

```
Calling Functions

Repeated ("hello", 3) /* C code */

Repeated "hello" 3 (* OCaml code *)

Repeated ( print_string ("hello"), 3) /* C code */

Repeated (print_string "hello") 3 (* OCaml code *)
```

```
Defining a Function

#let average a b = (a + .b) /, 20;
val average : float -> float -> float = <fun>

/*c code */
double average (double a, double b) {
    return (a+b)/2;
}
```

```
Defining a Function

#let average a b = (a + .b) /. 2.0;
val average : float -> float -> float = \( \frac{fun}{} \)

Auto inferring type Strongly Statically Typed Language
No implicit casting!

/*c code */
double average (double a, double b) {
    return (a+b)/2;
}
```

```
Basic Type

• int: 31-bit signed int (roughly +/- 1 billion) on 32-bit processors, or 63-bit signed int on 64-bit processors

• nativeint

• Big_int

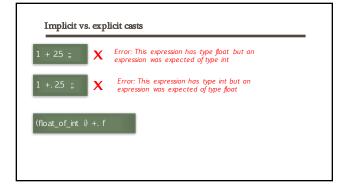
• float: IEEE double-precision floating point, equivalent to C's double

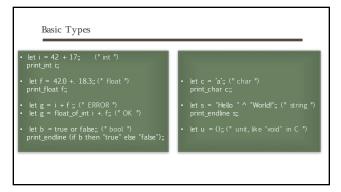
• bool: A boolean, written either true or false

• char: An 8-bit character string A string

• unit: Written as ()

• similar to void
```





```
Recursive Function

By default the function name is not visible in its default expression

#let rec range a b =
    if a > b then ||
    else a : range (a+1) b;;
val range : int -> int -> int list = \langle fun\rangle

#let rec fac n =
    if n \langle 2 then 1
    else n * fac n-1;;
else n * fac1 n
    and fac1 n = fac (n-1);

Mutual Recursion
```

Types derived from Function declaration

- f : arg1 → arg2 → ... → argn → rettype
- int_to_char
- Repeated "hello" 3 ?
- average a b = (a + .b) /. 2.0;
- output_char

```
#let give_three x = 3;
val give_theree: 'a -> int = \( \frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\fir}{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac{\frac
```

Automatic Type Inference: How?

#let average a b = (a +. b) /. 20; val average : float -> float -> float = $\langle fun \rangle$

- 1. checks where a and b are used? \rightarrow (a +. b)
- 2 +. function always takes two floats \Rightarrow a and b have to be floats
- 3. /. function return float \rightarrow retrun type has to be float

Structural vs. Physical Equality

- Structural equality (=) compares values;
- physical equality (==) compares pointers.
- Compare strings and floating-point numbers structurally.

	Structural comparison (polymorphic)
== <u> </u> =	Physical comparison (polymorphic)
⟨ ⟩ ⟨= ⟩=	Comparisons (polymorphic)

If then else

- if expr1 then expr2 else expr
- It is an expression in ocaml
- Else part is compulsory
- if 3 = 4 then 42 else 17;
- if "a" = "a" then 42 else 17;;
- if true then 42 else "17";;
 - This expression has type string but is here used with type int

Local "variable" /*c code */ double average (double a, double b) { double sum = a+b; return sum/2; } #let average a b = let sum = (a + b) in sum / . 20; val average : float -> float -> float = <fun) The standard phrase let name = expr1 in expr2 is used to define a named local expression, and name can then be used later on in the function instead of expression, the function instead of expression ends the block of code.

#let avg = let average a b = let sum = (a +. b) in sum /. 20; val average : float -> float -> float = <fun> bind name to expression in everything that follows let name = expression Any use of let ..., whether at the top level (globally) or within a function, is often called a let-binding.

```
References: real variables

• Create a reference : ref 0;

• A reference is created, but as we did not name it a garbage collector will immediately collect it.

• Name it: let my_ref = ref 0;

• This reference is currently storing a zero integer.

• my_ref := 100; ← updating the value

• printing reference: lmy_ref

The := operator is used to assign to references, and the ! operator dereferences to get out the contents.
```

```
# tet my_ref = ref 0;
val my_ref: int ref = {contents = 0}
# my_ref: 100;
-: unit = ()
# lmy_ref;
-: int = 100

/*c code */
int a = 0;
int *my_ptr = &a;
*my_ptr = 100;
*my_ptr = 100;
```

```
# let read_whole_channel chan =

let buf = Buffer.create 4096 in

let rec loop () =

let newline = input_line chan in

Buffer.add_string buf newline;

Buffer.add_char buf `\n';

loop ()

in try

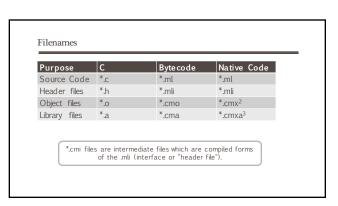
loop ()

with

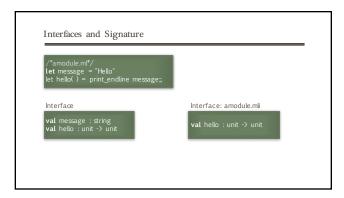
End_of_file -> Buffer.contents buf;;

val read_whole_channel : in_channel -> string = \( \frac{4}{3} \text{un} \rangle \)
```

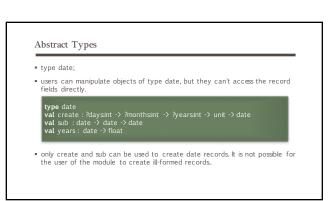
Modules Ocaml has modules in /usr/lib/ocaml/ For example, to use the functions in Graphics module, simply do: open Graphics; at top and call its functions, say open_graph later or Graphics.open_graph "640x480"; (* To compile this example: ocamlc graphics.cma grtest.ml - o grtest *) Rename module: module Gr = Graphics; Gr.open_graph

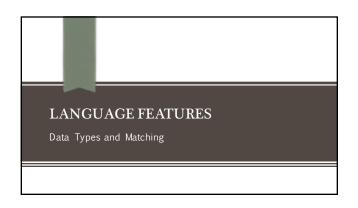


Creating Modules In OCaml, every piece of code is wrapped into a module. A module itself can be a submodule of another module, pretty much like directories in a file system-but we don't do this very often. Amodule.ml*/ let hello() = print_endline "Hello"



Types • modules often define new types • type name = typedef type date = { day : int; month : int; year : int } • There are four options when it comes to writing the corr. .mli file: 1. The type is completely omitted from the signature. 2. The type definition is copy-pasted into the signature. 3. The type is made abstract: only its name is given. (eg. type date) 4. The record fields are made read-only: type date = prwate { ... }





Linked Lists OCaml has support for lists built into the language. All elements of a list in OCaml must be the same type. I [12;3]; Empty list = [] Head: the first element 1 Tail: the rest of the list [2,3] Alternate ways to write list: [12;3] 1:22:3] 1:22:3]

Type of Linked List The type of a linked list of integers is int list the type of a linked list of foos is foo list. 'a list? Type of anything Does not mean that each individual element has a different type A linked list must have the same type. But the type can be polymorphic List.length: 'a list -> int

```
Structures

/*c code */
struct pair_of_ints {
    int a, b;
};

Tuples: Pairs or tuples of different types separated by commas.

(3,4)
    -: int * int = (3, 4)
    (3, "hello", "X)
    -: int * string * char = (3, "hello", "X)

Records: allow name of the elements

# type pair_of_ints = { a : int; b : int };;

(*Use*)
# {a = 3; b = 5}
    -: pair_of_ints = {a = 3; b = 5}
```

```
Variants (qualified unions and enums)

/*c code */
struct foo {
    int type;
    #define TYPE_INT 1
    #define TYPE_STRING 3
    union {
        int i; // if type == TYPE_INT.
        int pair/2l; // if type == TYPE_STRING.
    } u;
    } u;
};
```

```
Variants (qualified unions)

/*c code */
struct foo {
    int type;
    #define TYPE_INT 1
#define TYPE_PAIR_OF_INTS 2
#define TYPE_PAIR_OF_INTS 2
#define TYPE_STRING 3
    union {
     int i; // if type == TYPE_INT.
     int pairIzl; // if type == TYPE_STRING.
    } u;
} u;
};

Vse:
Nothing
    foo Nothing
    int 3
    foo Nothing
    int 3
         foo Nothing
    int 3
         foo Pair (4, 5)
         String "hello"
         foo Sing "hello"
```

```
Variants (enums)

/*c code */
enum sign { positive, zero, negative };

/*OCaml code */
# type sign = Positive | Zero | Negative
```



```
# type 'a equiv_list =

| Nil
| Cons of 'a * 'a equiv_list;;
type 'a equiv_list = Nil | Cons of 'a * 'a equiv_list

# Nil;

-: 'a equiv_list = Nil
# Cons(1, Nil);

-: int equiv_list = Cons (1, Nil)
# Cons(1, Cons(2, Nil));

-: int equiv_list = Cons (1, Cons (2, Nil))
```

Commana Example type definition Example usage list int list [1; 2; 3] (3, "hello") (a = 3; b = "hello") (a = 3; b

```
Pattern Matching

• match value with
| pattern -> result
| pattern -> result ...
```

```
Pattern Matching

# let xor p = match p
with (false, false) > false
| (false, false) > tase
| (false, false) > false;

val xor: bool * bool > bool = <false

A name in a pattern matches arrything and is bound when
the pattern matches. Each may appear only once per pattern.

# let xor p = match p
with (false, x) > x
| (false, x) > x
| (false, x) > x
| (false, x) > tase | false
| wal xor: bool * bool > bool = <false|
# xor (false, false) | false|
# xor (false) | false|
# xo
```

Pattern Matching with Wildcards Underscore () is a wildcard that will match anything, useful as a default or when you just don't care. #let xor p= match p with (me, false) | (false, me) > me | -> false; | -> false; | val xor : bool * bool >> bool = cfan> # xor (me, tue); -: bool = false # xor (me, false); -: bool = false # xor (me, false); -: bool = false | false | false | false | false | | false | false | false | false | | false | false | false | false | | false | false | false | false | | false | false | false | false | | false | false | false | false | | false | false | false | false | | false | false | false | false | | false | false | false | false | | false | false | false | false | | false | false | false | false | | false | false | false | false | | false | false | false | false | | false | false | false | false | | false | false | false | false | | false | false | false | false | | false | false | false | false | | false | false | false | false | | false | false | false | false | | false | false | false | false | | false | false | false | false | | false | false | false | false | | false | false | false | false | | false | false | false | false | | false | false | false | false | | false | false | false | false | | false | false | false | false | | false | false | false | false | | false | false | false | false | | false | false | false | false | | false | false | false | false | | false | false | false | false | | false | false | false | false | | false | false | false | false | | false | false | false | false | | false | false | false | false | | false | false | false | false | | false | false | false | false | | false | false | false | false | | false | false | false | false | | false | false | false | false | | false | false | false | false | | false | false | false | false | | false | false | false | false | | false | false | false | false | | false | false | false | false | | false | false | false | false | | false | false | false | false | | false | false | false | false | | false | false | f

```
Pattern Matching with when and as

The "when" keyword lets you add a guard expression:

# let tall = function

| (0, s) when h > 180 ~ s ^ " is tall"
| (0, s) ~ s ^ " is short";

val tall: int * string ~ string = 6tm>

# List.ramp tall (1(83, "Sephen"); (150, "Nima"));
| : string list = ["Sephen is tall"; "Nima is short"]

The "as" keyword lets you name parts of a matched structure:

# match ((3,9), 4) with
| (a sa xa, 4) ~ xa
| (a 0,0);
| -: int * int = (3, 9)
```

```
Pattern Matching

n * (x + y)

# let rec to_string e = 
match e with
| Plus (left, right) > "(" ^ to_string left ^ " + " ^ to_string right ^ ")"
| Minus (left, right) > "(" ^ to_string left ^ " - " ^ to_string right ^ ")"
| Times (left, right) > "(" ^ to_string left ^ " - " ^ to_string right ^ ")"
| Divide (left, right) > "(" ^ to_string left ^ " - " ^ to_string right ^ ")"
| Value v > v;
| val to_string: lesepr > string = (fun)

# let print_expr e = print_endine (to_string e);;
val print_expr (Times (Value "n", Plus (Value "x", Value "y")));;
(n " (x + y))
| - : unit = 0
```

```
Pattern Matching

n^{*}(x+y) ==> (n^{*}x+n^{*}y)

# let rec multiply_out e = match e with | Times (e1, Plus (e2, e3)) > Plus (Times (inhliply_out e1, multiply_out e3)) | Times (e1, Plus (e2, e3)) > Plus (Times (inhliply_out e1, multiply_out e3)) | Times (Plus (e1, e2), e3) > Plus (Times (inhliply_out e1, multiply_out e3)) | Plus (limes (inhliply_out e1, multiply_out e3)) | Plus (let, night) > Plus (inhliply_out e1, multiply_out e1, multiply_out night) | Minus (let, night) > Minus (inhliply_out e1, multiply_out night) | Times (let, night) > Minus (multiply_out e1, multiply_out night) | Divide (let, night) > Divide (multiply_out let, multiply_out night) | Value v > Value v :: val multiply_out : expr > expr = (fun>) | value v | valu
```



Map Creates a "mapping". module MyUsers = Map.Make(String);; (* create a map MyUsers *) # let m = MyUsersempty; val m : 'a MyUserst = Kabstr> (*create an empty map*) # let m = MyUsersadd "fred" "sugarplums" m; val m : string MyUserst = Kabstr> (* add something to it *) Once we have added the string "sugarplums" we have fixed the types of mappings that we can do


```
List: Some useful List Function

Three great replacements for loops:

• List.map f [a1; ... ;an] = [f a1; ... ;f an]

• Apply a function to each element of a list to produce another list.

• List.fold_left f a [b1; ...;bn] = f (...(f (f a b1) b2)...) bn

• Apply a function to a partial result and an element of the list to produce the next partial result.

• List.iter f [a1; ...;an] = begin f a1; ...; f an; () end

• Apply a function to each element of a list; produce a unit result.

• List.rev [a1; ...; an] = [an; ...;a1]

• Reverse the order of the elements of a list.
```

```
# Listmap (fun a > a + 10) [42; 17; 128];

: :int list = [52; 27; 138]

# Listmap sting of int [42; 17; 128];

: :sbing list = [42; 47; 17; 128];

# Listfold_left (fun s e > s + e) 0 [42; 17; 128]; Listfold_left fa [b1; ...;bn] = f (...(f (f a b1) b2)...) bn

: :int = 187

# Listiter print_int [42; 17; 128];

4217128 : unit = 0

# Listiter (fun n > print_int n; print_newline (j) [42; 17; 128];

42 17 128 :: unit = 0

# Listiter print_endine (Listmap string_of_int [42; 17; 128]);

42 17 128 :: unit = 0
```

```
# let a = || 42, 17; 19 ||; (* Array literal *)

val a : int array = || 42, 17; 19 ||

# let aa = Array.make 5 0; (* Fill a new array *)

val aa : int array = || 0, 0, 0, 0, 0||

# a (0);

- : int = 42

# a (2);

- : int = 19

# a (3);

Exception: Invalid_argument "index out of bounds".

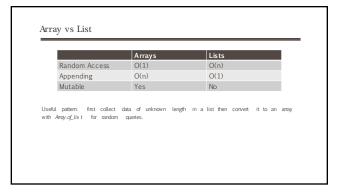
# a (2) < 20; (* Arrays are mutable! *)

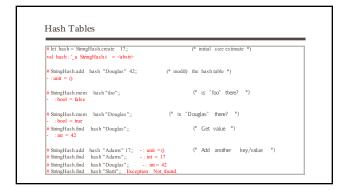
- : unit = 0

# a;

- : int array = || 42, 17; 20||
```

let l = |24, 32, 17|: | val l : int list = |24, 32, 17| | # let b = Array C[ist l; | val b : int array = ||24, 32, 17|| | # let c = Array.append a b; | val c : int array = ||42, 17; 20, 24, 32, 17||







Functional Language

• Functions are first class citizen

```
# let twice x = x * 2 in List.map twice [ 1; 2; 3 ];
-: int list = [2; 4; 6]
```

- \bullet The nested function twice takes an argument x and returns x * 2.
- \blacksquare Then map calls twice on each element of the given list ([1; 2; 3]) to produce the result: a list with each number doubled.
- \bullet map is known as a higher-order function (HOF): a function takes a function as one of its arguments.

let sqr x = x * x;; - : val sqr : int -> int = <fun> # sqr 5 (* calling *) # let appadd = fun f -> (f 42) + 17;; val appadd : (int -> int) -> int = <fun> # let plus5 x = x + 5; val plus5 : int -> int = <fun> # appadd plus5;; - : int = 64

Functional Language

· Functions are first class citizen

```
# let twice x = x * 2 in List.map twice [ 1; 2; 3 ];;
-: int list = [2; 4; 6]
```

- The nested function twice takes an argument x and returns x * 2.
- \bullet Then map calls twice on each element of the given list ([1; 2; 3]) to produce the result: a list with each number doubled.
- \blacksquare map is known as a $higher\hbox{-}order$ function (HOF): a function takes a function as one of its arguments.

Functional Language

```
#let multiply n list =
let f x = n * x in
 List.map f list;;
val multiply : int -> int list -> int list = <fun>
```

- nested function f is a closure.
 Closures are functions which carry around some of the "environment" in which they were defined.
 f uses the value of n which isn't passed as an argument but available in its environment
- map is defined in the List module
- we're passing f into the module defined somewhere else.
- the closure will ensure that f always has access back to its parental environment, and to n.

Currying

```
# let plus a b = a + b;;
val plus : int -> int -> int = (fun>
```

- plus??
- plus 2 3??
- plus 2??

let plus2 plus 2;; val plus2: int -> int = \(fun \)

• plus2 5??

Currying
Given a function
f. (X × Y) → Z,
currying constructs a new function
h:X>(Y>Z)

Currying

```
#let multiply n list =
let f x = n * x in
List.map f list;;
val multiply : int -> int list -> int list = ⟨fun⟩
```

- let double = multiply 2;;
- let triple = multiply 3;;

Other features

- · Pure vs impure
- Strictness vs laziness
- Boxed vs. unboxed types

