Error Handling and Testing

Concepts of Programming Languages
Lecture 9

CAS CS 320

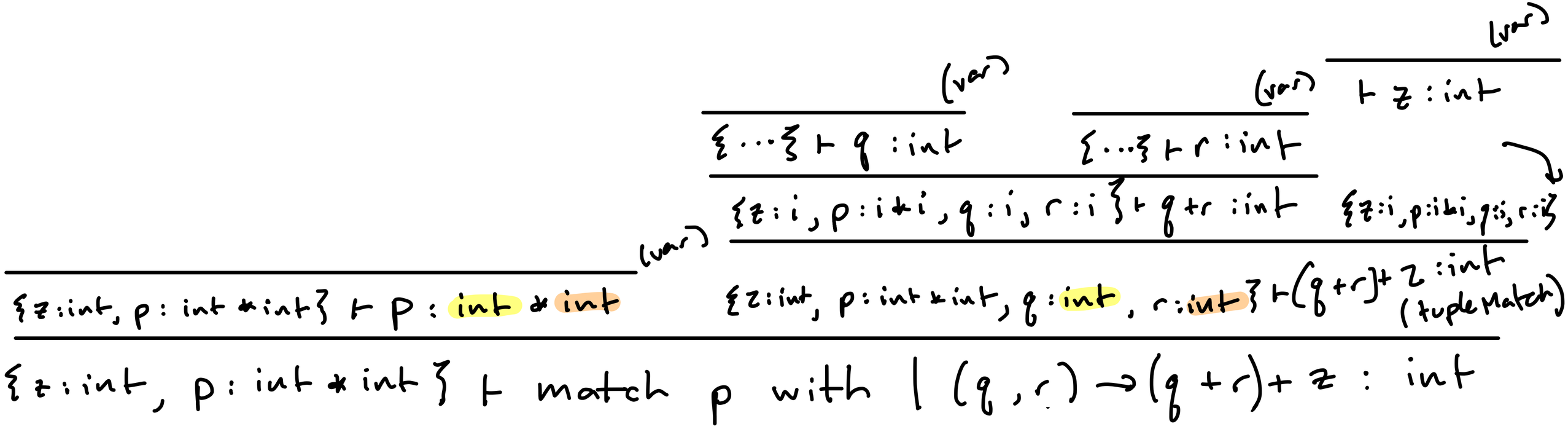
Outline

- Look more carefully at **options** and **results**
- See how **monads** (i.e., **option chaining**) can make working with options more convenient
- Introduce **exceptions** as a way of handling errors unsafely
- Give a short tutorial on **ounit** a unit test framework for OCaml

Practice Problem

$\Gamma \vdash$ `match p with | (x, y) ->(x + y)+ z` : τ

Determine the smallest context Γ *and the type* τ *such that the above judgment is derivable. Then give the derivation*


(var)
⊢ z : int
(var)
{…} ⊢ r : int
(var)
{…} ⊢ q : int
{z : i, p : i * i, q : i, r : i} ⊢ q + r : int
{z:i, p:i*i, q:i, r:i}
{z:int, p: int * int} ⊢ p : int * int
(var)
{z:int, p : int * int, q : int, r : int} ⊢ (q + r) + z : int
(tupleMatch)
{z : int, p : int * int} ⊢ match p with | (q , r) → (q + r) + z : int

Options and Results

Recall: Options

```
type 'a option =
  | None
  | Some of 'a
```

option is a polymorphic (parametrized) variant type with two constructors

An option is like a "box" that may or may not contain a value

This is useful for defining **partial functions**

Example

```
let rec first_such_that (p : 'a -> bool) (l : 'a list) : 'a option =
  match l with
  | [] -> None
  | x :: xs -> if p x then Some x else first_such_that p xs
```

Example

```
let rec first_such_that (p : 'a -> bool) (l : 'a list) : 'a option =
  match l with
  | [] -> None
  | x :: xs -> if p x then Some x else first_such_that p xs
```

Suppose we want to determine the first element in a list which satisfies a given predicate

Example

```
let rec first_such_that (p : 'a -> bool) (l : 'a list) : 'a option =
  match l with
  | [] -> None
  | x :: xs -> if p x then Some x else first_such_that p xs
```

Suppose we want to determine the first element in a list which satisfies a given predicate

It's not guaranteed that *any* element in a list satisfies the predicate. *What should we return in this case?*

Example

```
let rec first_such_that (p : 'a -> bool) (l : 'a list) : 'a option =
  match l with
  | [] -> None
  | x :: xs -> if p x then Some x else first_such_that p xs
```

Suppose we want to determine the first element in a list which satisfies a given predicate

It's not guaranteed that *any* element in a list satisfies the predicate. *What should we return in this case?*

If we return an option we can return **None**.

Important (Again)

Important (Again)

None is *not* the same as a null pointer or a null value

Important (Again)

None is *not* the same as a null pointer or a null value

None is a piece of data in the same way that **2** or **true** is

Important (Again)

None is *not* the same as a null pointer or a null value

None is a piece of data in the same way that **2** or **true** is

This means the type-checker *forces* us to handle the undefined case of partial functions

No Remembering Null Checks

```
void fill_foo(int* foo) {
    if (!foo) { // this is the NULL check
        printf("This is wrong\n");
        return;
    }
    *foo = 23;
}
```

https://softwareengineering.stackexchange.com/questions/151867/what-does-it-mean-to-do-a-null-check-in-c-or-c

No Remembering Null Checks

```
void fill_foo(int* foo) {
    if (!foo) { // this is the NULL check
        printf("This is wrong\n");
        return;
    }
    *foo = 23;
}
```

We don't have to *remember* to check in our value is null before doing our computation

https://softwareengineering.stackexchange.com/questions/151867/what-does-it-mean-to-do-a-null-check-in-c-or-c

No Remembering Null Checks

```
void fill_foo(int* foo) {
    if (!foo) { // this is the NULL check
        printf("This is wrong\n");
        return;
    }
    *foo = 23;
}
```

We don't have to *remember* to check in our value is null before doing our computation

If we don't do a "none check", then our code won't even pass the type checker

https://softwareengineering.stackexchange.com/questions/151867/what-does-it-mean-to-do-a-null-check-in-c-or-c

Recall: Results

```
type ('a, 'b) result =
  | Ok of 'a
  | Error of 'b
```

A **result** is the same as an option, except that we carry data in the "None" case

Results play the same role as options, except they may help with things like error messages

Example

```
type error = LessThanOne | MoreThanOne

let exactly_one (f : 'a -> bool) (l : 'a list) : ('a, error) result =
...
```

Example

```
type error = LessThanOne | MoreThanOne

let exactly_one (f : 'a -> bool) (l : 'a list) : ('a, error) result =
...
```

Suppose we want to find the *only* element in a list that satisfies a given property

Example

```
type error = LessThanOne | MoreThanOne

let exactly_one (f : 'a -> bool) (l : 'a list) : ('a, error) result =
...
```

Suppose we want to find the *only* element in a list that satisfies a given property

But we also want to know why we failed...

Example

```
type error = LessThanOne | MoreThanOne

let exactly_one (f : 'a -> bool) (l : 'a list) : ('a, error) result =
...
```

Suppose we want to find the *only* element in a list that satisfies a given property

But we also want to know why we failed...

Results allow use to return more information in the failed case

Practice Problem

Implement the function

```
val exactly_one : ('a -> bool)
                  -> 'a list
                  -> ('a, error) result
```

according to the idea from the previous slide

Higher Order Functions and Options/Results

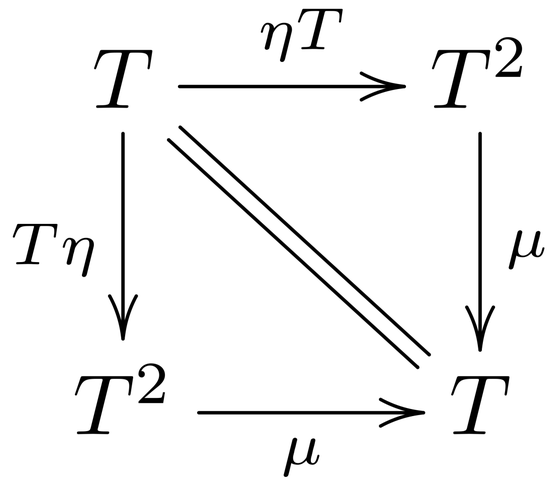
```
let rec exactly n f l =
  match n, l with
  | 0, [] -> Some []
  | n, [] -> None
  | n, x :: xs ->
    if n < 0
    then None
    else if f x
    then Option.map (fun xs -> x :: xs) (exactly (n - 1) f xs)
    else exactly n f xs
```

A reminder that **map** is a very useful function for error handling

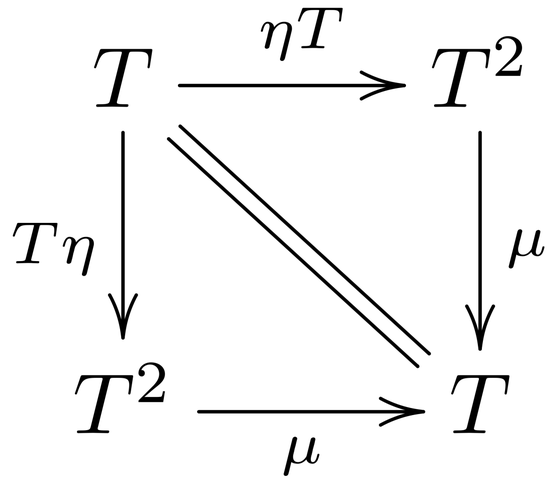
It allows us to work with the output of a recursive call as if it was an "actual" value

demo
(matrices)

Advanced: Monads

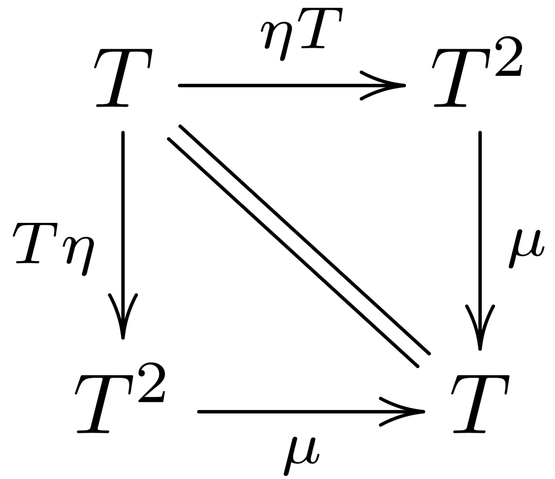


Advanced: Monads



Roughly speaking, a **monad** is a structure that "holds a thing". This includes **options** and **results**, but also **lists**

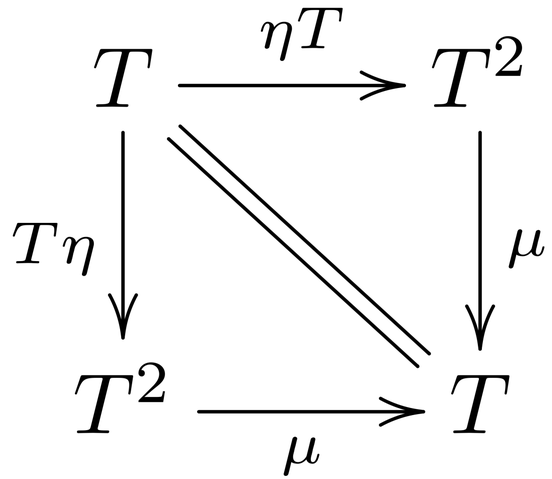
Advanced: Monads



Roughly speaking, a **monad** is a structure that "holds a thing". This includes **options** and **results**, but also **lists**

The key feature of a monad is the ability to **sequence** operations

Advanced: Monads



Roughly speaking, a **monad** is a structure that "holds a thing". This includes **options** and **results**, but also **lists**

The key feature of a monad is the ability to **sequence** operations

(see the textbook for more details, this is an idea that comes from a field of mathematics called *category theory*)

Aside: Option Chaining

```
let user = {}; // user has no address
alert( user?.address?.street ); // undefined (no error)
```

Aside: Option Chaining

```
let user = {}; // user has no address
alert( user?.address?.street ); // undefined (no error)
```

Option chaining is a feature used in languages like javascript for *sequencing* functions on a possibly nonexistent object without having to explicitly check

Aside: Option Chaining

```
let user = {}; // user has no address
alert( user?.address?.street ); // undefined (no error)
```

Option chaining is a feature used in languages like javascript for *sequencing* functions on a possibly nonexistent object without having to explicitly check

This is a pattern based on monads (taken from FP)

Bind a.k.a. "And Then"

```
let bind (x : 'a option) (f : 'a -> 'b option) : 'b option =
  match x with
  | None -> None
  | Some x -> f x
```

Bind a.k.a. "And Then"

```
let bind (x : 'a option) (f : 'a -> 'b option) : 'b option =
  match x with
  | None -> None
  | Some x -> f x
```

The core function of a monad is **bind**

Bind a.k.a. "And Then"

```
let bind (x : 'a option) (f : 'a -> 'b option) : 'b option =
  match x with
  | None -> None
  | Some x -> f x
```

The core function of a monad is **bind**

The idea is simple: *match, and then apply.* This gives us a cleaner way of doing sequences of pattern matches

Bind a.k.a. "And Then"

```
let bind (x : 'a option) (f : 'a -> 'b option) : 'b option =
  match x with
  | None -> None
  | Some x -> f x
```

The core function of a monad is **bind**

The idea is simple: *match, and then apply.* This gives us a cleaner way of doing sequences of pattern matches

We can think of this as "try to unwrap **x** **and then** do **f**"

Bind a.k.a. "And Then"

```
let bind (x : 'a option) (f : 'a -> 'b option) : 'b option =
  match x with
  | None -> None
  | Some x -> f x
```

The core function of a monad is **bind**

The idea is simple: *match, and then apply.* This gives us a cleaner way of doing sequences of pattern matches

We can think of this as "try to unwrap **x** **and then** do **f**"

(We have an analogous version for results as well)

demo

(assoc. lists w/ unique keys)

let* Syntax

```
let foo =
  let ( let* ) = Option.bind in
  let* x = ox in
  let* y = oy in
  let* z = oz in
  Some (x + y + z)
```

```
let bar =
  bind ox (fun x ->
  bind oy (fun y ->
  bind oz (fun z ->
  Some (x + y + z))))
```

```
let baz =
  match ox with
  | None -> None
  | Some x -> (
    match oy with
    | None -> None
    | Some y -> (
      match oz with
      | None -> None
      | Some z -> Some (x + y + z)
    )
  )
```

let* syntax allows use to shorthand the use of the **bind** function, which allows us to avoid sequences of (annoying) pattern matches

Again, this is exactly option chaining

Disclaimer: It is never required that you use monads in this course. That said, they're very cool

demo

(interpreters)

Important: Eager evaluation

```
let foo =
  match ox, oy with
  | Some x, Some y -> Some (x + y)
  | _ -> None
```

```
let foo =
  match ox with
  | None -> None
  | Some x -> (
    match oy with
    | None -> None
    | Some y -> Some (x + y)
  )
```

OCaml is **eagerly evaluated**

This means that arguments to functions/match expressions are *completely* evaluated before the function is called/match is checked

Exceptions

Raising Exceptions

```python
if f x:
  raise Exception("error message")
else:
  return 2
```

Python

```ocaml
exception MyError

let foo x =
  if f x
  then raise MyError
  else 2
```

OCaml

Like most languages, OCaml has a mechanism for programmatically *crashing* the program

Sometimes, we want this to happen. *Not all errors should be handled gracefully*

Data-Carrying Exceptions

```
exception A
exception B
exception Code of int
exception Details of string
```

Exceptions can carry data, just like constructors of a variant type

This data may be useful when trying to *handle* the exception instead of letting it crash the program

Exceptions and Pattern Matching

```
let hd l =
  match l with
  | [] -> Failure "hd called on empty list"
  | x :: _ -> x
```

raise

```
let hd_opt l =
  match hd l with
  | x -> Some x
  | exception (Failure _) -> None
```

OCaml has **try-style error-handling** (see the textbook for details) but it is far common in modern OCaml to **pattern match** on exceptions

Exceptions and Evaluation Order

```
(raise A, raise B)
```

≠

```
let a = raise A in
let b = raise B in
(a, b)
```

One issue with exceptions is that understanding their behavior requires understanding the *order* in which arguments are evaluated

The order in which elements in a tuple are **is not specified by OCaml**

Practice Problem

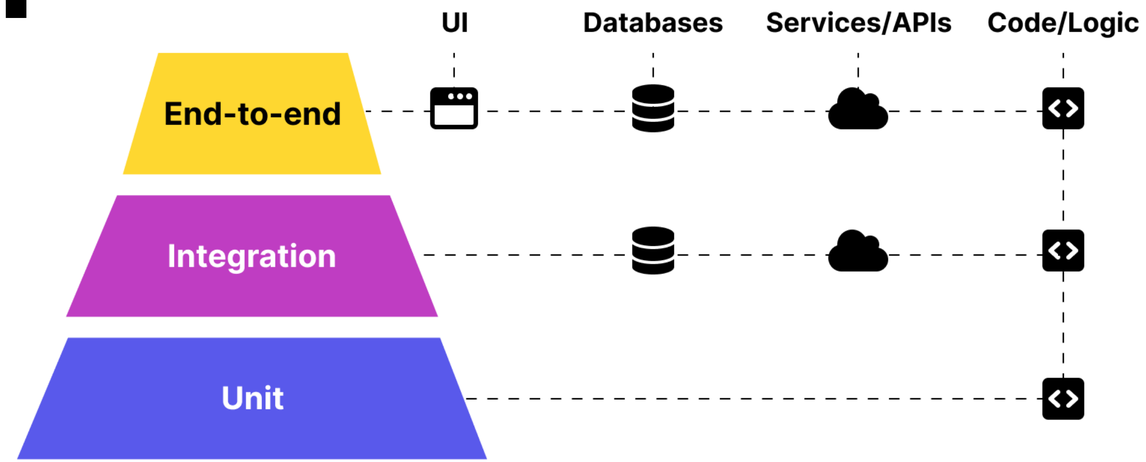
Reimplement the function

```
val exactly_one : ('a -> bool)
                  -> 'a list
                  -> 'a
```

so that it raises an exception if there is no element in the list satisfied by the given predicate

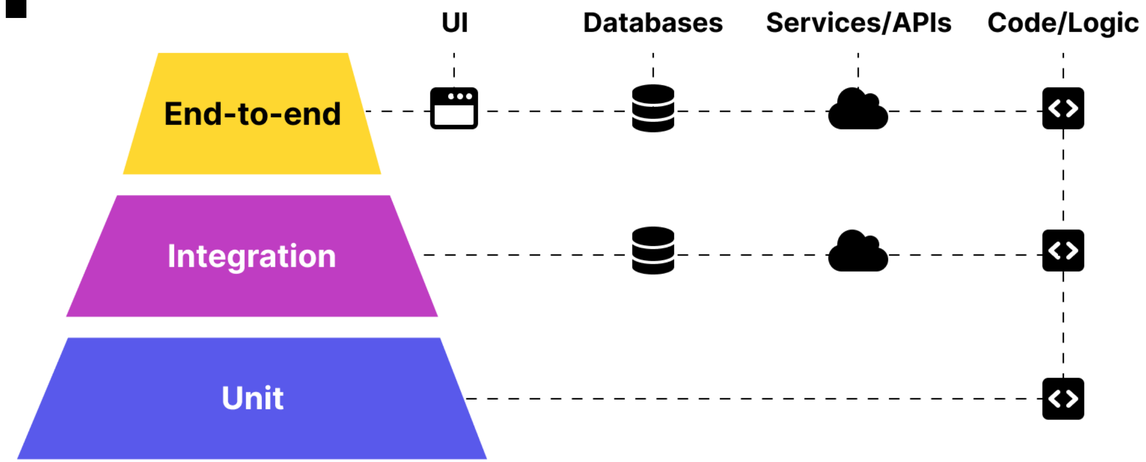
Unit Testing

High Level



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

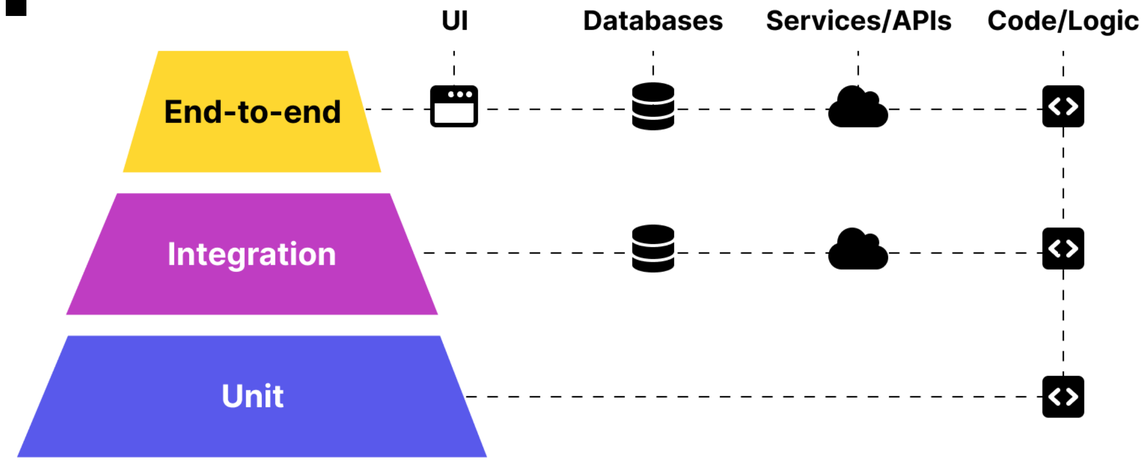
High Level



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

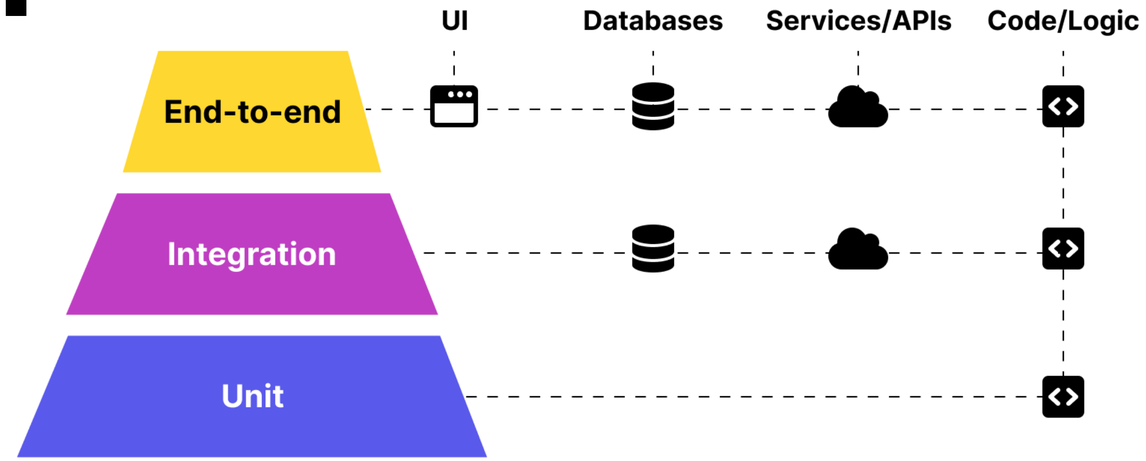


unit tests verify our code on a collection of hand-chosen inputs

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

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

High Level



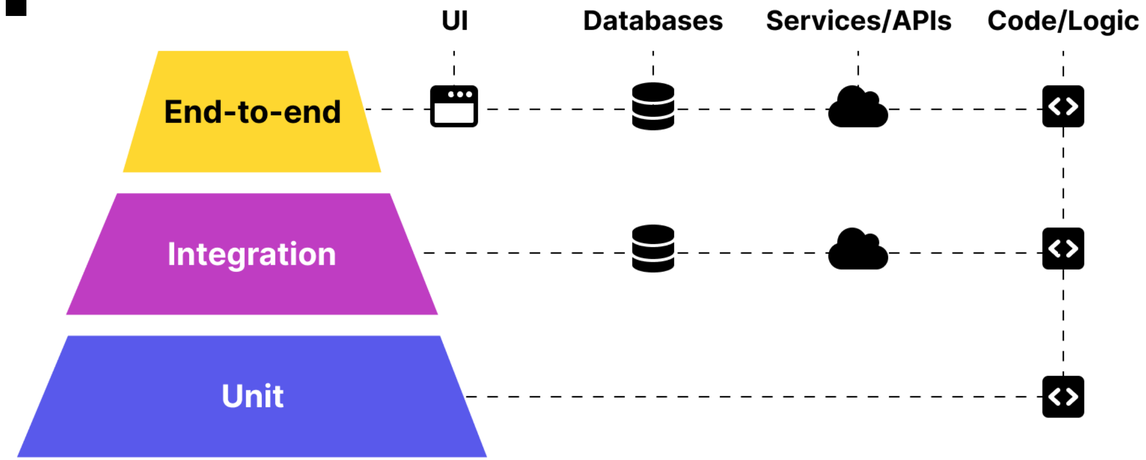
unit tests verify our code on a collection of hand-chosen inputs

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

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 call ***software verification***, in which we 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 how to use OUnit, 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 how to use OUnit, we just want to get familiar with the ideas

We will use OUnit from now on for 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 **OUnit** 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 **OUnit** 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...

demo

(testing exactly_one)

Summary

Error handling can be done safely with types and unsafely with exceptions

Either way we can test with unit test frameworks like OUnit

FP has a lot of beautiful mechanisms for working with complex types like options and results