Fall 2018

Introduction to OCaml

Yu Zhang

Course web site: http://staff.ustc.edu.cn/~yuzhang/tpl

References

- Learn X in Y Minutes – Ocaml
- Real World OCaml

- Cornell CS 3110 Spring 2018
  Data Structures and Functional Programming
  - Introduction Functions Lists
  - Data types More Variants
  - Higher-order programming

- Official website http://ocaml.org/
- A web-based interpreter: http://try.ocamlpro.com/

What is a functional language?

- A functional language:
  - defines computations as mathematical functions
  - avoids mutable state

State: the information maintained by a computation

Mutable: can be changed (antonym: immutable)

	Functional	**Imperative**
Abstraction level	Higher	Lower
Develop robust SW	Easier	Harder
State	Immutable	Mutable
Expression	What to compute	How to compute

Why study functional programming?

- Functional languages predict the future
  - Garbage collection

    Java [1995], LISP [1958]
  - Generics

    Java 5 [2004], ML [1990]
  - Higher-order functions

    C#3.0 [2007], Java 8 [2014], LISP [1958]
  - Type inference

    C++11 [2011], Java 7 [2011] and 8, ML [1990]
  - What’s next?

Why study functional programming?

- Functional languages are *sometimes* used in industry
  - Java 8 — ORACLE
  - F#, C# 3.0, LINQ — Microsoft, jet
  - Scala — twitter, foursquare, LinkedIn
  - Haskell — facebook, BARCLAYS, at&t
  - Erlang — facebook, amazon, T-Mobile
  - OCaml — facebook, Bloomberg, CITRIX

  https://ocaml.org/learn/companies.html — Jane Street

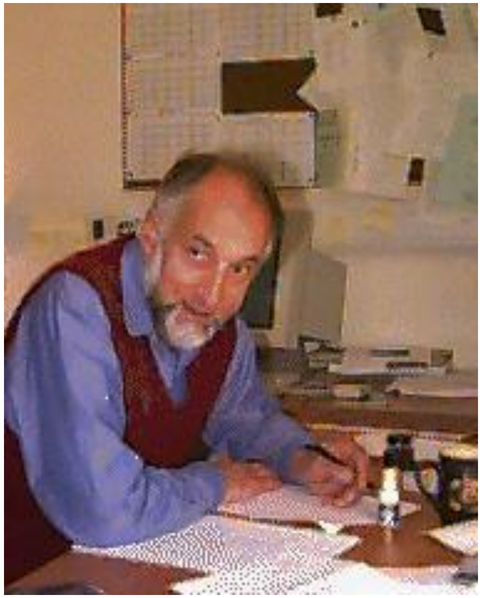
Why study functional programming?

- Functional languages are elegant
  - Elegant code is easier to read and maintain
  - Elegant code might (not) be easier to write

ML programming language

- Statically typed, general-purpose programming language
  - “Meta-Language” of the LCF theorem proving system

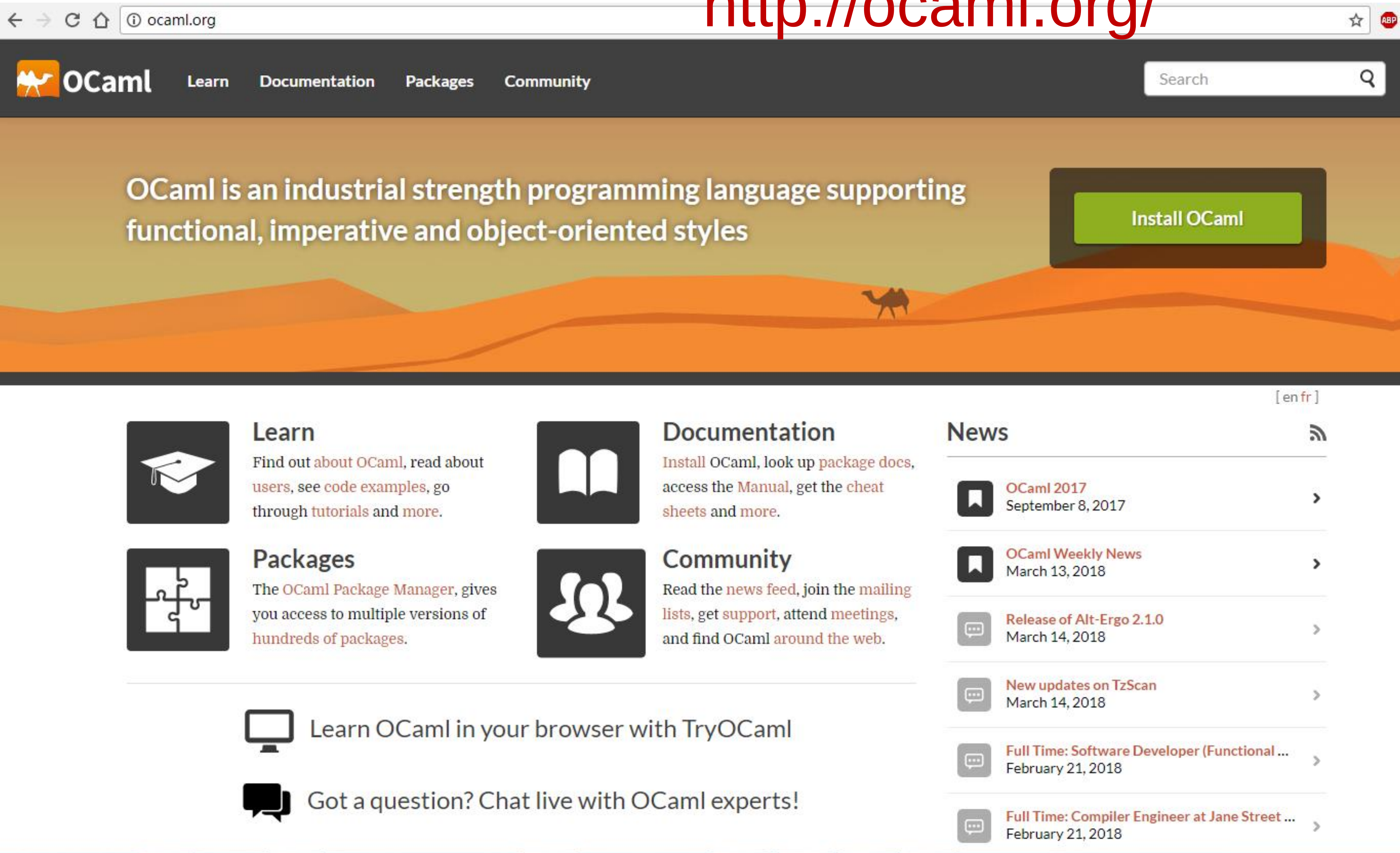
    LCF: Logic for Computable Functions
- Type safe, with formal semantics
- Compiled language, but intended for interactive use
- Combination of Lisp and Algol-like features
  - Expression-oriented
  - Higher-order functions
  - Garbage collection
  - Abstract data types
  - Module system
  - Exceptions



Robin Milner, ACM Turing-Award for ML, LCF Theorem Prover, …

OCaml (Objective Caml)

http://ocaml.org/



OCaml

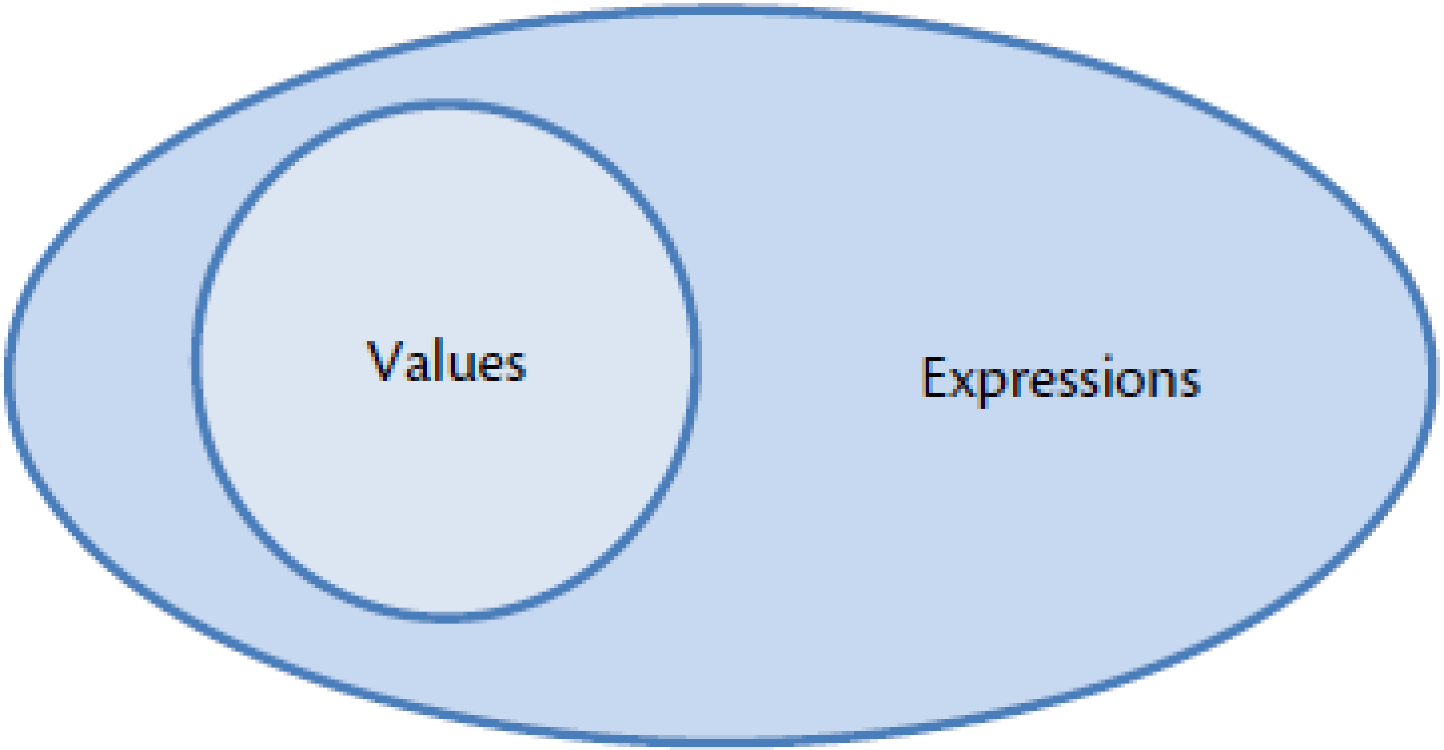
- Immutable programming
  Variable’s values cannot destructively be changed; makes reasoning about program easier!
- Algebraic datatypes and pattern matching
  Makes definition and manipulation of complex data structures easy to express
- First-class functions
  Functions can be passed around like ordinary values
- Static type-checking
  Reduce number of run-time errors
- Automatic type inference
  No burden to write down types of every single variable
- Parametric polymorphism
  Enables construction of abstractions that work across many data types
- Garbage collection
  Automated memory management eliminates many run-time errors
- Modules
  Advanced system for structuring large systems

Expressions (terms)

- Expressions:
  - Primary building block of OCaml programs
  - akin to *statements* or *commands* in imperative languages
  - can get arbitrarily large since any expression can contain subexpressions, etc.
- Every kind of expression has:
  - **Syntax**
  - **Semantics**:
    - Type-checking rules (static semantics): produce a type or fail with an error message
    - Evaluation rules (dynamic semantics): produce a value
      - (or exception or infinite loop)
      - Used only on expressions that type-check

Values

- A value is an expression that does not need any further evaluation
  - **34** is a value of type **int**
  - **34+17** is an expression of type **int** but is not a value



if expressions

- Syntax

  if e1 then e2 else e3

- Evaluation
  - if **e1** evaluates to **true**, and if **e2** evaluates to **v**, then **if e1 then e2 else e3** evaluates to **v**
  - if **e1** evaluates to **false**, and if **e3** evaluates to **v**, then **if e1 then e2 else e3** evaluates to **v**

- Type checking
  - if **e1: bool** and **e2:t** and **e3:t** then **if e1 then e2 else e3 : t**

Question

To what value does this expression evaluate?

if 22=0 **then** 1 **else** 2

if 22=0 **then** "bear" **else** 2

Question

To what value does this expression evaluate?

if 22=0 **then** 1 **else** 2

2

if 22=0 **then** "bear" **else** 2

Question

To what value does this expression evaluate?

if 22=0 **then** 1 **else** 2

2

if 22=0 **then** "bear" **else** 2

Does not type check!!!

So never gets a chance to be evaluated.

Function definitions

- Examples

rec is required because the body includes a recursive function call

```
let rec pow (x : int) (y : int) : int =
    if y=0 then 1
    else x * pow x (y-1)

let rec pow x y =
    if y=0 then 1
    else x * pow x (y-1)

let cube x = pow x 3

let cube (x : int) : int = pow x 3
```

no types written down!
compiler does *type inference*

Function definitions

- Syntax

  let rec f x1 x2 ... xn = e

  note: **rec** can be omitted if function is not recursive

- Evaluation

  Not an expression! Just defining the function; will be evaluated later, when applied

- Function Types
  - Type **t -> u** is the type of a function that takes input of type **t** and returns output of type **u**
  - Type **t1 -> t2 -> u** is the type of a function that takes input of type **t1** and another input of type **t2** and returns output of type **u**

Function definitions

- Syntax

  let rec f x1 x2 ... xn = e

  note: **rec** can be omitted if function is not recursive

- Evaluation

  Not an expression! Just defining the function; will be evaluated later, when applied

- Type-checking

  Conclude that **f : t1 -> ... -> tn -> u** if **e:u** under these assumptions:

  - **x1:t1, ..., xn:tn** (arguments with their types)
  - **f: t1 -> ... -> tn -> u** (for recursion)

Function application

- Syntax

  f e1 e2 ... en

- Evaluation

  1. Evaluate arguments **e1...en** to values **v1...vn**
  2. Find the definition of f: let f x1 ... xn = e
  3. Substitute **vi** for **xi** in **e** yielding new expression **e’**
  4. Evaluate **e’** to a value **v**, which is result

- Type-checking

  if **f : t1 -> ... -> tn -> u** and **e1:t1, ..., en:tn**

  then **f e1 ... En:u**

Anonymous functions

- **Examples**
  - **fun x -> x+1** is an anonymous function
  - and we can bind it to a name:

    let inc = fun x -> x+1
  - Note

    dual purpose for -> syntax: function types, function values

    fun is a keyword :)

Anonymous functions

- Syntax

  fun x1 x2 ... xn -> e

- Evaluation

  A function is a value

- Type-checking

  (fun x1 ... xn -> e) : t1->...->tn->t

  if **e:t** under assumptions **x1:t1, ..., xn:tn**

Lists

```
let lst = [1;2;3]
let empty = []

let longer = 5::lst
let another = 5::1::2::3::[]

let rec sum xs =
        match xs with
        | [] -> 0
        | h::t -> h + sum t
let six = sum lst
let zero = sum empty
```

Lists

```
let lst = [1;2;3]
let empty = []

let longer = 5::lst
let another = 5::1::2::3::[]

let rec sum xs =
    match xs with
    | [] -> 0
    | h::t -> h + sum t
let six = sum lst
let zero = sum empty
```

[1;2;3]: int **list**

[]: t **list** for any type t

If **e1 : t** and **e2 : t list** then **e1::e2 : t list**

Variants vs. records vs. tuples

	Define	Build	Access
Variant	`type`	Constructor name	Pattern matching
Record	`type`	Record expression with {…}	Pattern matching OR field selection with dot operator `.`
Tuple	N/A	Tuple expression with (…)	Pattern matching OR `fst` or `snd`

- **Variants**: one-of types aka **sum types**
  - **type t = C1 | ... | Cn**
- **Records, tuples**: each-of types aka **product types**
  - **type t = {f1:t1; ...; fn:tn}**

    {f1=p1; ...; fn=pn}

    e.f

(e1,e2,...,en)

Lists are just variants

- OCaml effectively codes up lists as variants

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

  - **list** is a type constructor parameterized on type variable **'a**
  - **[ ]** and **::** are constructors
  - Just a bit of syntactic magic in the compiler to use **[ ]** and **::** instead of alphabetic identifiers

Options are just variants

- OCaml effectively codes up options as variants

```
type 'a option = None | Some of 'a
```

  - **option** is a type constructor parameterized on type variable **'a**
  - **None** and **Some** are constructors

Exceptions are (mostly) just variants

- OCaml effectively codes up exceptions as slightly strange variants

  type exn
  exception MyNewException **of** string

  - Type **exn** is an *extensible* variant that may have new constructors added after its original definition
  - Raise exceptions with **raise e**, where **e** is a value of type **exn**
  - Handle exceptions with pattern matching, just like you would process any variant

Higher-order functions

- **let** double x = 2*x
  let square x = x*x

  let quad x = double (double x)
  let fourth x = square (square x)

- **let** twice f x = f (f x)
  val twice : ('a -> 'a) -> 'a -> 'a
- **let** quad x = twice double x
  let fourth x = twice square x

Currying

- Currying:
  - A function with multiple parameters is sugar for a function with a tuple parameter
  - Curried form: high-order function

let plus (x, y) = x + y

is sugar for

let plus (z : int * int) = **match** z **with** (x, y) -> x + y

let plus = **fun** (z : int * int) -> **match** z **with** (x, y) -> x +

curried form

let plus x y = x + y

Currying

- Currying:
  - A function with multiple parameters is sugar for a function with a tuple parameter
  - Curried form: high-order function

let plus (x, y) = x + y

is sugar for

plus: int * int -> int

let plus (z : int * int) = **match** z **with** (x, y) -> x + y

let plus = **fun** (z : int * int) -> **match** z **with** (x, y) -> x + y

curried form

let plus x y = x + y

plus: int -> int -> int

Map and reduce

- Fold has many synonyms/cousins in various functional languages, including **scan** and **reduce**
- Google organizes large-scale data-parallel computations with MapReduce [OSDI 2004 Jeff Dean et al.]

“[Google’s MapReduce] abstraction is inspired by the map and reduce primitives present in Lisp and many other functional languages. We realized that most of our computations involved applying a map operation to each logical record in our input in order to compute a set of intermediate key/value pairs, and then applying a reduce operation to all the values that shared the same key in order to combine the derived data appropriately." [Dean and Ghemawat, 2008]

Map

```
let rec add1 = function
        | [ ] -> [ ]
        | h::t -> (h+1)::(add1 t)

let rec concat3110 = function
        | [ ] -> [ ]
        | h::t -> (h^"3110")::(concat3110 t)

let rec map f = function
        | [ ] -> []
        | x::xs -> (f x)::(map f xs)
map : ('a -> 'b) -> 'a list -> 'b list
```

Map

```
let rec add1 = function
        | [ ] -> [ ]
        | h::t -> (h+1)::(add1 t)
```

```
let add1 =
   List.map (fun x -> x+1)
```

```
let rec concat3110 = function
        | [ ] -> [ ]
        | h::t -> (h^"3110")::(concat3110 t)
```

```
let concat3110 =
   List.map (fun s -> s^"3110")
```

```
let rec map f = function
        | [ ] -> []
        | x::xs -> (f x)::(map f xs)
map : ('a -> 'b) -> 'a list -> 'b list
```

Filter

```
let rec filter f = function
        | [] -> []
        | x::xs -> if f x
                then x::(filter f xs)
                else filter f xs
```

filter : ('a -> bool) -> 'a list -> 'a list