Session 2: Persistent Data Structures

Léon Gondelman

September 9, 2026

SW7 Programming Paradigms Course
Aalborg University @ Copenhagen

Roadmap for today

Lecture.

- *Persistent data structures*: ephemeral versus persistent structures, the checkpoint trick and the case in which it fails, immutability $\Rightarrow$ safe sharing $\Rightarrow$ persistence, in Java and in Python.
- *A first look at OCaml*: a language in which persistence is the default, and in which the distinction is visible *in the types*.

Lab. We build a *persistent queue* from two lists and use it to explore a maze. It holds the *frontier* — the cells found but not yet visited — and does two jobs at once.

- **It chooses the search.** With the queue the maze is explored **breadth-first**; swap in today's persistent *stack* and the same code explores **depth-first**, unchanged.
- **It keeps the history.** No version is ever destroyed, so every earlier frontier still exists. We can step back through the search and ask what it held at earlier steps, having recorded nothing.

Part 1: Persistent Data Structures

Data structure = data + structural relations.

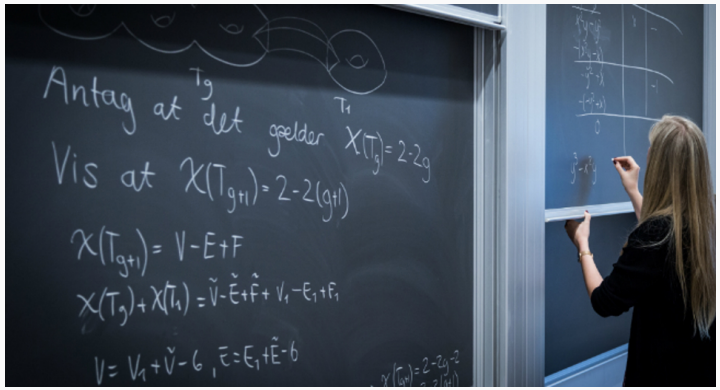
Computer programs usually operate on tables of information. In most cases these tables are not simply amorphous masses of numerical values; they involve important structural relationships between the data elements. In order to use a computer properly, we need to understand the structural relationships present within data, as well as the basic techniques for representing and manipulating such structure within a computer.

D. E. Knuth, The Art of Computer Programming, vol 1, chap 2, "Information structures", 1968.

An algorithm is like a cooking recipe!

- Input data and intermediate results are not preserved during computation.
- Fundamental formalism: the Turing machine.

An algorithm is a mathematical definition



- Computation seen as a sequence of steps constructing the final result, without ever erasing inputs or intermediate results.
- Fundamental formalisms: general recursive functions, λ-calculus, rewriting systems, formal semantics.
- "Declarative" programming (as opposed to "imperative"): functional, logic, constraint-based, etc.

What algorithms for declarative programming?

- Declarative programming is reputed inefficient because it cannot use *ephemeral* (in-place modified) data structures (e.g. arrays).
- **Persistent data structures** address this criticism:
  - Their interface only exposes operations that do not modify the structure in place, but instead return **new** modified structures.
  - Their implementation achieves, or comes close to, the complexity of the best known ephemeral structures.

The relevance of persistence for imperative programming

Even in imperative programming and classical algorithms, persistent data structures can be relevant:

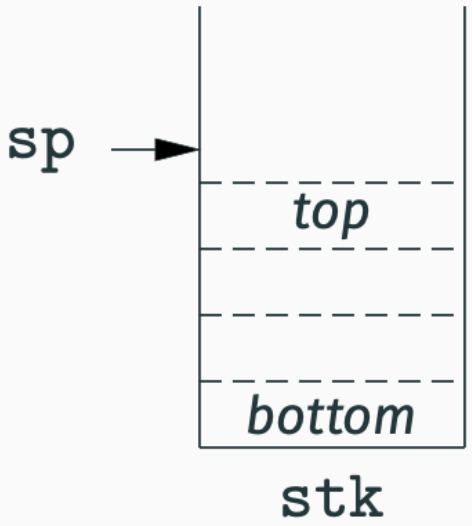
- **Backtracking** (paths exploration, checkpoints) are trivial.
- One can keep **the complete history of updates** of the structure.
- The absence of in-place modification allows **memory sharing** between successive versions, enabling a compact representation of history.
- **concurrency** (e.g. immutable versions can be shared across threads without locks)

Example: A stack

Operations:

- `init` empties the stack
- `push(v)` pushes `v` on the top
- `top` returns the value at the top
- `pop` removes the value at the top

An ephemeral stack implemented with an array

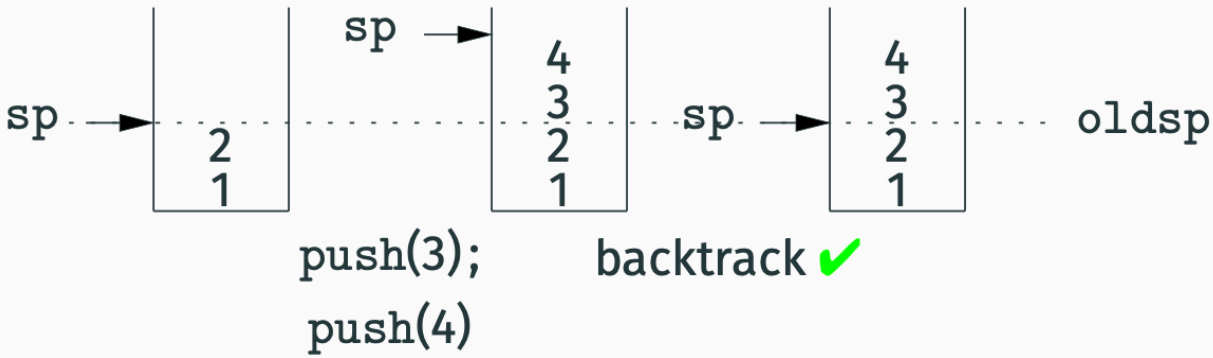



```
int stk[SIZE];
int sp;

void init(void) { sp = 0; }

void push(int v) {
  assert (sp < SIZE);
  stk[sp] = v; sp = sp + 1;
}

int top(void) {
  assert (sp > 0); return stk[sp - 1];
}

void pop(void) {
  assert (sp > 0);  sp = sp - 1;
}
```

Backtracking (checkpointing & backtracking)

Efficient but incomplete approach:

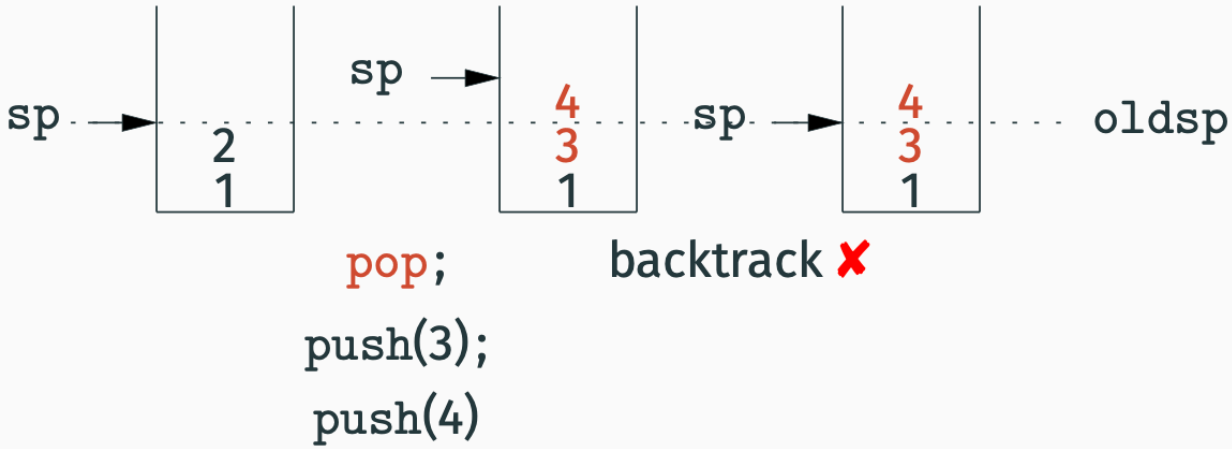
- Checkpoint: `oldsp = sp` (save the stack pointer)
- Backtrack: `sp = oldsp` (restore the stack pointer)



Backtracking (checkpointing & backtracking)

Efficient but incomplete approach:

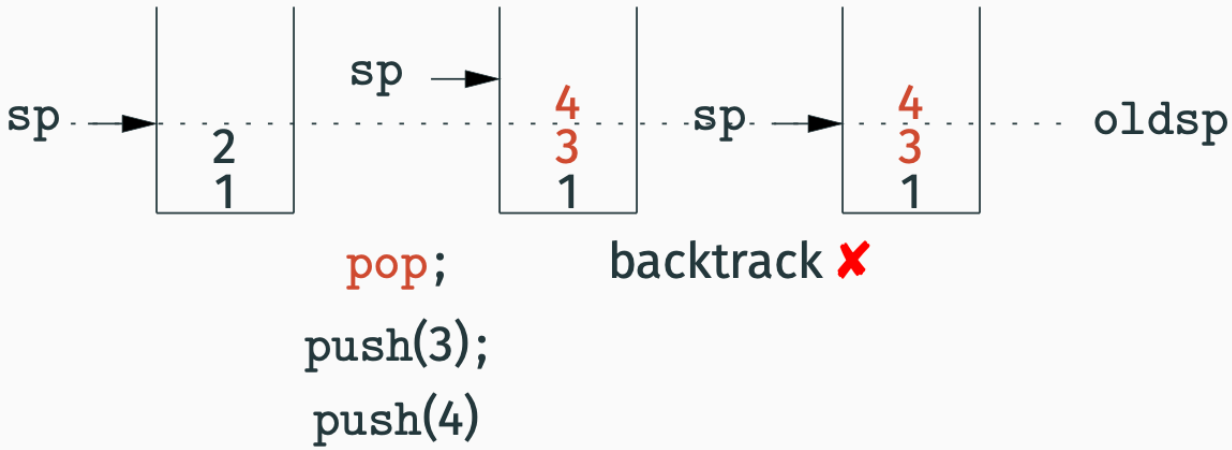
- Checkpoint: `oldsp = sp` (save the stack pointer)
- Backtrack: `sp = oldsp` (restore the stack pointer)



Backtracking (checkpointing & backtracking)

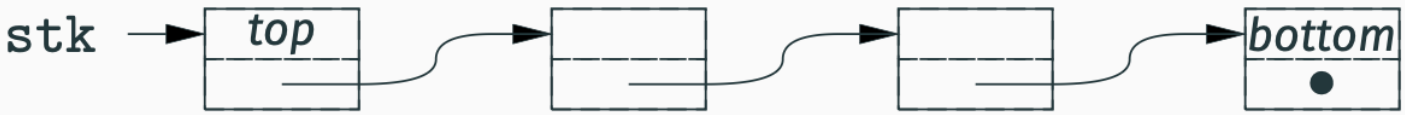
Efficient but incomplete approach:

- Checkpoint: `oldsp = sp` (save the stack pointer)
- Backtrack: `sp = oldsp` (restore the stack pointer)



Always correct but inefficient alternative:
copy `stk[0..sp[` to another array.

A persistent stack implemented as a linked list



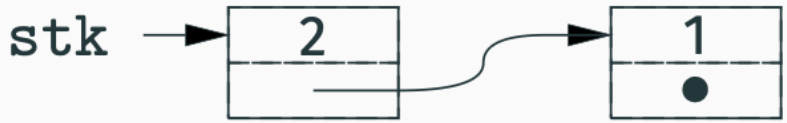

```
final class Stack {
  private final int hd;
  private final Stack tl;
  private Stack(int v, Stack s) { hd = v; tl = s; }

  static final Stack empty = null;
  static Stack push(int v, Stack s) { return new Stack(v, s); }
  static int   top (Stack s) { return s.hd; }
  static Stack pop (Stack s) { return s.tl; }
}
```

The interface of the data structure changes: `push` and `pop` *return* the new stack value (no in-place mutation).

The implementation relies on the language memory management (GC) to reclaim list cells after `pop`.

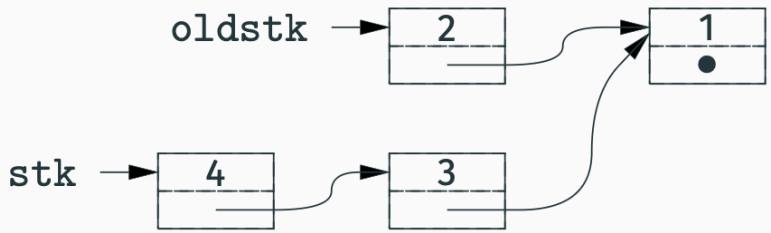
Note every `private` and every `final`. We come back to why.

Persistent stacks and backtracking



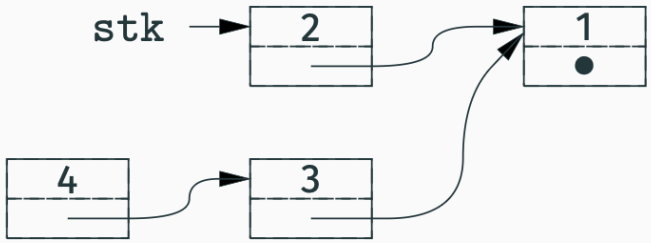

```
stk = Stack.push(2, Stack.push(1, Stack.empty));
```

Persistent stacks and backtracking




```
stk = Stack.push(2, Stack.push(1, Stack.empty));

// Checkpoint
oldstk = stk;
// Work
stk = Stack.push(4, Stack.push(3, Stack.pop(stk)));
```

Persistent stacks and backtracking



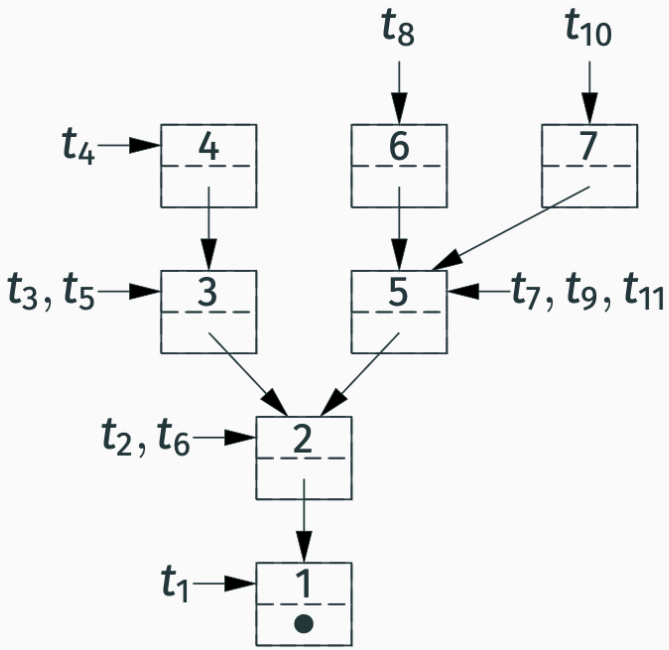

```
stk = Stack.push(2, Stack.push(1, Stack.empty));

// Checkpoint
oldstk = stk;
// Work
stk = Stack.push(4, Stack.push(3, Stack.pop(stk)));

// Backtrack
stk = oldstk;
```

Persistent stacks and memory sharing

Key observation: a stack produced by `push` or `pop` **shares** all but one node with the previous stack.

This enables keeping *all* successive states economically, so that total memory for a sequence of N pushes and M pops is about N nodes.

$$
\begin{aligned}
t_1 &= \mathtt{push}(1, \mathtt{empty}) \\
t_2 &= \mathtt{push}(2, t_1) \\
t_3 &= \mathtt{push}(3, t_2) \\
t_4 &= \mathtt{push}(4, t_3) \\
t_5 &= \mathtt{pop}(t_4) \\
t_6 &= \mathtt{pop}(t_5) \\
t_7 &= \mathtt{push}(5, t_6) \\
t_8 &= \mathtt{push}(6, t_7) \\
t_9 &= \mathtt{pop}(t_8) \\
t_{10} &= \mathtt{push}(7, t_9) \\
t_{11} &= \mathtt{pop}(t_{10})
\end{aligned}
$$



Why `private`? Why `final`?

Two keywords we put on the Java linked-list stack, and never justified.

```
Stack v1 = push(3, push(2, push(1, empty)));   // [3, 2, 1]
Stack v2 = pop(v1);                            // [2, 1]
     // v2 IS the second node of v1 -- the two versions SHARE it

v2.tl = null;   // if tl were public and non-final
```

What is `v1` now?

Immutability, Sharing, Persistence

- **Immutability**: operations never overwrite state; they *return a new value*.
- **Structural sharing**: a new version *reuses the unchanged parts* of the old one — a new node whose tail *is* the old stack.
- **Safe** sharing: no version can be changed *through another version's pointer*. Sharing on its own is merely aliasing — the previous slide. *Immutability buys this outright*, by making the change impossible.
- **Persistence**: every old version stays valid and reachable.

$$\text{immutability} \Rightarrow \text{safe sharing} \Rightarrow \text{persistence}$$

An implication, not an equivalence — next slide.

Immutability is one route, not the only one

Safe sharing asks for one thing: *no version may be changed through another version's pointer*. There is more than one way to arrange that.

Route	Safe because	Checked by
Immutability	nothing ever changes	compiler: `final`, `frozen`
Encapsulation	only the owner may change it	compiler, for clients
Ownership	aliasing and mutation never meet	compiler (Rust)
Benign effects	it changes, nobody can tell	us, by argument
Copy-on-write	a writer gets its own copy	the runtime

Today we take immutability: the compiler checks it, and it preserves *structural* sharing. Copy-on-write is just as safe, but duplicates the whole structure at the first write — the very thing we are avoiding.

Later we will need benign effects. Session 5 repairs today's queue with a thunk that overwrites itself once forced; session 7 compresses paths in a persistent Union-Find. Both mutate, and no client can tell — which is why that row is checked by argument rather than by a compiler.

If immutability or safe sharing is missing

Sharing without immutability $\Rightarrow$ *corruption*.

```
v2.tl = null;   // v1 was [3,2,1] -- now [3,2]
```

An old version changed although nobody touched it. Restore that checkpoint while backtracking, and we resume from a state that never existed.

Immutability without sharing $\Rightarrow$ *copying*. To keep a checkpoint we must copy the whole structure: a stack of 10^6 nodes costs 10^6 nodes per checkpoint instead of one pointer. Backtracking degrades from $O(1)$ to $O(n)$ per step.

We need both: drop either one and persistence becomes either *wrong* or *unaffordable*.

Java: why `private` and `final`?

Back to the open question: `v2.tl = null` does not compile. That is the whole point.

- `private fields`: encapsulates representation — clients cannot mutate internals.
- `final` **fields**: enforce immutability after construction ⇒ not even `Stack` itself can rewrite a node once it exists, and the compiler checks it.
- `final` **class** (optional but recommended): prevents subclassing that could reintroduce mutability or break invariants.

Bonus, and a session 7 teaser: `final` also makes these fields safely visible to other threads, with no locking.

Python: tuples-as-cons — immutability & sharing

In Python once tuples are created, their items can't be updated.
Stack as nested tuples: (hd, tl), where tl is another pair or None.

```
EMPTY = None

def push(v, s):     # O(1)
    return (v, s)  # NEW tuple; 's' is SHARED (structural sharing)

def top(s):
    if s is None: raise IndexError("empty")
    return s[0]

def pop(s):                          # O(1)
    if s is None: raise IndexError("empty")
    return s[1]

# Example: versions share structure
s1 = push(1, EMPTY)                  # [1]
s2 = push(2, s1)                     # [2,1]  (shares node '1')
s3 = push(3, s2)                     # [3,2,1] (shares tail [2,1])

# s2[1] = None        # -> TypeError (tuples are immutable)
```

Sharing: push creates 1 new pair & reuses the whole tail $\Rightarrow$ $O(1)$ upd.

Python: immutable cons-list (structural sharing)

A persistent stack needs nodes that can never change once built. Python says so in one word: `frozen`.

```
from dataclasses import dataclass
from typing import Optional

@dataclass(frozen=True)
class Stack:
    hd: int
    tl: "Optional[Stack]" = None

def push(v, s): return Stack(v, s)   # one new node; tail s is SHARED
def pop(s):
    if s is None: raise IndexError("empty")
    return s.tl

s1 = push(1, None); s2 = push(2, s1); s3 = push(3, s2)
s2.tl = None        # FrozenInstanceError -- s3 is protected
```

Same story as in Java, one line shorter: `frozen=True` generates the `__setattr__` that refuses the write. Without it, `s2.tl = None` would retroactively change s3, which shares s2 as its tail.

Some languages make persistence the default

Everything so far was persistence obtained *against the language's defaults*: Java needed `private` and `final`, Python needed `frozen`. Some languages default the other way.

Lisp / Scheme (1958 onwards) — the cons cell is immutable by convention, and the whole language is built on it:

```
(define s1 (cons 1 '()))      ; [1]
(define s2 (cons 2 s1))       ; [2,1]  -- s1 is SHARED, not copied
```

Haskell (1990) — a *pure* language: ordinary code has no assignment at all, so every structure is persistent whether we intended it or not. Mutation exists, but only where the type admits it:

```
s1 = [1]
s2 = 2 : s1     -- the same cons, spelled with a colon
```

The same data structure, and no keyword required: the default has done the work for us.

OCaml: the stack, one more time

Our language for the remainder of the course. Immutable by default, yet mutation costs nothing to reach for — where Haskell makes us say so in the type. Both languages let us *choose*; they differ in what the choice costs, and that is one of the subjects of this course.

The persistent stack, in five lines:

```
type 'a t = 'a list

let empty = []
let push v s = v :: s
let pop s  = match s with [] -> None | v :: rest -> Some (v, rest)
let peek s = match s with [] -> None | v :: _    -> Some v
```

Reading it: `let` binds a name; `'a` is Java's <T>; `[]` is the empty list and `v :: s` puts `v` in front of `s`; `match` performs case analysis on the shape of a value — the switch statement we always wanted.

`v :: s` *is* the sharing diagram we drew earlier — one new cell, and the old stack reused entire.

Part 2: Introduction to OCaml

OCaml

A general-purpose language, *statically and strongly typed*, from the ML family (with Standard ML and F#).

- Designed and implemented at Inria, in France: Caml from 1985, and OCaml since 1996, by **Xavier Leroy**, Damien Doligez, Jérôme Vouillon, Didier Rémy and others — Leroy's is the course we followed for the first half of today.
- Types are *checked* everywhere but *written* almost nowhere: the compiler infers them, and gives strong guarantees in return.
- Compiled to native code, and garbage collected.
- Immutable by default — yet mutation is available when we ask for it. That last point is why it belongs in this course.

A language that has an opinion, but still lets us choose.

Who writes OCaml

- **Jane Street** — a trading firm whose entire system is OCaml; one of the largest single codebases in any typed functional language.
- **Proof assistants and verification tools**: Rocq (until recently Coq), Why3, Frama-C, Alt-Ergo — the tools used to prove that programs do what they claim.
- **Static analysis**: Astrée, used on Airbus flight-control software; Infer and Flow at Meta; Semgrep.
- **Systems**: the Tezos blockchain, the Unison file synchroniser, MirageOS unikernels.

Not an accident: OCaml turns up where being wrong is expensive.

`let`: definitions, and the types we never write

```
let x = 40 + 2                  (* a value *)

let double n = 2 * n            (* a function: no braces, no return *)

let area r =
  let pi = 3.14159 in           (* a LOCAL binding, in force below *)
  pi *. r *. r                  (* *. is multiplication on floats *)
```

There is no `return`: the value of the body *is* the result.

In the interactive toplevel the compiler tells us what it worked out:

```
# let double n = 2 * n ;;
val double : int -> int = <fun>
```

We wrote no type at all, and yet `double` is fully typed.

OCaml is not “the functional language”

Mutable state is available — we need only ask for it by name.

```
let counter = ref 0              (* a mutable cell *)

let tick () =
  counter := !counter + 1;       (* := writes, ! reads *)
  !counter
```

`tick` takes `()`, the single value of type `unit` — OCaml's way of saying *no interesting argument*, and the return type of anything whose purpose is an effect.

The standard library ships a mutable `Stack`, `Queue` and `Hashtbl` beside the immutable `List` and `Map`. *The paradigm is a choice made inside the language, not by it.*

Lists: a linked list with the links hidden

```
let nothing = []                (* the empty list *)
let l   = [1; 2; 3]             (* a literal -- note the SEMICOLONS *)
let l' = 1 :: 2 :: 3 :: []      (* exactly the same list *)
```

A list is a chain of cells, each holding a value and a pointer to the rest — the linked list from the first half of today. The pointer is hidden, as in Java and unlike in C; and unlike Java it cannot be assigned either. There is no `.next` to write to, and no assignment for list cells anywhere in the language.

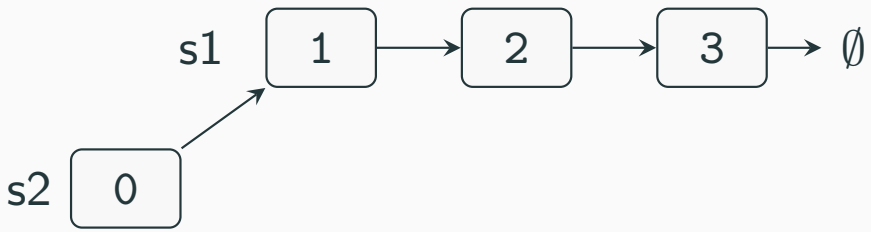
So a list behaves as a Python *tuple* behaves — we may read it and build new ones from it, but we cannot change one. The nested pairs we built by hand in Python are what OCaml calls, without ceremony, a list.

Careful: `[1, 2, 3]` is not an error. It is a list of *one* element, the triple `(1, 2, 3)` — and the complaint arrives later, somewhere else.

Lists: immutable, and shared

```
let s1 = [1; 2; 3]          (* separator is ; not , *)
let s2 = 0 :: s1            (* [0; 1; 2; 3] *)
```

s2 is one new cell whose tail *is* s1. Nothing was copied, and nothing can be modified: there is no assignment to a list cell in the language at all.



This is the sharing diagram from the first half, now built into the language.

`match`: asking what shape a value has

```
let rec sum l =
  match l with
  | []        -> 0
  | x :: rest -> x + sum rest
```

A list is either empty or a head followed by a tail; `match` names those two cases and binds `x` and `rest` in the second.

Two things a `switch` never did for us:

- it *destructures* — `x` and `rest` come out of the value, no accessors needed;
- it is *checked* — omit a case and the compiler says which one. Session 3 makes that guarantee the whole point.

`let rec`: recursion

A function that calls itself must say so: `let rec`.

```
let rec fact n =
  if n <= 1 then 1 else n * fact (n - 1)

let rec length l =
  match l with
  | []      -> 0
  | _ :: tl -> 1 + length tl
```

On numbers we recurse towards zero; on lists we recurse towards the empty list. The shape of the data decides the shape of the function — an idea we take much further in session 3. *Loops are not forbidden; they are not the first thing we reach for.*

Records: the type our queue will have

Given to us in the lab. A record groups several values under names:

```
type 'a t = { front : 'a list; back : 'a list }

let empty = { front = []; back = [] }      (* construction *)
let is_empty q = q.front = []              (* field access *)
```

And a shorthand worth recognising before we meet it:

```
{ front; back }          (* means  { front = front; back = back } *)
```

'a is the type variable — Java's <T>. A record is immutable unless a field is declared `mutable`, so every operation on this queue necessarily *returns a new one*.

`option`: OCaml has no `null`

What should `dequeue` return on an empty queue? OCaml refuses to let us answer "nothing, silently".

```
val dequeue : 'a t -> ('a * 'a t) option
```

A value of type `'a option` is either `None` or `Some v`. The empty case is *in the type*, so a caller cannot overlook it: the value is not reachable until the `None` case has been handled.

`option` is a type defined by *listing its cases* — our first *algebraic data type*, and the only one we borrow today. Declaring our own is session 3.

Python	`raise IndexError`
Java	`throw new NoSuchElementException`
OCaml	`None` — checked at compile time

The same decision, made three times; only one of them is enforced.

A record may be mutable — and the paradigm changes

A field marked `mutable` can be assigned with `<-`. Here is the same stack, written imperatively:

```
type 'a stack = { mutable elems : 'a list }

let create () = { elems = [] }
let push v s  = s.elems <- v :: s.elems
let pop s =
  match s.elems with
  | []        -> None
  | v :: rest -> s.elems <- rest; Some v
```

Same language, same data, same operations. The whole difference is in one type:

imperative	`val push : 'a -> 'a stack -> unit`
persistent	`val push : 'a -> 'a t -> 'a t`

`unit` says: I changed something we cannot see.

Running OCaml from a terminal

Interactively — `utop` is friendlier, if installed:

```
$ ocaml
# 1 + 2 ;;
- : int = 3
```

As a script, for a single file:

```
$ ocaml tour.ml
```

Compiled, which is what we normally want:

```
$ ocamlfind ocamlopt -package str hello.ml -o hello
$ ./hello
```

`ocamlopt` produces native code; `ocamlc` produces bytecode. For more than two files this becomes tedious quickly — which is the point of the next slide.

`tour.ml` in the lab repository is this part of the lecture, as a program we can run.

dune: the build system we will actually use

A Makefile would work. dune is what the OCaml world settled on, and it needs almost no configuration — two small files describe the whole lab:

```
(* dune-project *)      (lang dune 3.0)

(* lib/dune *)          (library (name pqueue_lib) (modules pqueue))
```

Three commands are all we need today — compile, compile and test, and run:

```
$ dune build
$ dune runtest
$ dune exec bin/solver.exe -- ../mazes/medium.txt --bfs
```

dune works out the dependencies between files itself; there is no list of source files to maintain.

`.ml` and `.mli`: the interface is the contract

Each module may carry an interface. The `.mli` says what exists; the `.ml` says how.

```
(* pqueue.mli -- the contract *)
type 'a t              (* abstract -- no definition here *)

val empty   : 'a t
val enqueue : 'a -> 'a t -> 'a t
val dequeue : 'a t -> ('a * 'a t) option
```

Because `'a t` is given *without* a definition, no client can see that a queue is two lists, and therefore no client can build one by hand or depend on the representation. We are free to change it tomorrow.

The compiler checks the `.ml` against the `.mli`. This is the whole of today's first half, said in one file — and session 11 returns to it properly.

What we have not said yet

Today was the reading knowledge needed for the lab. The rest arrives when there is a reason for it.

Idea	Today	Properly in
Sum types, exhaustiveness	`option` only	session 3
Functions as values, `fun`	mentioned	session 4
Recursion and tail calls	two examples	session 4
`map`, `filter`, `fold`, closures	used once, unexplained	session 5
Lazy evaluation, `Seq`	the cliffhanger	session 5
Mutation, and when it is safe	`ref`	session 7
Polymorphism, `'a`, inference	`'a` as `<T>`	session 10
Modules, signatures, functors	one `.mli`	session 11

Enough OCaml to build a queue is a surprisingly small language.

Acknowledgments

The persistence material of this lecture follows

Xavier Leroy's course at the Collège de France.

The amortization and queue material follows

Chris Okasaki, *Purely Functional Data Structures*,
Cambridge University Press, 1998.

Lab Session and Homework

Lab session, and what to do afterwards

In the lab (2 hours).

- *On paper:* the queue's invariant, and why the $O(n)$ reversal is affordable.
- *In Python:* we implement it, then drive the maze solver — breadth-first, the one-line exchange to depth-first, the history view.
- *In OCaml:* the same queue again, against a given `.mli`.
- *And then we break it:* one version dequeued repeatedly redoes the same $O(n)$ reversal every time. Session 5 makes that result shared.

Homework.

- Finish the lab, and write the queue a third time in **Java**.
- Small **OCaml exercises** — lists and pattern matching only.
- The **proofs** on the paper sheet: the invariant, the FIFO argument, and the assumption persistence breaks.