Compiler

Introduction

Jyun-Ao Lin

iFIRST & CSIE, NTUT

Table of contents

Introduction

Credits

A large part of this course is based on the Compilation Course of J.-C. Filliâtre at ENS Ulm.

Course organization

- Course Tuesday 13:10 – 15:00, Friday 15:10 – 16:00
- Location 第三教學大樓 506(e)
- Office hour Tuesday 15:10 – 17:10, Friday 13:00 – 15:00
- Course website `https://carquois42.github.io/compiler.html`
- Course Teams `https://is.gd/gSPaT4`
- Contact jalin@ntut.edu.tw, 先鋒大樓 1310

Evaluation

- no exam!
- several handwritten assignments and programming assignments: 50%
  (we will program in OCaml)
- a final project = a mini compiler: 50%
  - some parts will be done during the courses, some at home
  - along or in pair
  - however, if you wish, you can use whatever programming language you like
- no textbooks but some useful references
  - [ALSU06] Alfred V. Aho, Monica S. Lam, Ravi Sethi et Jeffrey D. Ullman, *Modern Compilers: Principles, Techniques, and Tools*, the Dragon Book.
  - [BO16] Randal E. Bryant et David R. O'Hallaron, Computer Systems: A Programmer's Perspective.
  - [Pie02] Benjamin C. Pierce, Types and Programming Languages.

Course objectives

Understand the mechanisms behind compilation,
that is, the translation from one language to another.

Understand the various aspects of programming languages,
via their compilation.

The need of new compilers

- new kinds of optimizations for new kinds of hardware and new kinds of domain-specific semantics (e.g. R, Tensorflow, ..etc)
  - $O(m \times n)$:
  - $m = \#$ of programming languages
  - $n = \#$ of target hardwares
- no one knows how to write compilers to unlock the full potentials of those domain-specific hardwares[1]
  i.e. the science of compilers is not advanced yet and has not caught up with the massive explosion of domain-specific languages and novel hardware accelerators

[1] c.f. [MDHS09] for some interesting facts

Compilation

A compiler translate a "program" from a source language to a target language, possibly signaling errors.

$$
\begin{tikzcd}
\text{source language} \arrow[r] & \text{compiler} \arrow[r] \arrow[d] & \text{target language} \\
 & \text{errors} &
\end{tikzcd}
$$

Detecting (static) errors like

- malformed identifiers, unclosed comments ...
- incorrect syntactic constructions
- undeclared identifiers
- mistyped expressions, e.g. `if 3 then "toto" else 4.5`
- uninstantiated references
- etc

Compilation to machine language

Compilation typically evokes translating a high-level language (C, Java, OCaml, etc.) to some machine language

```
% gcc -no-pie -o sum sum.c
```

source `sum.c` ⟶ C compiler (`gcc`) ⟶ executable `sum`

```c
int main(int argc, char **argv) {
    int i, s = 0;
    for (i = 0; i <= 100; i++) s += i*i;
    printf("0*0+...+100*100 = %d\n", s);
}
```

⟶

```
00100111101111011111111111100000
10101111101111110000000000010100
10101111101001000000000000100000
10101111101001010000000000100100
10101111101000000000000000011000
10101111101000000000000000011100
10001111101011100000000000011100
...
```

Target language

In this lecture, we are going to consider compiling to assembly, indeed, but this is just only one aspect of compilation

Many techniques used in compilers are not related to the production of assembly code.

Moreover, some languages are instead

- interpreted (Basic, COBOL, Ruby, etc.)
- compiled into some intermediate language, which is then interpreted (Java, Python, OCaml, Scala, etc.)
- just-in-time (on-the-fly) compiled (Julia, etc.)
- compiled into another high-level language

Differences between a compiler and an interpreter

- A compiler translates a program P into a program Q such that for any input x, the output of $Q(x)$ is identical to that of $P(x)$

$$\forall P \; \exists Q \; \forall x \ldots$$

- An interpreter is a program that, given a program P and some input x, computes the output s of $P(x)$

$$\forall P \; \forall x \; \exists s \ldots$$

Differences between a compiler and an interpreter

In other words

- the compiler performs a more complex task only once, to produce a code that accepts any input
- the interpreter performs a simpler task, but repeats it for every input

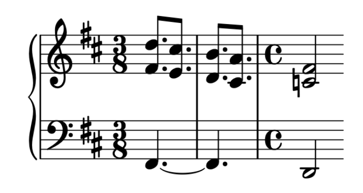
Another difference: in general compiled code is typically more efficient than interpreted code

- best interpreters are at least 10x slower than compiled code (e.g. Python ~30–50x)
  $\Rightarrow$ use up $>$ 10x more energy
  $\Rightarrow$ Facebook compiles PHP to C++

Example of compilation and interpretation[2]

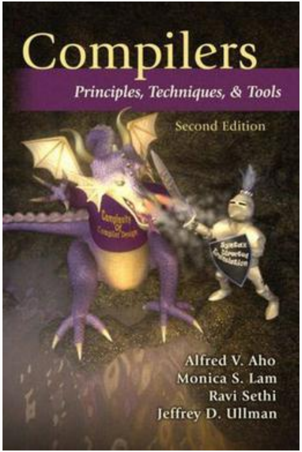
source → lilypond → PDF file → evince → image

```
\new PianoStaff <<
  \new Staff { \clef "treble" \key d \major \time 3/8
               <<d8. fis,8.>> <<cis'8. e,8.>> | ... }
  \new Staff { \clef "bass" \key d \major
               fis,,4. ~ | fis4. | \time 4/4 d2 }
>>
```



[2] Adopted from J.-C. Filliâtre

Quality of a compiler



How can we evaluate the quality of a compiler[a]?

- its soundness (i.e. correctness)
- the performance of the compiled code
- the performance of the compiler itself

"Optimizing compilers are so difficult to get right that we dare say that no optimizing compiler is completely error-free! Thus, the most important objective in writing a compiler is that it is correct."[b]

(Dragon Book [ALSU06]).

[a] c.f. SIGPLAN Empirical Evaluation Guidelines

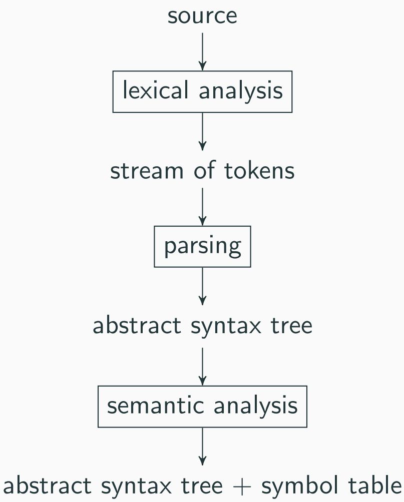
[b] Now we have at least a verified compiler `CompCert C` for C language by Xavier Leroy [Ler06]: the test generation tool CSmith [YCER11] found 79 bugs in GCC and 202 bugs in LLVM but was unable to find any bugs in the verified parts of CompCert.

Compiler phases

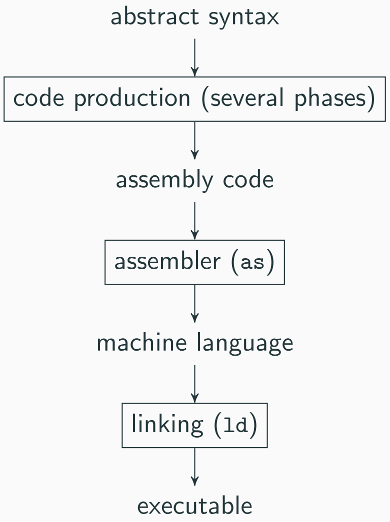
Typically, the compiler decomposes into

- a frontend/analysis phase
    - recognizes the program and its meaning
    - signals errors and thus can fail
      (syntax errors, scoping errors, typing errors, etc.)
- and a backend/synthesis phase
    - produces the target code
    - uses many intermediate languages
    - sometimes optimizes
    - must not fail

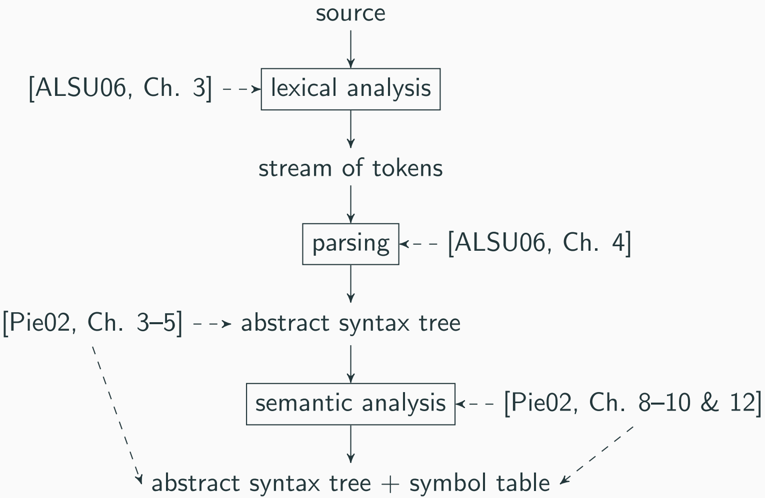
Standard Frontend



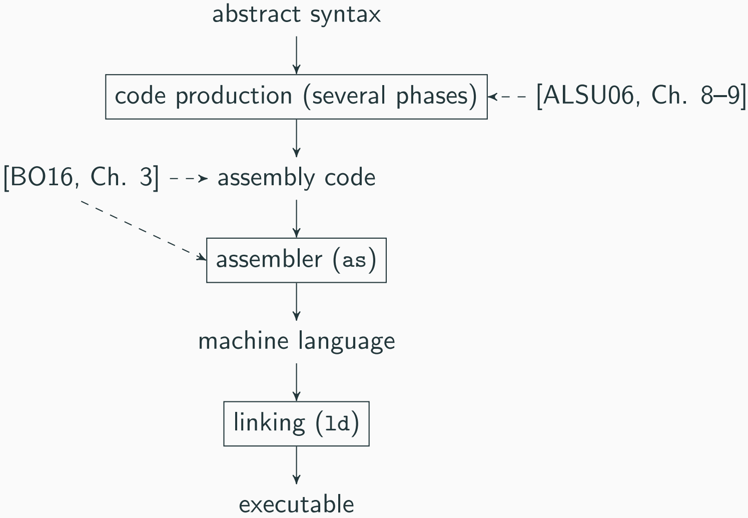
Standard Backend



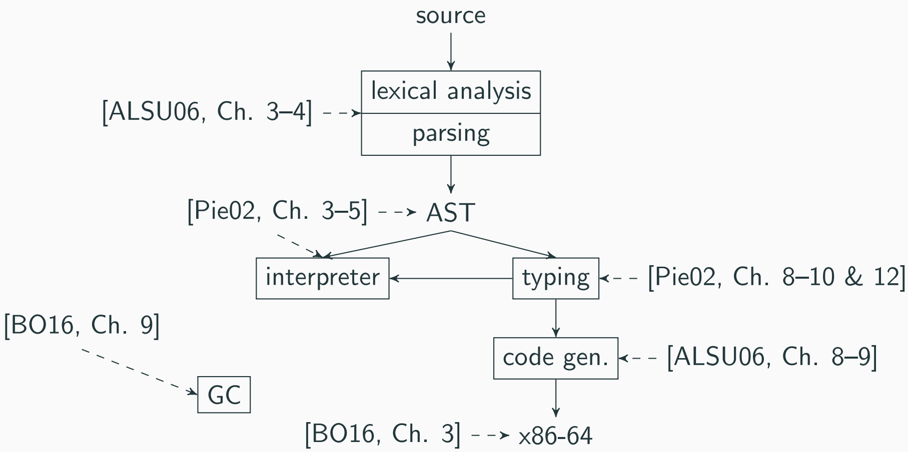
Standard Frontend



Standard Backend



Overview of the course



References i

Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman.
Compilers: Principles, Techniques, and Tools (2nd Edition).
Addison-Wesley Longman Publishing Co., Inc., USA, 2006.

R.E. Bryant and D.R. O'Hallaron.
Computer Systems: A Programmer's Perspective.
Always Learning. Pearson, 2016.

Xavier Leroy.
Formal certification of a compiler back-end, or: programming a compiler with a proof assistant.
In *33rd ACM symposium on Principles of Programming Languages*, pages 42–54. ACM Press, 2006.

References ii

Todd Mytkowicz, Amer Diwan, Matthias Hauswirth, and Peter F. Sweeney.
Producing wrong data without doing anything obviously wrong!
In *Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XIV, page 265–276, New York, NY, USA, 2009. Association for Computing Machinery.

Benjamin C. Pierce.
Types and Programming Languages.
The MIT Press, 1st edition, 2002.

Xuejun Yang, Yang Chen, Eric Eide, and John Regehr.
Finding and understanding bugs in c compilers.
SIGPLAN Not., 46(6):283–294, jun 2011.

Questions?

First step in OCaml

OCaml

OCaml is a general-purpose, strongly typed programming language

Successor of Caml Light (itself successor of Caml),
part of the ML family (SML, F#, etc.)

Designed and implemented at Inria Rocquencourt by Xavier Leroy and others

Some applications: symbolic computation and languages (IBM, Intel, Dassault Systèmes), static analysis (Microsoft, ENS), file synchronization (Unison), peer-to-peer (MLDonkey), finance (LexiFi, Jane Street Capital), teaching

there is no good programming language, there are only good programmers

The first program

hello.ml

```
print_string "hello world!\n"
```

compiling

```
% ocamlopt -o hello hello.ml
```

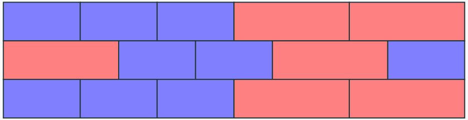
executing

```
% ./hello
hello world!
```

A little

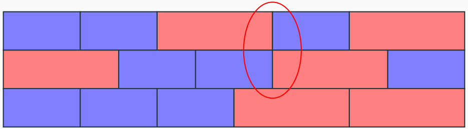
We want to build a wall with bricks of length 2 () and of length 3 (), which we have respectively infinite quantities.

For example, here we have a wall of length 12 and of height 3:



a little

to be solid, the wall must not overlap two joints excepts for the boundaries



Question

How many ways to construct a wall of length 32 and height 10?

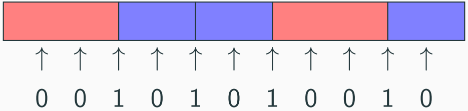
First Idea

We are going to calculate recursively the number of ways $C(r, h)$ to construct a wall of height h, whose lowest row of bricks r is given:

- Base case:
$$C(r, 1) = 1$$
- Inductive cases:
$$C(r, h) = \sum_{r' \text{ compatible with } r} C(r', h-1)$$

Second Idea

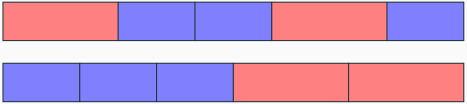
We are going to represent the rows of bricks by integers in base 2 where the digit 1's correspond to the presences of the joins.



e.g. this row is represented by the integer 338 ($= 00101010010_2$)

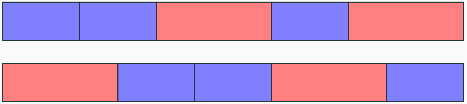
Why?

It is then easy to check that two rows are compatible by a simple logical "and" operation (land in OCaml), namely,



$$
\begin{array}{ll}
 & 00101010010_2 \\
\texttt{land} & 01010100100_2 \\
= & 00000000000_2 = 0
\end{array}
$$

but



$$
\begin{array}{ll}
 & 01010010100_2 \\
\texttt{land} & 00101010010_2 \\
= & 00000010000_2 \neq 0
\end{array}
$$

Arrange the bricks

Write a function `add2` which adds a brick of length 2 to the right of a row of bricks r

It shift the bits twice to the left and add 10_2.

Similarly we define a function `add3` which adds a brick of length 3 to the right of a row.

List all the rows of bricks

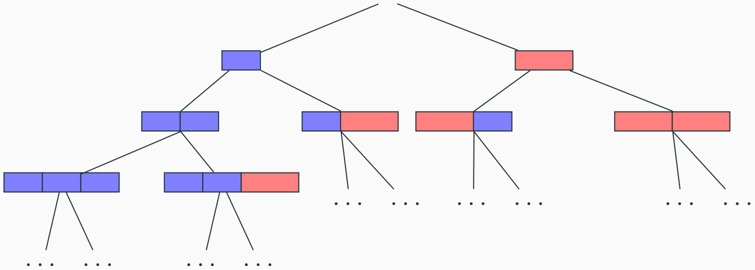
We will construct a list of all possible rows of bricks of length 32

In OCaml, lists are constructed from

- the empty list is denoted by `[]`
- adding an element x to the start of a list l is denoted by `x :: l`

List all the rows of bricks

We are going to write a recursive function `fill` which goes through this tree (conceptually)



until we find the rows of the right length.

Summation over a list

To write the recursive function C, we start by writing a function sum which calculates

$$\texttt{sum}\ f\ l = \sum_{x \in l} f(x)$$

that is

```
sum: (int -> int) -> int list -> int
```

Recursively countdown

Finally, let us write a recursive function `count` corresponding to function C:

and to obtain the solution of the problem, it suffices to consider all the possible basic rows.

Deception

Unfortunately, it takes much much much much too much time.....

Third Idea

The problem is that we have reached the same pair (r, h) as the arguments of the `count` function too often, therefore we calculate the same things several times...

Hence, a third idea: storing $C(r, h)$'s that have been calculated in a table $\longrightarrow$ this is what we call memorization.

What kind of a table?

Therefore we need a table which associates with certain keys (r, h) the value $C(r, h)$.

We will use a hash table.

Hash Table

The idea is very simple: we are given an arbitrary function

$$hash : keys \rightarrow \texttt{int}$$

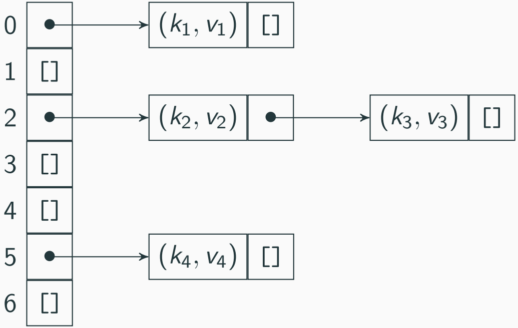
of values in $0, 1, \ldots, n-1$ and a table of size n.

For each key k associated with a value v, we place the pair (k, v) in the box of the table $hash(k)$ mod n.

Note: several keys can be found in the same box $\Rightarrow$ each box is a list.

Hash Table

Hence, if $n = 7$, $hash(k_1) = 0 \mod 7$, $hash(k_2) = hash(k_3) = 2 \mod 7$ and $hash(k_4) = 5 \mod 7$, then



Memorization

We can now use the hash table in the function C.

We are going to write two mutually recursive function `count` and `memo_count`

- `count` performs the calculation, by calling `memo` recursively.
- `memo_count` consults the table, and if necessary, calls `count` to fill it.

We win!

We finally obtain the result

```
% ocamlopt wall.ml -o wall
% time ./wall
806844323190414

real 0m1.072s
```

Summary

Declaration

Program = sequence of declarations and expressions to evaluate

Example

```
let x = 1 + 2;;
print_int x;;
let y = x * x;;
print_int y;;
```

- Interpretation, possibly interactive
- two compilers: `ocamlc` (bytecode) and `ocamlopt` native

Variables

`let x = e` introduces a global variable

differences wrt usual notion of variable:

1. necessarily initialized
2. types not declared but inferred
3. cannot be assigned

Java	OCaml
`final int x = 42;`	`let x = 42`

References

A variable to be assigned is called a reference

It is introduced with `ref`

```
let x = ref 1;;
print_int !x;;
x := !x + 1;;
print_int !x;;
```

Expression and Instructions

There is no distinction between expression and instructions in the syntax: only expressions

All the expressions are typed

Some usual constructs:

- conditional

```
if i = 1 then 2 else 3
```

- for loop

```
for i = 1 to 10 do x := !x + i done
```

- sequence

```
x := 1; 2 * !x
```

`unit` type

Expressions with no meaningful value (assignment, loop, ...) have type `unit`
This type has a single value, written `()`

It is the type given to the `else` branch when it is omitted

correct:

```
if !x > 0 then x := 0
```

incorrect:

```
2 + (if !x > 0 then 1)
```

Local variables

in C or Java, the scope of a local variable extends to the bloc:

```
{
    int x = 1; ...
}
```

in OCaml, a local variable is introduced with `let in`:

```
let x = 10 in x * x
```

As for a global variable:

- necessarily initialized
- type inferred
- immutable
- but scope limited to the expression following in

`let in =` expression

`let x = e1 in e2` is an expression

Its type and value are those of `e2`,

in an environment where `x` has the type and value of `e1`

Example

```
let x = 1 in (let y = 2 in x + y) * (let z = 3 in x * z)
```

Parallel

Java	OCaml
`{ int x = 1;`	`let x = ref 1 in`
`x = x + 1;`	`x := !x + 1;`
`int y = x * x;`	`let y = !x * !x in`
`System.out.print(y); }`	`print int y`

Recap

- program = sequence of expressions and declarations
- variables introduced with let and immutable
- no distinction expression / statement

Interactive loop

Interactive version of the compiler

```
% ocaml
        OCaml version 4.14.1
        ...
# let x = 1 in x + 2;;
```

```
- : int = 3
```

```
# let y = 1 + 2;;
```

```
val y : int = 3
```

```
# y * y;;
```

```
- : int = 9
```

Functions

Functions

- functions = values like the others: local, anonymous, arguments of other functions, etc
- partially applied
- the function call is not expensive
- polymorphic

Syntax

```
# let f x = x * x;;
```

```
val f : int -> int = <fun>
```

- body = expression (no `return`)
- type is inferred (types of argument `x` and result)

```
# f 4;;
```

```
- : int = 16
```

Parallel

Java	OCaml

`static int f(int x) {`
`    return x * x;`
`}`
	`let f x =`
	`    x * x`

Procedure

a procedure = a function whose result type is unit

Example

```
# let x = ref 1;;
# let set v = x := v;;
```

```
val set : int -> unit = <fun>
```

```
# set 3;;
```

```
- : unit = ()
```

```
# !x;;
```

```
- : int = 3
```

Function without arguments

takes an argument of type `unit`

Example

```
# let reset () = x := 0;;
```

```
val reset : unit -> unit = <fun>
```

```
# reset ();;
```

Function with several arguments

```
# let f x y z = if x > 0 then y + x else z - x;;
```

```
val f : int -> int -> int -> int = <fun>
```

```
# f 1 2 3;;
```

```
- : int = 3
```

Local function

function local to an expression

```
# let sqr x = x * x in sqr 3 + sqr 4 = sqr 5;;
```

```
- : bool = true
```

function local to another function

```
# let pythagorean x y z =
    let sqr n = n * n in
    sqr x + sqr y = sqr z;;
```

```
val pythagorean : int -> int -> int -> bool = <fun>
```

Function as first-class citizen

function = yet another expression, introduced with `fun`

```
# fun x -> x+1
```

```
- : int -> int = <fun>
```

```
# (fun x -> x+1) 3;;
```

```
- : int = 4
```

Internally

```
let f x = x+1;;
```

is identical to

```
let f = fun x -> x+1;;
```

Partial application

```
fun x y -> x*x + y*y
```

is the same as

```
fun x -> fun y -> x*x + y*y
```

one can apply a function partially

Example

```
# let f x y = x*x + y*y;;
```

```
val f : int -> int -> int = <fun>
```

```
# let g = f 3;;
```

```
val g : int -> int = <fun>
```

```
# g 4;;
```

```
- : int = 25
```

Partial application

A partial application is a way to return a function

but one can also return a function as the result of a computation

```
# let f x = let x2 = x * x in fun y -> x2 + y * y;;
```

```
val f : int -> int -> int = <fun>
```

a partial application of `f` computes `x*x` only once

Partial application: example

```
# let count_from n =
    let r = ref (n-1) in fun () -> incr r; !r;;
```

```
val count_from : int -> unit -> int = <fun>
```

```
# let c = count_from 0;;
```

```
val c : unit -> int = <fun>
```

```
# c ();;
```

```
- : int = 0
```

```
# c ();;
```

```
- : int = 1
```

Higher-order functions

a function may take functions as arguments

```
# let integral f =
    let n = 100 in
    let s = ref 0.0 in
    for i = 0 to n-1 do
        let x = float i /. float n in s := !s +. f x
    done;
    !s /. float n
```

```
# integral sin;;
```

```
- : float = 0.455486508387318301
```

```
# integral (fun x -> x*.x);;
```

```
- : float = 0.32835
```

Iteration

In Java, one iterates over a collection with a cursor

```
for (Elt x: s) {
    ... do something with x ...
}
```

In OCaml, we typically write

```
iter (fun x -> ... do something with x ...) s
```

where `iter` is a function provided with the data structure, with type

```
val iter: (elt -> unit) -> set -> unit
```

Example

```
iter (fun x -> Printf.printf "%s\n" x) s
```

Difference wrt to function pointers

"in C one can pass and return function pointers"

But OCaml functions are more than function pointers

```
let f x = let x2 = x * x in fun y -> x2 + y * y;;
```

The value of `x2` is captured in a closure

Note: there are closures in Java (≥ 8) too

```
s.forEach(x -> { System.out.println(x); });
```

Recursive functions

In OCaml, it is idiomatic to use recursive functions, for

- a function call is cheap
- tail calls are optimized

Example

```
let zero f =
    let rec lookup i = if f i = 0 then i else lookup (i+1) in
    lookup 0
```

Recursive code ⇒ clearer, simpler to justify

Polymorphism

```
# let f x = x;;
```

```
val f : 'a -> 'a = <fun>
```

```
# f 3;;
```

```
- : int = 3
```

```
# f true;;
```

```
- : bool = true
```

```
# f print_int;;
```

```
- : int -> unit = <fun>
```

```
# f print_int 1;;
```

```
1- : unit = ()
```

Polymorphism

OCaml always infers the most general type

Example

```
# let compose f g = fun x -> f (g x);;
```

```
val compose : ('a -> 'b) -> ('c -> 'a) -> 'c -> 'b = <fun>
```

where `'a` represents a variable of type

Recap

- functions = first-class values: local, anonymous, arguments of other functions, etc.
- partially applied
- polymorphic
- function call is cheap

Memory Allocation

GC

Memory allocation is handled by a garbage collector (GC)

Benefits:

- unused memory is reclaimed automatically
- efficient allocation

$\Rightarrow$ forget about "dynamically allocate is expensive"
... but keep worrying about complexity!

Arrays

- allocation

```
let a = Array.make 10 0
val a : int array = [|0; 0; 0; 0; 0; 0; 0; 0; 0; 0|]
```

- necessary initialized

```
# let a = [| 1; 2; 3; 4 |];;
```

- access

```
# a.(1);;
- : int = 2
```

- assignment

```
    a.(1) <- 5
    - : unit = ()
```

parallel

Java	OCaml
`int[] a = new int[42];`	`let a = Array.make 42 0`
`a[17]`	`a.(17)`
`a[7] = 3;`	`a.(7) <- 3`
`a.length`	`Array.length a`

Example: insertion sort

```
let insertion_sort a =
    let swap i j =
      let t = a.(i) in a.(i) <- a.(j); a.(j) <- t
    in
    for i = 1 to Array.length a - 1 do
      (* insert element a[i] in a[0..i-1] *)
      let j = ref (i - 1) in
      while !j >= 0 && a.(!j) > a.(!j + 1) do
        swap !j (!j + 1); decr j
      done
    done
```

Example: insertion sort

```
let insertion_sort a =
    let swap i j =
      let t = a.(i) in a.(i) <- a.(j); a.(j) <- t
    in
    for i = 1 to Array.length a - 1 do
      (* insert element a[i] in a[0..i-1] *)
      let rec insert j =
        if j >= 0 && a.(j) > a.(j+1) then
        begin swap j (j+1); insert (j-1) end
      in
      insert (i-1)
    done
```

Records

Like in most other languages

a record type is first declared

```
type complex = { re : float; im : float }
```

allocation and initialization are simultaneous:

```
let x = { re = 1.0; im = -1.0 }
```

```
val x : complex = {re = 1.; im = -1.}
```

access with the usual notation:

```
x.im
```

```
- : float = -1.
```

mutable fields

```
type person = { name : string; mutable age : int}
```

```
# let p= {name="Martin";age=23};;
```

```
val p : person = {name = "Martin"; age = 23}
```

Edit in place:

```
# p.age  <- p.age + 1;;
```

```
- : unit = ()
```

```
# p.age;;
```

```
- : int = 24
```

parallel

Java	OCaml

`class T {`
`    final int v; boolean b;`
`    T(int v, boolean b) {`
`        this.v = v; this.b = b;`
`    }`
`}`
	`type t = {`
	`    v: int;`
	`    mutable b:  bool;`
	`}`

`T r = new T(42, true);`	`let r = { v = 42; b = true }`
`r.b = false;`	`r.b <- false`
`r.v`	`r.v`

Reference

a reference = a record of that predefined type

```
type 'a ref = { mutable contents : 'a }
```

`ref`, `!` and `:=` are syntactic sugar

only arrays and mutable fields can be mutated

tuples

usual notation

```
# (1,2,3);;
```

```
- : int * int * int = (1, 2, 3)
```

```
# let v = (1, true, "hello", 'a');;
```

```
val v : int * bool * string * char =
    (1, true, "hello", 'a')
```

access to components

```
# let (a,b,c,d) = v;;
```

```
val a : int = 1
val b : bool = true
val c : string = "hello"
val d : char = 'a'
```

tuples

useful to return several values

```
# let rec division n m =
    if n < m then (0, n)
    else let (q,r) = division (n - m) m in (q + 1, r);;
```

```
val division : int -> int -> int * int = <fun>
```

function taking a tuple as argument

```
# let f (x,y) = x + y;;
```

```
val f : int * int -> int = <fun>
```

```
# f (1,2);;
```

```
- : int = 3
```

lists

predefined type of lists, α `list`, immutable and homogeneous

built from the empty list `[]` and addition in front of a list `::`

```
# let l = 1 :: 2 :: 3 :: [];;
```

```
val l : int list = [1; 2; 3]     - : int = 3
```

shorter syntax

```
# let l = [1; 2; 3];;
```

Pattern matching

pattern matching = case analysis on a list

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

```
val sum : int list -> int = <fun>
```

```
# sum [1;2;3];;
```

```
- : int = 6
```

shorter notation for a function performing pattern matching on its argument

```
let rec sum = function
    | []      -> 0
    | x :: r -> x + sum r;;
```

representation in memory

OCaml lists = identical to lists in C or Java

the list `[1; 2; 3]` is represented as

algebraic data types

lists = particular case of algebraic data type

algebraic data type = union of several constructors

```
type fmla = True | False | And of fmla * fmla
```

```
# True;;
```

```
- : fmla = True
```

```
# And (True, False);;
```

```
- : fmla = And (True, False)
```

lists predefined as

```
type 'a list = [] | :: of 'a * 'a list
```

pattern matching

pattern matching generalizes to algebraic data types

```
# let rec eval = function
    | True  -> true
    | False -> false
    | And (f1, f2) -> eval f1 && eval f2;;
```

```
val eval : fmla -> bool = <fun>
```

pattern matching

patterns can be nested:

```
let rec eval = function
    | True  -> true
    | False -> false
    | And (False, f2) -> false
    | And (f1, False) -> false
    | And (f1, f2)    -> eval f1 && eval f2;;
```

pattern matching

patterns can be omitted or grouped

```
let rec eval = function
    | True  -> true
    | False -> false
    | And (False, _) | And (_, False) -> false
    | And (f1, f2) -> eval f1 && eval f2;;
```

parallel

Java

```
abstract class Fmla { }
class True extends Fmla { }
class False extends Fmla { }
class And extends Fmla {
    Fmla f1, f2; }

abstract class Fmla {
    abstract boolean eval(); }
class True { boolean eval() {
    return true; } }
class False { boolean eval() {
    return false; } }
class And { boolean eval() {
    return f1.eval()&&f2.eval();
}}
```

OCaml

```
type fmla =
    | True
    | False
    | And of fmla * fmla


let rec eval = function
| True -> true

| False -> false

| And (f1, f2) ->
    eval f1 && eval f2

        .
```

pattern matching

pattern matching is not limited to algebraic data types

```
let rec mult = function
    | []     -> 1
    | 0 :: _ -> 0
    | x :: l -> x * mult l
```

one may write `let pattern = expression` when there is a single pattern
(as in `let (a,b,c,d) = v` for instance)

recap

- allocation is cheap
- memory is reclaimed automatically
- allocated values are necessarily initialized
- most values cannot be mutated
  (only arrays and mutable record fields can be)
- efficient representation of values
- pattern matching = case analysis over values

Execution model

Values

A value is

- either a primitive value (integer, floating point, Boolean, `[]`, etc.)
- or a pointer (to an array, a constructor such as `And`, etc.)

It fits on 64 bits

Passing mode is by value

In particular, no value is ever copied

It is exactly as in Java

No `null` value

In OCaml, there is no such thing as `null`

In particular, any value is necessarily initialized

Sometimes a pain, but it's worth the effort:

an expression of type τ whose evaluation terminates necessarily has a legal value of type τ

This is known as strong typing

No such thing as `NullPointerException`
(neither `segmentation fault` as in C/C++)

Comparison

Equality written `==` is physical equality,
that is, equality of pointers or primitive values

```
# (1, 2) == (1, 2);;
```

```
- : bool = false
```

as in Java

Equality written `=`, on the contrary, is `structural equality`,
that is, recursive equality descending in sub-terms

```
# (1, 2) = (1, 2);;
```

```
- : bool = true
```

it is equivalent to `equals` in Java (when suitably defined)

Exceptions

Exceptions

Usual notion

an exception may be raised

```
let division n m =
     if m = 0 then raise Division_by_zero else ...
```

and later caught

```
try division x y with Division_by_zero -> (0,0)
```

one can introduce new exceptions

```
exception Error
exception Unix_error of string
```

Idiom

in OCaml, exceptions are used in the library to signal exceptional behavior

Example

`Not_found` to signal a missing value

```
try
    let v = Hashtbl.find table key in
    ...
with Not_found ->
    ...
```

(where Java typically returns `null`)

modules and functors

Software engineering

When programs get big we need to

- split code into units (modularity)
- hide data representation (encapsulation)
- avoid duplicating code

In OCaml, this is provided by modules

files and modules

Each file is a module

If `arith.ml` contains

```
let pi = 3.141592
let round x = floor (x +. 0.5)
```

we compile it with

```
% ocamlopt -c arith.ml
```

We use it within another module `main.ml`:

```
let x = float_of_string (read_line ());;
print_float (Arith.round (x /. Arith.pi));;
print_newline ();;
```

```
% ocamlopt -c main.ml
```

```
% ocamlopt arith.cmx main.cmx
```

Encapsulation

we can limit what is exported with an interface

in a file `arith.mli`

```
val round : float -> float
```

```
% ocamlopt -c arith.mli
% ocamlopt -c arith.ml
```

```
% ocamlopt -c main.ml
File "main.ml", line 2, characters 33-41:
Unbound value Arith.pi
```

Encapsulation

An interface may also hide the definition of a type

in `set.ml`

```
type t = int list
let empty = []
let add x l = x :: l
let mem = List.mem
```

but in `set.mli`

```
type t
val empty : t
val add : int -> t -> t
val mem : int -> t -> bool
```

type `t` is an abstract type

Separate compilation

the compilation of a file only depends on the interfaces of the other files

$\Rightarrow$ fewer recompilation when a code changes but its interface does not

Module system

Not limited to files

```
module M = struct
    let c = 100
    let f x = c * x
end
```

```
module A = struct
    let a = 2
    module B = struct
      let b = 3
      let f x = a * b * x
    end
    let f x = B.f (x + 1)
end
```

Module system

Similar for interfaces

```
module type S = sig
    val f : int -> int
end
```

interface constraint

```
module M : S = struct
    let a = 2
    let f x = a * x
end
```

```
# M.a;;
```

```
Unbound value M.a
```

Recap

- code split into units called modules
- encapsulation of types and values, abstract types
- separate compilation
- organizes the name space

Functors

functor = module parameterized with other modules

Example (Hash table)
one has to parameterize wrt hash function and equality function

the solution: a functor

```
module type HashedType = sig type elt
    val hash: elt -> int
    valeq :elt->elt->bool
end
```

```
module HashTable(X: HashedType) = struct ... end
```

Functor definition

```
module HashTable(X: HashedType) = struct
    type t = X.elt list array
    let create n = Array.make n []
    let add t x =
        let i = (X.hash x) mod (Array.length t) in
        t.(i) <- x :: t.(i)
    let mem t x =
        let i = (X.hash x) mod (Array.length t) in
        List.exists (X.eq x) t.(i)
end
```

Inside, `X` is used as any regular module

Functor definition

```
module HashTable(X: HashedType) : sig
    type t
    val create : int -> t
    val add : t -> X.elt -> unit
    val mem : t -> X.elt -> bool
end
```

Functor use

```
module Int = struct
    type elt = int
    let hash x = abs x
    let eq x y = x=y
end
```

```
module Hint = HashTable(Int)
```

```
# let t = Hint.create 17;;
```

```
val t : Hint.t = <abstr>
```

```
# Hint.add t 13;;
```

```
- : unit = ()
```

```
# Hint.add t 173;;
```

```
- : unit = ()
```

parallel

Java	OCaml
`interface HashedType<T> {`	`module type HashedType = sig`
	`    type elt`
`    int hash();`	`    val hash:  elt -> int`
`    boolean eq(T x);`	`    val eq:  elt -> elt -> bool`
`}`	`end`
`class HashTable`	`module HashTable(E: HashedType) =`
`    <E extends HashedType<E>> {`	`    struct`
`        ...`	`    ...`

Applications of functors

1. data structures parameterized with other data structures
   - `Hashtbl.Make` : hash tables
   - `Set.Make` : finite sets implemented with balanced trees
   - `Map.Make` : finite maps implemented with balanced trees
2. algorithms parameterized with data structures
   (e.g. DFS algorithm)

Dijkstra's algorithm

```
module Dijkstra
    (G: sig
        type graph
        type vertex
        val succ: graph -> vertex -> (vertex * float) list
      end) :
    sig
    val shortest_path:
      G.graph -> G.vertex -> G.vertex -> G.vertex list * float
    end
```

persistence

Immutable data structures

In OCaml, most data structures are immutable
(exceptions are arrays and records with `mutable` fields)

In other word:

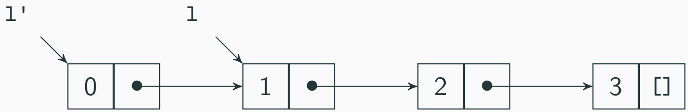
- a value is not modified by an operation,
- but a new value is returned

Terminology: this is called applicative programming or functional programming

Example of immutable structure: lists

```
let l = [1; 2; 3]
```

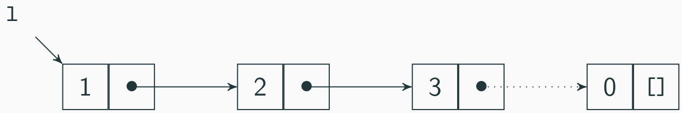
```
let l' = 0 :: l
```



no copy, but sharing

counterpart

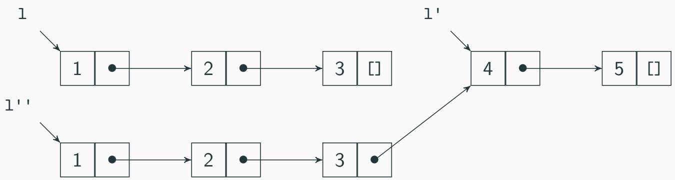
adding an element at the end of the list is not simple:



Concatenating two lists

```
let rec append l1 l2 = match l1 with
    | [] -> l2
    | x :: l -> x :: append l l2
```

```
let l = [1; 2; 3]
let l' = [4; 5]
let l'' = append l l'
```



blocs of `l` are copied, blocs of `l'` are shared

mutable linked lists

Note: one can implement traditional linked lists,
for instance with

```
type 'a mlist = Empty | Element of 'a element
and 'a element = { value: 'a; mutable next: 'a mlist }
```

but then be careful with sharing (*aliasing*)

Another example: trees

```
type tree = Empty | Node of int * tree * tree

val add : int -> tree -> tree
```

Again, few copies and mostly sharing

Benefits of persistence

1. correctness of programs
    - code is simpler
    - mathematical reasoning is possible
2. easy to perform backtracking
    - search algorithms
    - symbolic manipulation and scopes
    - error recovery

Persistence and backtracking

search for a path in a maze

```
type state
val is_exit : state -> bool
type move
val moves : state -> move list
val move : state -> move -> state
```

```
let rec search e =
    is_exit e || iter e (moves e)
and iter e = function
    | []     -> false
    | d :: r -> search (move d e) || iter e r
```

Without persistence

with a mutable, global state

```
let rec search () =
    is_exit () || iter (moves ())
and iter = function
    | []     -> false
    | d :: r -> (move d; search ()) || (undo d; iter r)
```

i.e. one has to undo the side effect
(here with a function `undo`, inverse of `move`)

Persistence and backtracking (2)

simple Java fragments, represented with

```
type stmt =
    | Return of string
    | Var    of string * int
    | If     of string * string * stmt list * stmt list
```

Example

```
int x = 1;
int z = 2;
if (x == z) {
    int y = 2;
    if (y == z) return y; else return z;
}   else
    return x;
```

Persistence and backtracking (2)

let us check that any variable which is used was previously declared
(within a list of statements)

```
val check_stmt : string list -> stmt -> bool
val check_prog : string list -> stmt list -> bool
```

Persistence and backtracking (2)

```
let rec check_instr vars = function
    | Return x ->
        List.mem x vars
    | If (x, y, p1, p2) ->
        List.mem x vars && List.mem y vars &&
        check_prog vars p1 && check_prog vars p2
    | Var _ ->
        true
```

```
and check_prog vars = function
    | [] ->
        true
    | Var (x, _) :: p ->
        check_prog (x :: vars) p
    | i :: p ->
        check_instr vars i && check_prog vars p
```

Persistence and backtracking (3)

a program handles a database

non atomic updates, requiring lot of computation

with a mutable state

```
try
    ... performs update on the database ...
with e ->
    ... rollback database to a consistent state ...
    ... handle the error ...
```

Persistence and backtracking (3)

with a persistent data structure

```
let bd = ref (... initial database ...)
    ...
try
    bd := (... compute the update of !bd ...)
with e ->
    ... handle the error ...
```

Interface and persistence

the persistent nature of a type is not obvious

the signature provides implicit information

mutable data structure

```
type t
val create : unit -> t
val add : int -> t -> unit
val remove : int -> t -> unit
...
```

persistent data structure

```
type t
val empty : t
val add : int -> t -> t
val remove : int -> t -> t
...
```

Persistence and side effects

persistence does not mean absence of side effects

$$\text{persistent} = \text{observationally immutable}$$

only one way

$$\text{immutable} \Rightarrow \text{persistent}$$

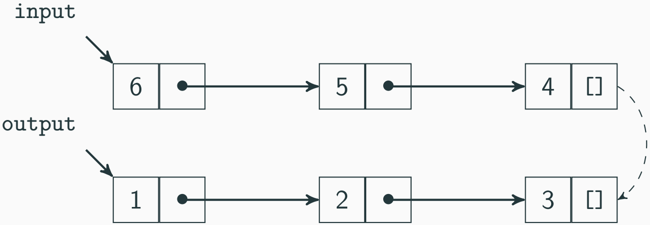
the reciprocal is wrong

Example: persistent queues

```
type 'a t
val create : unit -> 'a t
val push : 'a -> 'a t -> 'a t
exception Empty
val pop : 'a t -> 'a * 'a t
```

Example: persistent queues

Idea: a queue is a pair of lists,
one for insertion, and one for extraction



stands for the queue $\rightarrow 6, 5, 4, 3, 2, 1 \rightarrow$

Example: persistent queues

```
type 'a t = 'a list * 'a list

let create () = [], []

let push x (e,s) = (x :: e, s)

exception Empty

let pop = function
    | e, x :: s -> x, (e,s)
    | e, [] -> match List.rev e with
        | x :: s -> x, ([], s)
        | [] -> raise Empty
```

Example: persistent queues

When accessing several times the same queue whose second list is empty, we reverse several times the same list

Let us add a reference to register the list reversal the first time it is performed

```
type 'a t = ('a list * 'a list) ref
```

The side effect is done “under the hood”, in a way not observable from the user, the contents of the queue staying the same

Example: persistent queues

```
let create () = ref ([], [])

let push x q = let e,s = !q in ref (x :: e, s)

exception Empty
```

```
let pop q = match !q with
    | e, x :: s -> x, ref (e,s)
    | e, [] -> match List.rev e with
        | x :: s as r -> q := [], r; x, ref ([], s)
        | [] -> raise Empty
```

Recap

- persistent structure = no observable modification
  - in OCaml: `List`, `Set`, `Map`
- can be very efficient (lot of sharing, hidden side effects, no copies)
- idea independent of OCaml