class values, we *must* use the **module** keyword when defining a structure

We can put anything in a structure that we can put in a standalone .ml file (and vice versa, more on this later)

```
module Foo = struct
  let double (x : int) : int = x + x

  let is_whitespace (c : char) =
    List.mem c [' '; '\n'; '\t'; '\r']

  let version = 225

end
```

Signatures

```
module type FOO =
  sig
    val double : int -> int
    val is_whitespace : char -> bool
    val version : int
    exception MyException
  end
```

Signatures

A **signature** is a collection of *specifications*

```
module type FOO =
  sig
    val double : int -> int
    val is_whitespace : char -> bool
    val version : int
    exception MyException
  end
```

Signatures

A **signature** is a collection of *specifications*

A specification is a name together with a type

```
module type FOO =
  sig
    val double : int -> int
    val is_whitespace : char -> bool
    val version : int
    exception MyException
  end
```

Signatures

A **signature** is a collection of *specifications*

A specification is a name together with a type

A signature defines a **module type**

```
module type FOO =
  sig
    val double : int -> int
    val is_whitespace : char -> bool
    val version : int
    exception MyException
  end
```

Signatures

A **signature** is a collection of *specifications*

A specification is a name together with a type

A signature defines a **module type**

A module **implements** a signature if it's defined as a structure which has the values required by the signature

```
module type FOO =
  sig
    val double : int -> int
    val is_whitespace : char -> bool
    val version : int
    exception MyException
  end
```

General Syntax

```
module ModuleName : SIG_NAME = struct
  val val_name1 : ty
  val val_name2 : ty
  ...
end
```

```
module L = List
module S = String
```

General Syntax

```
module ModuleName : SIG_NAME = struct
  val val_name1 : ty
  val val_name2 : ty
  ...
end
```

```
module L = List
module S = String
```

Module names are usually CamelCase and module types in SCREAMING_SNAKE

General Syntax

```
module ModuleName : SIG_NAME = struct
  val val_name1 : ty
  val val_name2 : ty
  ...
end
```

```
module L = List
module S = String
```

Module names are usually CamelCase and module types in SCREAMING_SNAKE

The inner part of the **struct** is anything we could write in a **.ml** file

General Syntax

```
module ModuleName : SIG_NAME = struct
  val val_name1 : ty
  val val_name2 : ty
  ...
end
```

```
module L = List
module S = String
```

Module names are usually CamelCase and module types in SCREAMING_SNAKE

The inner part of the **struct** is anything we could write in a **.ml** file

The **module** keyword is like the **let** keyword except that the RHS of the "=" must be a structure *or another module*

General Syntax

```
module ModuleName : SIG_NAME = struct
  val val_name1 : ty
  val val_name2 : ty
  ...
end
```

```
module L = List
module S = String
```

Module names are usually CamelCase and module types in SCREAMING_SNAKE

The inner part of the **struct** is anything we could write in a **.ml** file

The **module** keyword is like the **let** keyword except that the RHS of the "=" must be a structure *or another module*

Trick: We can write shorthand names for module names we use frequently

Signature Inference and Interface Files

```
let double (x : int) : int = x + x

let is_whitespace (c : char) =
    List.mem c [' '; '\n'; '\t'; '\r']

let version = 225

let helper (x : int) : int = ……
```

foo.ml

```
val double : int -> int

val is_whitespace : char -> bool


val version : int
```

foo.mli

Signature Inference and Interface Files

```ocaml
let double (x : int) : int = x + x

let is_whitespace (c : char) =
    List.mem c [' '; '\n'; '\t'; '\r']

let version = 225

let helper (x : int) : int = ……
```

foo.ml

```ocaml
val double : int -> int

val is_whitespace : char -> bool


val version : int
```

foo.mli

In most cases, OCaml infers the signature of a given module (the annotation is optional)

Signature Inference and Interface Files

```
let double (x : int) : int = x + x

let is_whitespace (c : char) =
    List.mem c [' '; '\n'; '\t'; '\r']

let version = 225

let helper (x : int) : int = ……
```

foo.ml

```
val double : int -> int

val is_whitespace : char -> bool


val version : int
```

foo.mli

In most cases, OCaml infers the signature of a given module (the annotation is optional)

In fact, we've been defining modules the entire time: *every file defines a module, whose name is the same as the filename (capitalized)*

Signature Inference and Interface Files

```ocaml
let double (x : int) : int = x + x

let is_whitespace (c : char) =
    List.mem c [' '; '\n'; '\t'; '\r']

let version = 225

let helper (x : int) : int = ……
```

foo.ml

```ocaml
val double : int -> int

val is_whitespace : char -> bool


val version : int
```

foo.mli

In most cases, OCaml infers the signature of a given module (the annotation is optional)

In fact, we've been defining modules the entire time: *every file defines a module, whose name is the same as the filename (capitalized)*

We can make signatures of files explicit with **.mli** files

Working with Modules

```
module type FOO =
  sig
    val double : int -> int
    val is_whitespace : char -> bool
    val version : int
    exception MyException
  end
```

```
let check c =
  if Foo.version > 300 && Foo.is_whitespace c
  then "okay"
  else "not okay"
```

Once a module is defined, we can use values defined therein by **dot notation**

(This should feel somewhat familiar, again, we've been working with modules this whole time)

Opening Modules

```
open Foo

let check c =
  if version > 300 && is_whitespace c
  then "okay"
  else "not okay"
```

Opening Modules

```ocaml
open Foo

let check c =
  if version > 300 && is_whitespace c
  then "okay"
  else "not okay"
```

We can bring all definitions in a module into scope with the **open** keyword

Opening Modules

```
open Foo

let check c =
  if version > 300 && is_whitespace c
  then "okay"
  else "not okay"
```

We can bring all definitions in a module into scope with the **open** keyword

Caution: Do this sparingly, it's like **import** * except worse because there's no overloading in OCaml

Opening Modules

```
open Foo

let check c =
  if version > 300 && is_whitespace c
  then "okay"
  else "not okay"
```

We can bring all definitions in a module into scope with the **open** keyword

Caution: Do this sparingly, it's like **import *** except worse because there's no overloading in OCaml

If there are multiple definition of the function, the most recent open prevails

.(...) Syntax

```
let check c =
  Foo.(if version > 300 && is_whitespace c
       then "okay"
       else "not okay")
```

It's possible to parenthesize expressions after the dot notation!

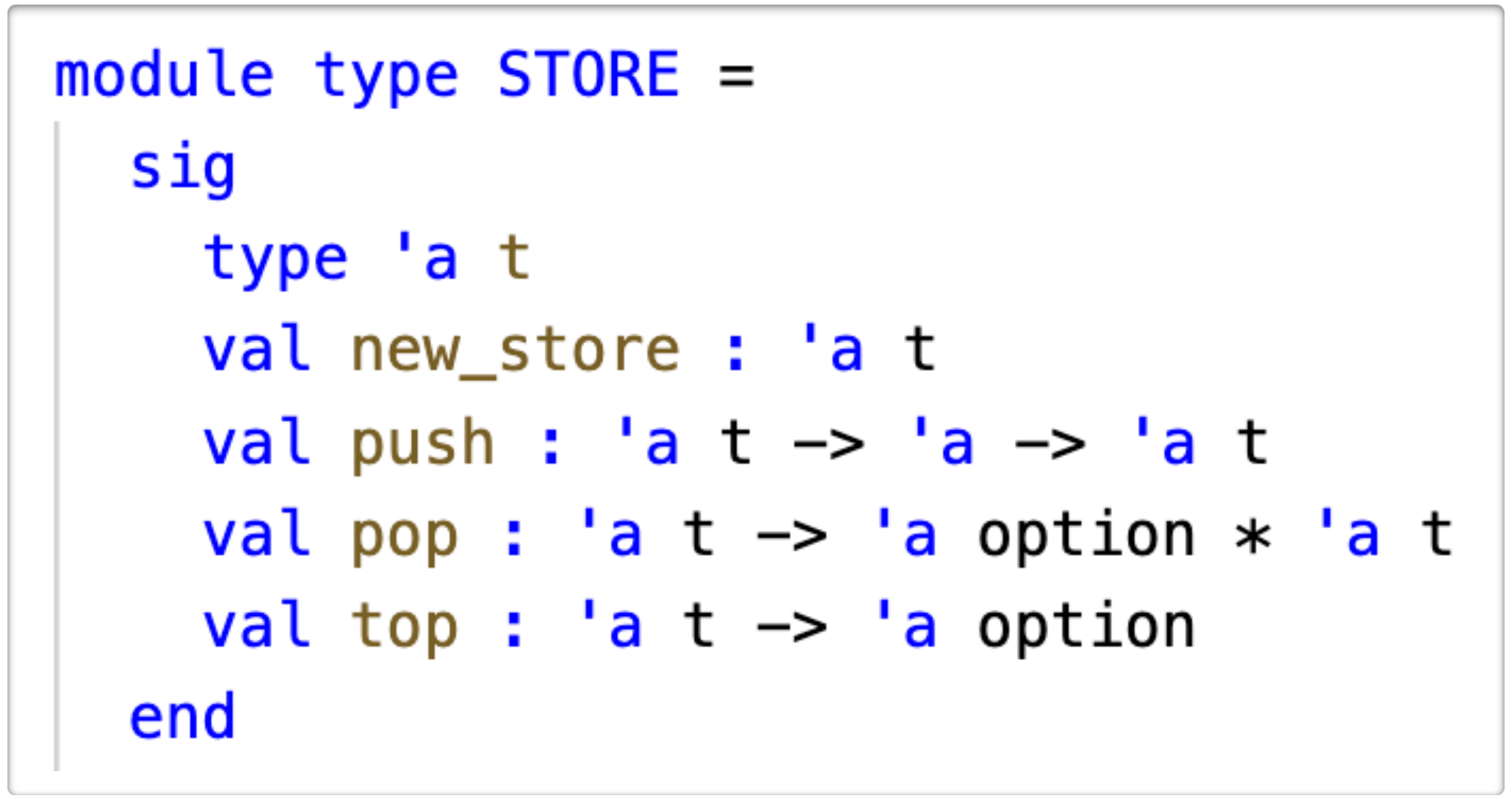
This will evaluate the expression *as if* the module was opened

Functional Data Structures

Interfaces for Functional Data Structures

```
module type STORE =
  sig
    type 'a t
    val new_store : 'a t
    val push : 'a t -> 'a -> 'a t
    val pop : 'a t -> 'a option * 'a t
    val top : 'a t -> 'a option
  end
```

Interfaces for Functional Data Structures



```
module type STORE =
  sig
    type 'a t
    val new_store : 'a t
    val push : 'a t -> 'a -> 'a t
    val pop : 'a t -> 'a option * 'a t
    val top : 'a t -> 'a option
  end
```

So we can define modules which expose an abstract interface, *without* exposing the data representation

Interfaces for Functional Data Structures

```ocaml
module type STORE =
  sig
    type 'a t
    val new_store : 'a t
    val push : 'a t -> 'a -> 'a t
    val pop : 'a t -> 'a option * 'a t
    val top : 'a t -> 'a option
  end
```

So we can define modules which expose an abstract interface, *without* exposing the data representation

This allows us to "swap out" our store type without affecting any code which depends on the module

Interfaces for Functional Data Structures

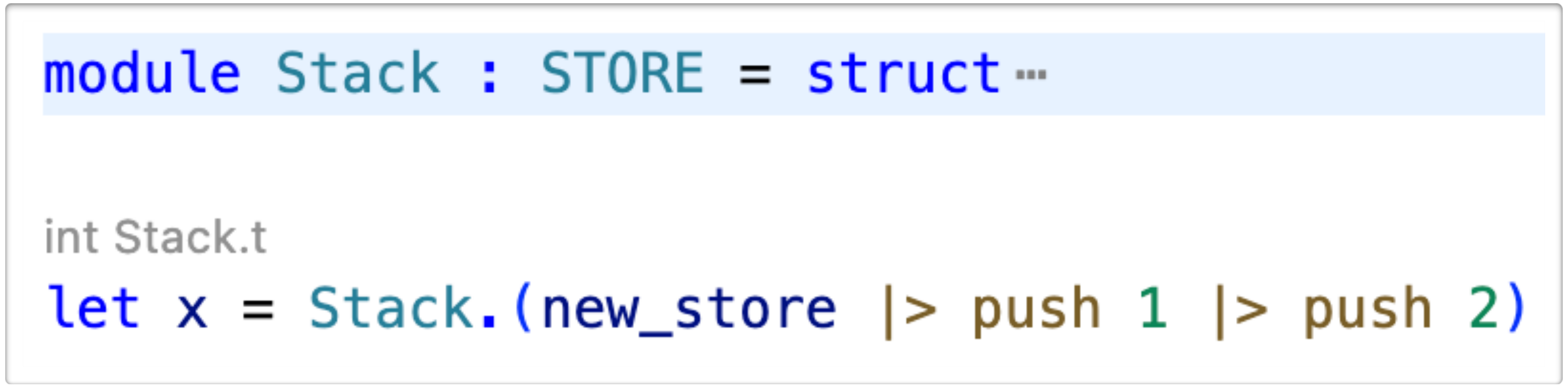
```
module type STORE =
  sig
    type 'a t
    val new_store : 'a t
    val push : 'a t -> 'a -> 'a t
    val pop : 'a t -> 'a option * 'a t
    val top : 'a t -> 'a option
  end
```

So we can define modules which expose an abstract interface, *without* exposing the data representation

This allows us to "swap out" our store type without affecting any code which depends on the module

This is just good abstraction: don't expose the low-level details unless it's necessary

Abstract Types are Opaque



```
module Stack : STORE = struct…


int Stack.t
let x = Stack.(new_store |> push 1 |> push 2)
```

We can't make *any* assumptions about an abstract type if we don't expose it

Our code must still work if the abstract type changes

Important: This is not OOP

```
module type STORE =
  sig
    type 'a t
    val new_store : 'a t
    val push : 'a t -> 'a -> 'a t
    val pop : 'a t -> 'a option * 'a t
    val top : 'a t -> 'a option
  end
```

A module is not the same thing as a class, from which objects are instantiated (i.e., there is no **new** constructor)

Functions in structures are not *methods* of a given type of object

(and there's still no mutability)

demo

(modules of stores)

Summary

We can test our code with **unit test** frameworks like OUnit2

We can **encapsulate** data and define **interfaces** for types or data structures all with the same construct

When we write code in a file, we're building a module