Programming Languages & Translators

INTRODUCTION TO OCAML

Baishakhi Ray

Fall 2018

These slides are motivated from Prof. Stephen Edwards

Useful Link

- https://ocaml.org/learn/tutorials/
- http://www.cs.columbia.edu/~%20sedwards/classes/2017/4115-fall/ocaml.pdf

Basics: Running OCaml Code

```
$ocaml
OCaml version <version>

# 1+1;;
- : int = 2
```

Command Line

Hello.ml

```
let hello = "Hello World!"
let _ = print_endline hello
```

Run it with the interpreter:

```
$ocaml hello.ml
Hello World!
```

Compile into bytecode and run:

```
$ocamlc -o hello hello.ml
$./hello Hello World!
```

Compile an executable and run:

```
$ ocamlopt -o hello hello.ml
$ ./hello Hello World!
```

Comments: (* …. *)

```
(* This is a single-line comment. *)

(* This is a
* multi-line
* comment.
*)
```

```
(* a + b

Is a broken comment

You have to close it *)
```

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) /. 2.0;;
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 = <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

Implicit vs. explicit casts

```
1 + 2.5 ;;
```

X *Error: This expression has type float but an expression was expected of type int*

```
1 +. 2.5 ;;
```

X *Error: This expression has type int but an expression was expected of type float*

```
(float_of_int i) +. f
```

Basic Types

```
• let i = 42 + 17;;      (* int *)
  print_int i;;

• let f = 42.0 +. 18.3;; (* float *)
  print_float f;;

• let g = i + f ;; (* ERROR *)
• let g = float_of_int i +. f;; (* OK *)

• let b = true or false;; (* bool *)
  print_endline (if b then "true" else "false");;
```

```
• let c = 'a';; (* char *)
  print_char c;;

• let s = "Hello " ^ "World!";; (* string *)
  print_endline s;;

• let u = ();; (* unit, like "void" in C *)
```

Standard Operators and Functions

+ - * / mod	Integer arithmetic
+. -. *. /. **	Floating-point arithmetic
ceil floor sqrt exp log log10 cos sin tan acos asin atan	Floating-point functions
not && ||	Boolean operators

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 = <fun>
```

```
#let rec fac n =
      if n < 2 then 1
      else n * fac n-1;;
```

```
#let rec fac n =
      if n < 2 then 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

Polymorphic Functions

```
#let give_three x = 3;;
val give_theree: 'a -> int = <fun>
```

'a : "any type you fancy"

Automatic Type Inference: How?

```
#let average a b = ( a +. b) /. 2.0;;
val average : float -> float -> float = <fun>
```

1. checks where a and b are used? ➔ (a +. b)
2. +. function always takes two floats ➔ a and b have to be floats
3. /. function return float ➔ 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)
== !=	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 /. 2.0;;
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, till a ;; which ends the block of code.

Global variables

```
#let avg =
  let average  a b =
  let sum =  ( a +. b)  in
  sum /. 2.0;;
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: !my_ref

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

References: real variables

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

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

Nested Function

```
# 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 = <fun>
```

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*

Filenames

Purpose	C	Bytecode	Native Code
Source Code	*.c	*.ml	*.ml
Header files	*.h	*.mli	*.mli
Object files	*.o	*.cmo	*.cmx[2]
Library files	*.a	*.cma	*.cmxa[3]

*.cmi files are intermediate files which are compiled forms of the .mli (interface or "header file").

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"
```

```
/*bmodule.ml*/
Amodule.hello ()
```

Interfaces and Signature

```
/*amodule.ml*/
let message = "Hello"
let hello( ) = print_endline message;;
```

Interface

```
val message : string
val hello : unit -> unit
```

Interface: amodule.mli

```
val hello : unit -> unit
```

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 = private { ... }

Abstract Types

- type date;
- users can manipulate objects of type date, but they can't access the record fields directly.

```
type date
val create : ?days:int -> ?months:int -> ?years:int -> unit -> date
val sub : date -> date -> date
val years : date -> float
```

- only create and sub can be used to create date records. It is not possible for the user of the module to create ill-formed records.

LANGUAGE FEATURES

Data Types and Matching

Linked Lists

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

Unlike list, different types of elements are allowed

Variants (qualified unions and enums)

```
/*c code */
struct foo {
  int type;
#define TYPE_INT 1
#define TYPE_PAIR_OF_INTS 2
#define TYPE_STRING 3
  union {
    int i; // If type == TYPE_INT.
    int pair[2]; // If type == TYPE_PAIR_OF_INTS.
    char *str; // If type == TYPE_STRING.
  } u;
};
```

Variants (qualified unions)

```
/*c code */
struct foo {
  int type;
#define TYPE_INT 1
#define TYPE_PAIR_OF_INTS 2
#define TYPE_STRING 3
  union {
    int i; // If type == TYPE_INT.
    int pair[2]; // If type == TYPE_PAIR_OF_INTS.
    char *str; // If type == TYPE_STRING.
  } u;
};
```

```
# type foo =
   | Nothing
   | Int of int
   | Pair of int * int
   | String of string;;
```

Use:

```
Nothing
-: foo Nothing
Int 3
-: foo Int 3
Pair (4, 5)
-: foo Pair (4, 5)
String "hello"
-: foo String "hello"
```

Variants (enums)

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

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

Recursive variants (used for trees)

```
/*OCaml code */
# type binary_tree =
        | Leaf of int
        | Tree of binary_tree * binary_tree;;
type binary_tree = Leaf of int | Tree of binary_tree * binary_tree
```

```
Leaf 3
Tree (Leaf 3, Leaf 4)
Tree (Tree (Leaf 3, Leaf 4), Leaf 5)
Tree (Tree (Leaf 3, Leaf 4), Tree (Tree (Leaf 3, Leaf 4), Leaf 5))
```

Parameterized variants

```
/*OCaml code */
# type ‘a binary_tree =
        | Leaf of ‘a
        | Tree of binary_tree * binary_tree;;
type ‘a binary_tree = Leaf of ‘a | Tree of ‘a binary_tree * ‘a binary_tree
```

```
Leaf 3
Tree (Leaf 3, Leaf 4)
Tree (Tree (Leaf 3, Leaf 4), Leaf 5)
Tree (Tree (Leaf 3, Leaf 4), Tree (Tree (Leaf 3, Leaf 4), Leaf 5))
```

Parameterized variants

```
# 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))
```

List, Structure, Variant---summary

```
OCaml name     Example type definition        Example usage

list           int list                       [1; 2; 3]
tuple          int * string                   (3, "hello")
record         type pair =                    { a = 3; b = "hello" }
                 { a: int; b: string }
variant        type foo =
                 | Int of int                 Int 3
                 | Pair of int * string
variant        type sign =
                 | Positive                   Positive
                 | Zero                       Zero
                 | Negative
parameterized  type 'a my_list =
variant          | Empty                      Cons (1, Cons (2, Empty))
                 | Cons of 'a * 'a my_list
```

Pattern Matching

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

Pattern Matching

```
# let xor p = match p
  with (false, false) -> false
     | (false, true) -> true
     | ( true, false) -> true
     | ( true, true) -> false;;
val xor : bool * bool -> bool = <fun>
# xor (true, true);; - : bool = false
```

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

```
# let xor p = match p
  with (false, x) -> x
     | ( true, x) -> not x;;
val xor : bool * bool -> bool = <fun>
# xor (true, true);; - : bool = false
```

Pattern Matching: Case Coverage

The compiler warns you when you miss a case or when one is redundant (they are tested in order):

```
# let xor p = match p
  with (false, x) -> x
     | ( x, true) -> not x;;
Warning P: this pattern-matching is not exhaustive. Here is an example of a value that is
not matched: (true, false) val xor : bool * bool -> bool = <fun>
val xor : bool * bool -> bool = <fun>

# let xor p = match p
  with (false, x) -> x
     |(true, x)->notx
     | (false, false) -> false;;
Warning U: this match case is unused.
val xor : bool * bool -> bool = <fun>
```

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 (true, false) | (false, true) -> true
     | _ -> false;;
val xor : bool * bool -> bool = <fun>

# xor (true, true);;  - : bool = false
# xor (false, false);; - : bool = false
# xor (true, false);;  -: bool = true

# let logand p = match p with
 (false, _) -> false
 | (true, x) -> x;;
val logand : bool * bool -> bool = <fun>
# logand (true, false);; - : bool = false
# logand (true, true);;  -: bool = true
```

Pattern Matching with Lists

```
# let length = function   (* let length = fun p -> match p with *)
      [] -> "empty"
    | [_] -> "singleton"
    | [_; _] -> "pair"
    | [_; _; _] -> "triplet"
    | hd :: tl -> "many";;
val length : 'a list -> string = <fun>

# length [];; - : string = "empty"
# length [1; 2];; - : string = "pair"
# length ["foo"; "bar"; "baz"];; - : string = "triplet"
# length [1; 2; 3; 4];; - : string = "many"
```

Pattern Matching with when and as

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

```
# let tall = function
      | (h, s) when h > 180 -> s ^ " is tall"
      | (_, s) -> s ^ " is short";;
val tall : int * string -> string = <fun>

# List.map tall [(183, "Stephen"); (150, "Nina")];;
- : string list = ["Stephen is tall"; "Nina is short"]
```

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

```
# match ((3,9), 4) with
      (_ as xx, 4) -> xx
      | _ -> (0,0);;
- : int * int = (3, 9)
```

Pattern Matching

n * (x + y)

```
# type expr =
      | Plus of expr * expr        (* means a + b *)
      | Minus of expr * expr       (* means a - b *)
      | Times of expr * expr       (* means a * b *)
      | Divide of expr * expr      (* means a / b *)
      | Value of string            (* "x", "y", "n", etc. *);;
```

```
# Times (Value "n", Plus (Value "x", Value "y"));;
- : expr = Times (Value "n", Plus (Value "x", Value "y"))
```

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 : expr -> string = <fun>

# let print_expr e = print_endline (to_string e);;
val print_expr : expr -> unit = <fun>
```

```
# print_expr (Times (Value "n", Plus (Value "x", Value "y")));;
(n * (x + y))
- : unit = ()
```

Pattern Matching

n * (x + y) ==> (n*x + n*y)

```
# let rec multiply_out e =
   match e with
  | Times (e1, Plus (e2, e3)) ->
        Plus (Times (multiply_out e1, multiply_out e2),
              Times (multiply_out e1, multiply_out e3))
  | Times (Plus (e1, e2), e3) ->
        Plus (Times (multiply_out e1, multiply_out e3),
             Times (multiply_out e2, multiply_out e3))
  | Plus (left, right) -> Plus (multiply_out left, multiply_out right)
  | Minus (left, right) -> Minus (multiply_out left, multiply_out right)
  | Times (left, right) -> Times (multiply_out left, multiply_out right)
  | Divide (left, right) -> Divide (multiply_out left, multiply_out right)
  | Value v -> Value v;;
val multiply_out : expr -> expr = <fun>
```

SOME DATA STRUCTURES

Map

- Creates a "mapping".
- **module** MyUsers = Map.Make(String);; (* create a map MyUsers *)
- # **let** m = MyUsers.empty;;
  -: val m : 'a MyUsers.t = <abstr> (*create an empty map*)
- # **let** m = MyUsers.add "fred" "sugarplums" m;;
  -: val m : string MyUsers.t = <abstr> (* add something to it *)
  - Once we have added the string "sugarplums" we have fixed the types of mappings that we can do

Map

```
(*Adding elements to the map*)
# let m = MyUsers.add "fred" "sugarplums" m;;
val m : string MyUsers.t = <abstr>
# let m = MyUsers.add "tom" "ilovelucy" m;;
val m : string MyUsers.t = <abstr>
# let m = MyUsers.add "mark" "ocamlrules" m;;
val m : string MyUsers.t = <abstr>
# let m = MyUsers.add "pete" "linux" m;;
val m : string MyUsers.t = <abstr>

(*Functions printing string *)
# let print_users key password = print_string(key ^ " " ^ password ^ "\n");;
val print_users : string -> string -> unit = <fun>

(*Map iteration*)
# MyUsers.iter print_users m;;

(*Find an element*)
# MyUsers.find "fred" m;;
```

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.

List: Some useful List Function

```
# List.map (fun a -> a + 10) [42; 17; 128];;
- : int list = [52; 27; 138]

# List.map string_of_int [42; 17; 128];;
- : string list = ["42"; "17"; "128"]

# List.fold_left (fun s e -> s + e) 0 [42; 17; 128];; List.fold_left f a [b1; ...;bn] = f (...(f (f a b1) b2)...) bn
- : int = 187

# List.iter print_int [42; 17; 128];;
4217128- : unit = ()

# List.iter (fun n -> print_int n; print_newline ()) [42; 17; 128];;
42 17 128 - : unit = ()

# List.iter print_endline (List.map string_of_int [42; 17; 128]);;
42 17 128 - : unit = ()
```

Arrays

```
# 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 = ()
# a;;
- : int array = [|42; 17; 20|]
```

Arrays

```
# let l = [24; 32; 17];;                (* Array from a list *)
val l : int list = [24; 32; 17]
# let b = Array.of_list l;;
val b : int array = [|24; 32; 17|]

# let c = Array.append a b;;
val c : int array = [|42; 17; 20; 24; 32; 17|]
```

Array vs List

	Arrays	Lists
Random Access	O(1)	O(n)
Appending	O(n)	O(1)
Mutable	Yes	No

Useful pattern: first collect data of unknown length in a list then convert it to an array with *Array.of_list* for random queries.

Hash Tables

```
# let hash = StringHash.create 17;;               (* initial size estimate *)
val hash : '_a StringHash.t = <abstr>

# StringHash.add hash "Douglas" 42;;              (* modify the hash table *)
- : unit = ()

# StringHash.mem hash "foo";;                     (* is "foo" there? *)
- : bool = false

# StringHash.mem hash "Douglas";;                 (* is "Douglas" there? *)
- : bool = true
# StringHash.find hash "Douglas";;                (* Get value *)
- : int = 42

# StringHash.add hash "Adams" 17;;  -: unit = ()  (* Add another key/value *)
# StringHash.find hash "Adams";;    - : int = 17
# StringHash.find hash "Douglas";;  -: int = 42
# StringHash.find hash "Slarti";;  Exception: Not_found.
```

FUNCTIONAL FEATURE

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

Functions as Argument

```
#let sqr x = x * x;;
- : val sqr : int -> int = <fun>
#sqr 5 (* calling *)
```

```
# let sqr1 = fun x-> x*x
- : val sqr1 : int -> int = <fun>
#sqr1 5 / (fun x->x*x) 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.
- Then map calls twice on each element of the given list ([1; 2; 3]) to produce the result: a list with each number doubled.
- map is known as a **higher-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 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

SOME EXAMPLES

Application: Length of a list

```
let rec length l =
  if l = [] then 0 else 1 + length (List.tl l);;
```

Correct but not very elegant. With pattern matching

```
let rec length = function
 [] ->0
| _::tl -> 1 + length tl;;
```

Elegant, but inefficient because it is not tail-recursive (needs $O(n)$ stack space).
Common trick: use an argument as an accumulator.

```
let length l =
let rec helper len = function
[] -> len
| _::tl -> helper (len + 1) tl
in helper 0 l
```

This is the code for the List.length standard library function.

Application: Length of a list

Ocaml Source Code

```
let length l =
let rec helper len = function
[] -> len
| _::tl -> helper (len + 1) tl
in helper 0 l
```

ocamlopt generates this x86 assembly

```
camlLength__helper:
 .L101:
cmpl $1, %ebx      # empty?
je .L100
movl 4(%ebx), %ebx # get tail
addl $2, %eax      # len++
jmp .L101
.L100:
ret

camlLength__length:
movl %eax, %ebx
movl $camlLength__2, %eax
movl $1, %eax # len = 0
jmp camlLength__helper
```

- Arguments in registers
- Pattern matching reduced to a conditional branch
- Tail recursion implemented with jumps