Programming Paradigms

Session 1: Introduction

Léon Gondelman

September 2, 2026

Aalborg University @ Copenhagen

Sum of integers

Let us start with a little exercise:

Write a program that computes the **sum of integers** from 1 up to some integer ***n*** given by user.

Use a programming language of **your choice** (or write it on paper if you didn't bring your computer).

$$\sum_{i=1}^{n} a_i$$

Sum of integers

Let's take a look:

- Which language did you use?
- Did you use loops?
- Did you use recursion?
- Another way?

Let's see some possible solutions together:

- Java, Python, OCaml
- loops, recursion, tail-rec

Language		Style/Features	
Python	6	Loop	21
Java	4	Recursion	1
C	4	Closed form	4
Js	2		
Pseudocode	2		
F#	1		

Paradigm is a choice inside a language

There are no functional languages, only languages with better defaults

	imperative by default	persistent by default
statically typed	Java · C# · Go/Elixir · Rust	OCaml · Haskell
dynamically typed	Python · JavaScript	Clojure · Erlang

OCaml has mutation · Python has immutable tuples · Java 21 has algebraic data types · Rust is typed and imperative and safe

This is not a course about OCaml

…or Haskell, or Go, or Rust, or Erlang.

Double every element — Use iteration/higher-order function as the pattern — Different Languages

OCaml	`List.map (fun x -> 2*x) xs`
Python	`[2*x for x in xs]`
JS	`xs.map(x => 2*x)`
Java	`xs.stream().map(x -> 2*x).toList()`
Rust	`xs.iter().map(|x| 2*x).collect()`

- **A language is a set of defaults. A paradigm is a set of decisions.**
- **Defaults change every five years. The decisions don't.**

Languages in this course: three tiers

Rule: we try to to see each concept in at least two languages

WRITE

OCaml — the new one · Python · JavaScript / Java — the ones you know

Labs are programmed in these. OCaml is taught only as far as each session needs.

PICK UP

Go/Elixir · SQL · Haskell · …

One lab each, learned/experimented on the fly from a cheat sheet/lab instructions/small tutorials

OBSERVE

Rust · Erlang · Haskell · Clojure · Julia · …

Read in lectures as specimens of a concept. Never examined on syntax.

The course in three parts

12 sessions, Wednesdays · lecture 12:30–14:15 · lab 14:15–16:15

Part 1 · Data & functions

S2 Persistence

S3 Algebraic data types

S4 Functional programming & λ-calculus

S5 Higher-order functions & iterators

Part 2 · Concurrency

S6 Asynchronous programming

S7 Multi-threading

S8 Message passing

Part 3 · Types & abstraction

S9 Static type systems

S10 Polymorphism

S11 Data abstraction

S12 Declarative programming

The course in three parts

12 sessions, Wednesdays · lecture 12:30–14:15 · lab 14:15–16:15

Part 1 · Data & functions

S2 Persistence

S3 Algebraic data types

S4 Functional programming & λ-calculus

S5 Higher-order functions & iterators

Part 2 · Concurrency

S6 Asynchronous programming

S7 Multi-threading

S8 Message passing

Part 3 · Types & abstraction

S9 Static type systems

S10 Polymorphism

S11 Data abstraction

S12 Declarative programming

How this course works

Concepts over languages — and you learn them by building

Programs with holes

- Lab hand you a large, working program with pieces missing
- The missing pieces are exactly that week's concept; tests guide you; something cool happens when they pass

Several languages at once

- The same abstraction in two or three languages, from one language-independent description
- The comparison is the lesson: what did each language check, and what did it merely trust?

Pen & paper too

- λ-calculus reductions, typing derivations, heap diagrams, amortization arguments
- Some ideas are clearer without a compiler in the way

Labs: 2 hours + homework

- In class: make the core tests pass. At home: stretch goals and a half-page reflection
- Reference solutions published every week — nobody is stuck for a month

Part 1 · Data & functions

What

S2 Persistence — data that never changes, versions for free

S3 Algebraic data types — model data so illegal states can't exist

S4 Functional programming — recursion, evaluation order, the λ-calculus

S5 Higher-order functions — closures, fold, iterators, generators

Why start here

- Mutable state is the most common source of bugs you have met so far — we make it a choice instead of a default
- Immutability buys undo, history, and safe sharing — Part 2 will bring that in
- ADTs and pattern matching quietly became mainstream: Java 21, Python 3.10, TypeScript, Rust
- The λ-calculus is the smallest complete language; understand it and closures, laziness and evaluation strategies stop being magic
- Higher-order functions are how modern code is shaped: streams, React, pandas, LINQ

You will have built: a persistent queue driving a maze solver, a red-black tree, and your own λ-interpreter

Part 2 · Concurrency

What

S6 Asynchronous programming — promises, async/await, the event loop

S7 Multi-threading — shared memory, data races, locks (and Rust's answer)

S8 Message passing — channels in Go/Elixir, actors in Erlang

Why three models, not one

- Every program you will ship is concurrent: a UI, a server, anything doing I/O
- The three models make different promises and fail differently — single-threaded interleaving, shared memory, share-nothing
- async/await is a compiler trick you will be able to explain after S4 and S5 (it is continuations and closures)
- This is where Part 1 pays off: immutable data can be shared across threads without locks
- OCaml steps aside on purpose: the concepts live in JavaScript, Java, Go, Erlang — proof they are not tied to a language

You will have built: e.g. your own event loop on generators, and a concurrent crawler in Go/Elixir

Part 3 · Types & abstraction

What

S9 Static type systems — what a compiler can prove before running

S10 Polymorphism — generics, type inference, subtyping, the expression problem

S11 Data abstraction — modules, interfaces, information hiding

S12 Declarative programming — SQL, spreadsheets, logic; and conclusion

Why end here

- A type is a theorem the compiler checks for you — the cheapest correctness guarantee in software
- Polymorphism is reuse without giving up safety; inference is getting it without writing types
- Abstraction boundaries are how teams scale: change the inside without breaking the clients
- Declarative programming moves the 'how' into an engine — SQL planners, spreadsheets, React — the endpoint of the whole course
- Questions planted in Parts 1–2 get answered here: the variants-vs-objects matrix, the .mli files, unification

You will have built: a type checker and type inference for your own λ-language — MiniML

Sum of integers: five answers, one course

Same problem, five programs — what did each choice buy, and what did it cost?

`a loop and a counter`	→ **state & mutation**	S2 · S7
`sum n = n + sum (n-1)`	→ **functions & recursion**	S4
`an accumulator, tail-recursive`	→ **evaluation & the stack**	S4
`fold / reduce over 1..n`	→ **higher-order functions**	S5
`n(n+1)/2 — say what, not how`	→ **declarative**	S12

…and five answers you will understand by December

Same problem. Not a loop, not a recursion.

Haskell	`scanl (+) 0 [1..] !! n`	*an infinite list — lazy evaluation*	**S4**
Prolog	`?- sigma(N, 55).  N = 10`	*a relation, so it runs backwards — logic*	**S12**
Erlang	`spawn(fun() -> Me ! {value, I} end)`	*one process per number — message passing*	**S8**
SQL	`SELECT SUM(i) FROM generate_series(1, n)`	*say what, not how — declarative*	**S12**
Lean	`theorem : 2 * sigma n = n * (n + 1)`	*a machine-checked proof — types as theorems*	**S9**

Files for all of these (plus APL, Forth, Smalltalk, C++ templates, Fortran, Rust…) will be on Moodle — read them for fun, not for the exam.

Why learn new paradigms at all?

You already program — four reasons that survive contact with real projects

Reasoning

- A pure function is an equation: same inputs, same output, nothing hidden
- Code you can reason about is code you can test, refactor, and trust

Concurrency

- Data that never changes can be shared by many threads with no locks
- Most concurrency bugs are mutation bugs wearing a disguise

The mainstream moved

- Java 21: records, sealed types, pattern matching · Python: match · JS: map/filter, React
- Rust: ownership as a type discipline — the ideas of this course, shipped

Judgement

- No paradigm is best; each buys something and costs something
- The exam asks you to justify a design choice — that skill is the point

What we deliberately leave out

Metaprogramming & macros	a great topic; costs a session and teaches less judgement than what is in
Garbage collection internals	we use it every week; we do not open it
Security as a subject	only micro-doses where a paradigm is the fix: ownership, encapsulation, injection
Array & scientific computing	a real paradigm (NumPy, Julia, JAX) — available as a mini-project instead
Learning any language 'properly'	not the goal; three languages seriously, several lightly, all comparatively

…and five answers we will not cover — on purpose

These are real paradigms too; they are just not ours this course.

APL	`+/⍳⍵`	*whole arrays at once — array programming*	—
Forth	`: sigma  dup 1+ * 2/ ;`	*no variables, only a stack — concatenative*	—
C++20	`static_assert(Sigma<10>::value == 55);`	*computed by the compiler — metaprogramming*	—
Mathematica	`Sum[i, {i, 1, n}]  ⟶  n(n+1)/2`	*an answer with n still in it — symbolic rewriting*	—
Bash	`seq 1 10 | paste -sd+ - | bc`	*small programs talking through text — pipes & filters*	—

Exam & portfolio

Format is set: oral exam with a portfolio — the details are ours to shape together

How it works

- Every lab leaves a working artifact; together they form your portfolio
- At the exam: present one artifact — a lab showpiece, or your mini-project (alone or in a group of 3–4)
- Then one question from a different part of the course
- Assessment is individual; the mini-project may be shared

Let's ~~decide~~* discuss together (today or by S2)

- Groups for mini-projects: 3-4, or same as SW7 project?
- Present a lab artifact vs. mini-project: your choice, or mini-project required?
- How much of the grade should the presentation carry?
- Written reflection per lab: half a page — useful, or too much?

***for all reasons such as what suits best to evaluate your understanding and skills, make it feasible, and compliant with study regulations, etc. I will choose the format before October, after consulting with study board the oral group exam format based on mini-projects.** The format will take into account all details such as group size, relation to semester project, assessment, deadlines, report, etc.

Before next Wednesday

S2's lab writes code in three languages — be ready

Install (30 minutes, do it early)

- OCaml 5 + dune via opam (opam init; opam install dune utop)
- VS Code + the 'OCaml Platform' extension
- Python 3.10+ (any recent version)
- Java 17+ (any JDK; javac must run)
- Smoke test: write the sum of integers in OCaml — recursion and a fold

If installation fights back

- Use the course devcontainer / Codespace (link on Moodle)
- Zero-install fallback for OCaml: try.ocaml.pro in the browser
- Bring the problem to the S2 lab — the first half hour is for exactly this
- Next week: persistent data structures, and a maze you will solve three times