

# CISS445 Lecture 11: OCAML Part 1

Yihsiang Liow

January 6, 2020

# Table of contents I

- 1 Resources
- 2 Functional languages
- 3 OCAML
- 4 Miscellaneous

# Resources

- PLP (Scott) Chapter 10
- OCAML
  - Homepage: <http://caml.inria.fr/>
  - Software: <http://caml.inria.fr/download.en.html>
  - User manual: <http://caml.inria.fr/pub/docs/manual-ocaml/index.html>

# Imperative languages

- **Imperative languages**
  - Computational model: A machine state consists of values of stored in memory. Each statement will modify a machine state to produce another machine state. We say that this computational model has side effects machine state before and after a statement can be different.
  - Follows the **von Neumann model of computation**
  - Examples: Fortran, C, C++, Pascal, Java, Python, etc.

# Functional languages

- **Functional languages**

- Also known as applicative languages
- Computational model: Programs are made up of functions that take arguments and return values; arguments and returned values may be functions.
- A pure functional language does not modify the state of the computational model no side effects.
- Follows the computational model of lambda calculus developed by Alonzo Church (1930s)
- Examples: LISP, ML, Haskell

# History I

- The imperative and functional models grew out of research in theory of computations.
- Earlier on there were many different mathematical models of computation.
- The imperative model comes from Alan Turing's model of computation where "memory" means an infinite tape of cells. The machine (automaton) has a read-write head. Based on a "program" the read-write head moves left or right and write symbols onto the tape. At some point, the "program" might cause the machine to halt. It's also possible for the machine to never halt.

## History II

- Later it was proven that all the earlier computational models are equivalent in power to Turing's model.
- Turing's model (based on "Turing machines") gave rise to von Neumann architecture of computers and imperative languages.
- Alonzo Church's model (based on "lambda calculus") gave rise to functional languages.
- Later, if there's time (probably not!), there will be a short intro to Turing machine. Details: CISS362 Automata.

## History III

- LISP
  - LISP = LISt Processor
  - One of the oldest and most widely used functional languages.
  - Developed by John McCarthy (late 1950s, MIT).
  - Motivation was study of AI.
  - There are many derivatives of LISP: Scheme, Common LISP, ELISP, etc.
  - Scheme is more academic and is frequently used as a teaching tool.
  - Common LISP is an industrial strength programming language, an ANSI standardization of LISP.



## History IV

- ML
  - **ML** = **M**eta-**L**anguage
  - Robin Milner (late 1970s, University of Edinburgh)
  - Developed ML for LCF (Logic for Computable Functions) which is an automated theorem proving system
  - Winner of 1991 ACM Turing Award
  - Derivatives:
    - SML - Standard ML (USA, UK)
    - CAML - Categorical Abstract Machine Lang (Europe)
    - OCAML
    - F# - Microsoft's implementation of OCAML

# History V

- OCAML
  - “O” = object-oriented.
  - Full general purpose programming language.
  - Automatic theorem provers, compilers and interpreters, program analyzers, used as teaching tool, file synchronizer, multi-platform multi-networks P2P client, modeling language for finance, etc.
  - Open source, multi-platform.
  - F# is Microsoft's implementation of OCAML
  - Prominent in ICFP Contest [https://en.wikipedia.org/wiki/ICFP\\_Programming\\_Contest](https://en.wikipedia.org/wiki/ICFP_Programming_Contest)

# History VI

- Some companies using OCAML or F#:

Facebook	Walmart	McKinsey
Deutsche Bank	Oracle	IBM
Northrop Grumman	Docker	Github
Jane Street	TiVo	Microsoft
Adobe	Goldman Sachs	Sabre
Bloomberg	Amazon	

# History VII

## Getting started: windows I

- Windows (I will be using linux): google, install, run. Test by entering "hello, world!";; into the OCAML shell



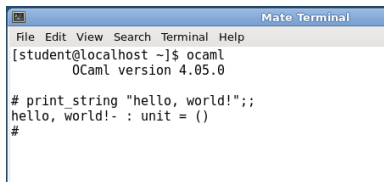
The screenshot shows a window titled "OCamlWinPlus v1.9RC4 - [Session transcript]". The window has a menu bar with "File", "Edit", "Workspace", "Window", and "Help". The main text area displays the following text:

```
Objective Caml version 3.08.3

# "hello, world!";;
- : string = "hello, world!"
# |
```

## Getting started: linux I

- Linux (also available on Cygwin, Mac, ...): I'm using the Fedora 29 virtual machine. In bash shell, as root, do `dnf -y install ocaml`. To test: run `ocaml` in your shell. Ctrl-D to end. Ctrl-C to stop an input.



```
Mate Terminal
File Edit View Search Terminal Help
[student@localhost ~]$ ocaml
OCaml version 4.05.0

# print_string "hello, world!";;
hello, world!- : unit = ()
#
```

## Getting started: linux II

- There's a better OCAML shell called utop. In bash shell, as root, do `dnf -y install utop`.

# Getting started: source programs I

- Besides writing OCAML code interactively into your OCAML shell, you can save your code into a file.

- Save this in file `helloworld.ml`:

```
print_string "hello world\n";;
```

- Now execute this in your terminal (bash) shell:

```
ocaml helloworld.ml
```

- Use double semicolon at the end of an expression while running in interactive mode (in the OCAML shell). You can use single semicolon in a stored program.



## Getting started: source programs II

- Also, note that in interactive mode, OCAML will print the value and type of every expression. But when you execute an OCAML program at the prompt, you get an output when you execute a print (or when there's an error in your program.)

## Basic types: strings I

- Type `"hello, world!";;` at the OCAML prompt. What does it say?

# Basic types: integers I

- In your OCAML shell type `1;;` and hit enter. What do you see?
- Now try to get OCAML to do the following
  - Sum of 2 and 3
  - Difference of 2 and 3
  - Quotient of 2 by 3
  - Remainder when 2 is divided by 3

# Basic types: booleans I

- In your OCAML shell type `true;;` and hit enter. What do you see?
- Try these:
  - `3 = 5;;`
  - `3 < 5;;`
  - `3 >= 7;;`
  - `3 <> 5;;`
  - `3 < 5 && 5 < 7;;` (\* & is deprecated \*)
  - `3 < 5 || 5 < 1;;` (\* or is deprecated \*)
  - `not (3 < 5);;`

## Basic types: floats I

- Try to do the following:
  - Sum of 1.2 and 3.4
  - Difference of 1.2 and 3.4
  - Product of 1.2 and 3.4
  - Quotient of 1.2 and 3.4
- Exercise. Is there an exponentiation operator for floats?
- Try this:

`1 + 2.2;;`

No automatic type conversion/coersion.

## Basic types: floats II

- What if you do `1.0 +. 2;;`?
- Type conversion: Try these:
  - `float_of_int 2;;`
  - `int_of_float 2.3;;`
  - `string_of_int 2;;`
  - `string_of_float 2.3;;`
- How do you get the interpreter to give you integer 42 from the string "42"? Float 3.14 from the string "3.14"?

# Basic types: characters and strings I

- Try this:  
`'a';;`
- And this:  
`"hello world";`
- Character from string:  
`"hello world".[0];;`
- Exercise. What is the string concatenation operator in OCAML?

# Print functions I

- Try `print_int 1;;` at the OCAML prompt.
- You see

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

  - The 1 is due to the the `print_int`.
  - The `-: unit = ()` is due to the return value and type of `print_int`. The `unit` is the same as `void` in C/C++.
- What is the print function for strings? Redo your hello world using this print function.



# Expressions I

- Recall from C/C++ that an expression is something that will evaluate to a value. Also has statements.
- An OCAML program is made up of only expressions (and declarations).
- Every expression must have a value of some type. OCAML is strongly typed.
- The `unit` type has only one value: `()`.
- The `int` type has many values: `0`, `2`, `-5`, ...

# Expressions II

- Expressions have types.
- So far the primitive types are:
  - `int`: -1, 2, 3
  - `bool`: `true`, `false`
  - `float`: 3.1415
  - `char`: `'a'`
  - `string` `"hello world"`
  - `unit`: `()`

# Declaration I

- Try `let x = 1;;` at the OCAML prompt. What do you see?
- Now try `let x = 1.2;;` at the OCAML prompt. What do you see?
- Notice that variable `x` does not have a type. `x` is a name for a value. The type is associated to the value. At this point, we say “`x` is bound to `1.2`”.

# Environment I

- An **environment** is just a list of name, value bindings, i.e. a map of names to values. It's usually written in the following way:

$$\rho = \{\text{name}_1 \rightarrow \text{value}_1, \text{name}_2 \rightarrow \text{value}_2, \dots\}$$

- This is usually implemented as list or stack.
- New bindings are inserted on the left.
- To search for a name, start from the left and scan right.
- A declaration adds binding to environment.

## Environment II

- Example:

<code>let x = 0;;</code>	$\rho_1 = \{x \rightarrow 0\}$
<code>let y = x + 1;;</code>	$\rho_2 = \{y \rightarrow 1, x \rightarrow 0\}$
<code>let x = 3.1;;</code>	$\rho_3 = \{x \rightarrow 3.1, y \rightarrow 1, x \rightarrow 0\}$

Note new binding of `x` after last `let` declaraton.

- In the above,  $y \rightarrow 1$  is a binding or a value binding for `y`.
- Very important concept for interpreters.

## Environment III

- Try this:

```
let x = 0 let y = 1;;
```

the same as:

```
let x = 0;;
```

```
let y = 1;;
```

- Try this:

```
let x = 0 let y = x + 1;;
```

- Exercise: What about “let A = 1;;”?

## Environment IV

- Try these:

```
let x = 0 and y = 1;;
```

(Compare with `let x = 0 let y = 1;;`)

- Try this:

```
let x = 0 and y = x + 1;;
```

Problem ... compare with

```
let x = 0 let y = x + 1;;
```

# Local binding I

- Compare this (you have already seen this):

```
let a = 2;;
```

with

```
let a = 2 in a*a;;
```

- Look at the response from OCAML.
- Now try this:

```
let b = let a = 41 in a + 1;;
```



## Local binding II

- Finally try this:

```
let a = 0;;  
let b = let a = 41 in a + 1;;  
print_int a;;
```

## Local binding III

- Common style:

```
# let x =  
    let a = 1 in  
    let b = 2 in  
    let c = 3 in  
    a + b + c;;  
val x : int = 6
```

## Local binding IV

- Compare above with

```
# let x =  
    let a = 1 and  
    let b = 2 and  
    let c = 3 in  
    a + b + c;;  
val x : int = 6
```

## Local binding V

- The environment

```
let a = 0;;
```

$$\rho_1 = \{a \rightarrow 0\}$$

```
let b = 1;;
```

$$\rho_2 = \{b \rightarrow 1, a \rightarrow 0\}$$

```
let x =
```

```
    let a = 2 in
```

$$\rho_3 = \{a \rightarrow 2, b \rightarrow 1, a \rightarrow 0\}$$

```
    a + b;;
```

$$\rho_4 = \{x \rightarrow 3, b \rightarrow 1, a \rightarrow 0\}$$

The environments shown are AFTER the expressions are evaluated.

# if-else I

- Of course we must have the selection structure
- Try these:

```
if true then 2 else 3;;  
if false then 2 else 3;;  
if true then 2;;  
if true then 2 else 0.0;;
```

## if-else II

- Exercise. What is the type of `z` and its value, or is there an error?

```
let x = 0;;  
let y = 1;;  
let z = if x < y then x;;
```

- Exercise. What is the type of `z` and its value, or is there an error?

```
let x = 0;;  
let y = 1;;  
let z = x + (if x < y then x else y);;
```

## if-else III

- Exercise. What is the type of `z` and its value, or is there an error?

```
let x = 1;;  
let y = 1.1;;  
let z = if x < 0 then x else y;;
```

- Exercise. What is the type of `z` and its value, or is there an error?

```
let x = 0.0;;  
let y = 1.1;;  
let z = (if x < y then x else x + 1);;
```

# Functions I

- Try this:

```
# fun x -> x + 1;;  
- : int -> int = <fun>
```

- The expression is a function.
- Furthermore this is a function from an integer to another integer. How does OCAML figure out the type of the parameter and the return type? (Type inference)
- The type of this value is `int -> int`.
- The above function is nameless - anonymous



## Functions II

- Try this:

```
# (fun x -> x + 1) 41;;  
- : int = 42
```

- Try this:

```
# let inc = fun x -> x+1;;  
val inc : int -> int = <fun>  
# inc 41;;  
- : int = 42
```

## Functions III

- The following

```
let inc x = x + 1;;
```

is syntactic sugar for

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

- Instead of

```
let inc = (fun x -> x + 1);;
```

you can do this:

```
let inc = (function x -> x + 1);;
```

## Functions IV

- This is a function:

```
fun x -> x + 1;;
```

When you "call a function":

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

we say that this expression is a function application.

- We say that we're applying the function to 3.
- Try this:

```
let inc x = x + 1;;  
print_int (inc "a");;
```

Read the error message carefully and do not ignore them.

# Functions V

- Exercise. Write an anonymous function that decrements an integer by 1. Apply this function to 43.
- Exercise. Write an anonymous function that decrements a float by 1. Apply this function to 43.0.
- Exercise. Write a function `iseven` that returns true if the value of the argument applied to the function is even; otherwise the function returns false. Test your function.
- Exercise. Write a function `iseven` that returns true if the value of the argument applied to the function is even; otherwise the function returns false. Test your function.

## Functions VI

- Exercise. Write a function called `askdeepblue` that if the function is called with an integer  $< 42$  it prints "too low", if it is called with an integer  $> 42$  it prints "too high", and it is called with 42, it print "that's the answer".

## Functions VII

- What is the value of the last expression?

```
let x = 1;;  
let f = fun y -> y + x;;  
f 5;;  
let x = 2;;  
f 5;;
```

## Functions: more than one param I

- Try this:

```
# let diff x y = x - y;;  
val diff : int -> int -> int = <fun>
```

- Think of `int -> int -> int` as `int -> (int -> int)`.

- Try this:

```
diff 45 3;; (* no surprises *)
```

- Now this:

```
let f = diff 45;; (* f is int->int function! *)  
f 3;;
```

## Functions: more than one param II

- Here's diff again:

```
let diff x y = x - y;;
```

- Removing syntactic sugar this is actually:

```
let diff = fun x -> (fun y -> x - y);;
```

- So

```
let f = diff 45;;
```

is the same as

```
let f = fun y -> 45 - y;;
```



## Functions: more than one param III

- Another example:

```
# let sum x y = x + y;;  
val sum : int -> int -> int = <fun>  
# let inc = sum 1;;  
val inc : int -> int = <fun>  
# let inctwice = sum 2;;  
val inctwice : int -> int = <fun>  
# inc 41;;  
- : int = 42  
# inctwice 40;;  
- : int = 42
```

## Functions: more than one param IV

- Exercise: What is the type of  
`(fun x -> (fun y -> 2 * x + y));;`
- Exercise: What is the type of  
`(fun x -> (fun y -> 2 * x + y)) 3;;`
- Exercise: What is the type of  
`(fun x -> (fun y -> 2 * x + y)) 3 5;;`

## Functions: more than one param V

- Exercise: What is the type of  
`fun x -> fun y -> if x + 1 < y then x else y;;`
- Exercise: What is the type of  
`(fun x -> fun y -> if x + 1 < y then x else y) 3;;`
- Exercise: What is the type of  
`(fun x -> fun y -> if x + 1 < y then x else y) 1 1;;`

## Functions: more than one param VI

- Exercise: Write a function `clip` so that `(clip m M x)` will evaluate to `m` if `x` is less than or equal to `m`, to `M` if `x` is greater than or equal to `M`, otherwise it evaluates to `x`. Now do

```
let f = clip 0;;  
let g = f 1;;  
let h = g (-1);;
```

What is the type of `g`? What about `h`?

# Functions: untyped arguments I

- Try this:

```
# let lessthan x y = x < y;;  
val lessthan : 'a -> 'a -> bool = <fun>  
# lessthan 1 2;;  
- : bool = true  
# lessthan 1.1 2.2;;  
- : bool = true
```

- Exercise: Rewrite `lessthan` without syntactic sugar.

## Functions: untyped arguments II

- Here's `lessthan` again:

```
# let lessthan x y = x < y;;  
val lessthan : 'a -> 'a -> bool = <fun>
```

The `<` operator is polymorphic, i.e., it can accept two int, two floats, two chars, ...

- The `'a` is a **type variable**.
- Exercise. What is the type of

```
let f = lessthan 1;;
```

## Functions: untyped arguments III

- Try this:

```
let f x y = y;;
```

- Exercise. Rewrite `f` without syntactic sugar.
- Look at the type of `f`.
- In general, OCAML will create type variables names in this order: 'a, 'b, 'c, 'd, ... (In formal OCAML papers/books, the type variables are  $\alpha$ ,  $\beta$ ,  $\gamma$ ,  $\delta$ , ...)

## Functions: untyped arguments IV

- However suppose you only want to allow arguments of certain type, you can do this:

```
# let lessthan (x:int) (y:int) = x < y;;  
val lessthan : int -> int -> bool = <fun>
```

- Also, try this (why?):

```
# let lessthan (x:int) y = x < y;;
```

- Exercise: Rewrite the above w/o syn sugar. Of course there's no point doing this (why?):

```
let f (x:int) y = x + y;;
```



## Functions: untyped arguments V

- Exercise: Let `h` be bound to the following:

```
(fun x -> (fun y ->  
  if x < y then fun x->x+1 else fun y->2*y))
```

- What is the type of `h`?
- What is the type of `(h 1 2)`?
- What is the type of `(h 2 1)`?
- What is the type and value of `(h 1 2 3)`?
- What is the type and value of `(h 2 1 3)`?

## Functions: function as argument I

- Recall that you can define a function to add 2 using `inc`:

```
let inc x = x + 1;;  
let inctwice x = inc(inc x);;
```

- Now let's do it this way:

```
# let twice f x = f(f x);;  
val twice : ('a -> 'a) -> 'a -> 'a = <fun>  
# let twiceinc = twice inc;;  
val twiceinc : int -> int = <fun>
```

- First note that `twice` accepts a function as a first parameter.

## Functions: function as argument II

- Second note that OCAML deduced the relationship between the type of `x` and the types of the parameter and return value of `f`.
- Note that the `twice` function allows you to apply any function twice.
- Exercise. Rewrite `twice` without syntactic sugar.

## Functions: function as argument III

- Exercise. Write a function, `twist`, that accepts a function `f` (of two arguments) and returns another function that behaves in the same way as `f` but the arguments are "twisted". In other words:

```
let tf = twist f;; (* (tf 3 2) if (f 2 3) *)  
let tg = twist g;; (* (tg 'b' 'a') is (g 'a' 'b') *)  
let th = twist h;; (* (th 2.3 1.1) is (h 1.1 2.3) *)
```

# xemacs I

- Optional. Google and install tuareg 2.0.4 for your xemacs:
  - Put the unzipped tuareg folder, tuareg-2.0.4, in ~/.xemacs/.
  - Add the following to ~/.xemacs/init.el:

```
(add-to-list 'load-path "~/.xemacs/tuareg-2.0.4")  
(add-to-list 'auto-mode-alist '("\\.ml[iylp]?" . tuareg-mode))  
(autoload 'tuareg-mode "tuareg"  
          "Major mode for editing Caml code" t)  
(autoload 'camldebug "camldebug" "Run the Caml debugger" tS)
```
- The default xemacs has some support for OCAML editing (the CAML mode). Tuareg adds more.

## xemacs II

- You can also execute a section of code:
  - Execute expression at cursor: C-c C-e
  - Execute selected code (region): C-c C-s or C-M-x.
- Google tuareg for more info.

## Printf.printf function

- Recall that you can print integers, floats, ...:

```
print_int 3;;  
print_string "hello world\n";;
```

- Try this:

```
Printf.printf "%s, %s, %s ... %d" "life"  
"universe" "everything" 42;;
```

- Exercise. Fill in the ?? in

```
Printf.printf "%??, %??, %??" true 3.14 'A';;
```

# Debug printing I

- Suppose you want to debug this function:

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

- Say you want to insert some print statement(s) in the function:

```
let f x =  
  let _ = print_string "entering f ... \n" in  
  x + 1;;
```

(Or you can check the ocaml debugger.)



## Debug printing II

- Or you can do this:

```
let f x =  
    (print_string "entering f ...\n"; x + 1);;
```

- (e1; e2; e3; e4) is a sequence of expressions. The value of this is the value of the last expression e4. All the others should have () value or you'll get a warning.

# Multifile I

- Suppose you want to use function `f` declared in `useful.ml`, you do this:

```
#use "useful.ml";;
```

- This reads and executes the OCAML code in `useful.ml`.
- Make sure you include the `#` in your command.
- Note that this will execute all the code in `useful.ml`.