

# **CSE216**

## **Foundations of Computer Science**

**Instructor: Zhoulai Fu**

**State University of New York, Korea**

Many slides taken from Prof. YoungMin Kwon. Thanks!

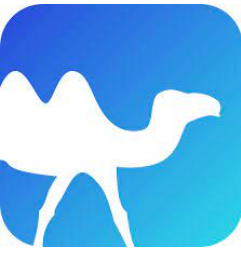
# Plan

- Today: functions in Ocaml
- Midterm 1 results will be announced by Thursday's recitation. Most did a good job!

**A missing explanation**

# 'a type

- # max ;;
- - : 'a -> 'a -> 'a = <fun>



# if c then e1 else e2

```
# if true then "hello" else 5;;
```

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

```
# if true then "hello";;
```

**Error:** This expression has type string but an expression was expected of type unit

because it is in the result of a conditional with no else branch

.. ■

# Ocaml Functions

# Function Definitions

(also known as Procedural Abstraction)

*let* <name> <formal parameters> = <body>

```
# let square x = x * x;;  
val square : int -> int = <fun>
```

name

formal  
parameter

body

# Function Application

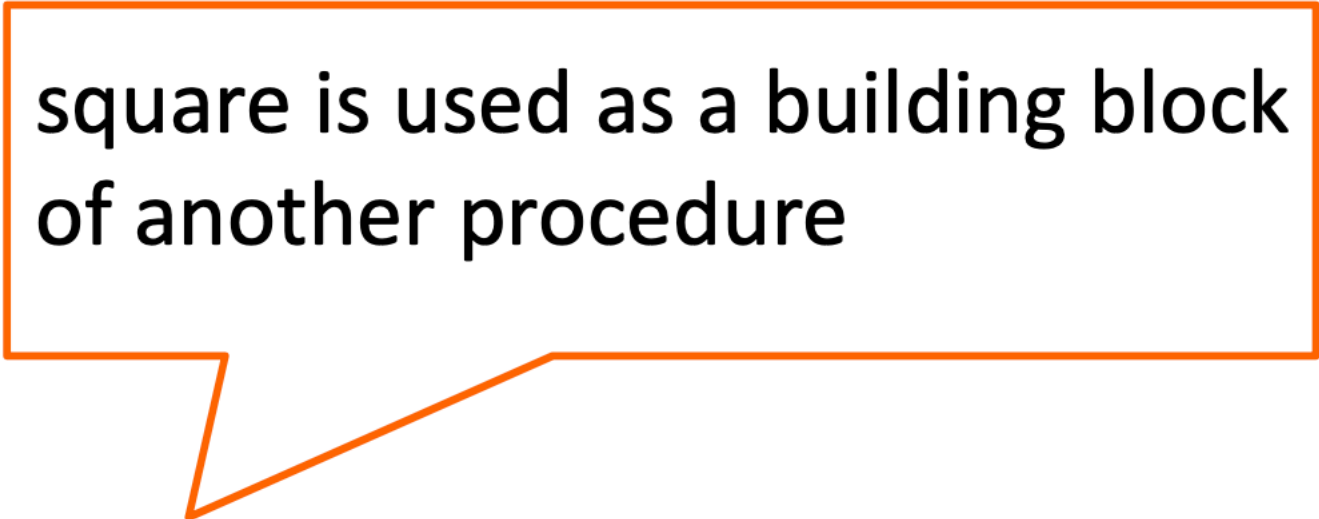
```
# square 3;;  
- : int = 9
```

```
# square (1+2);;  
- : int = 9
```

```
# square (square 3);;  
- : int = 81
```

```
# let sum_of_squares x y = square x + square y;;  
val sum_of_squares : int -> int -> int = <fun>
```

```
# sum_of_squares 3 4;;  
- : int = 25
```



square is used as a building block  
of another procedure



# Anonymous functions

*fun* <formal parameters> -> <body>

```
# fun x -> x * x;;  
- : int -> int = <fun>
```

anonymous  
function

formal  
parameter

body

```
# let square = fun x -> x * x;;  
val square : int -> int = <fun>
```

# Recursive function

*let **rec** <name> = fun <formal parameters> ->  
<body>*

# let **rec** **fact** = fun x -> if x = 0 then 1 else x \* **fact** (x  
- 1);;

val **fact** : int -> int = <fun>

# Multi-parameter functions

```
let add x y = x + y  
≡ let add = fun x -> fun y -> x + y  
≡ let add = fun x -> (fun y -> x + y)
```

```
add 2 3  
≡ (add 2) 3
```

# Tuple parameter function

```
# let add (a, b) = a + b;;  
val add : int * int -> int = <fun>  
# add (1, 2);;  
- : int = 3
```

```
# let add' a b = a + b;;  
val add' : int -> int -> int = <fun>  
# add' 1 2;;  
- : int = 3
```

```
# let inc = add' 1;;  
val inc : int -> int = <fun>  
# inc 2;;  
- : int = 3
```

# The `|>` operator

The `|>` operator represents reverse function application.

You can put the function *after* the value you want to apply it to. This allows building up something that looks like a Unix pipeline:

```
# let ( |> ) x f = f x;;  
val ( |> ) : 'a -> ('a -> 'b) -> 'b = <fun>  
# 0.0 |> sin |> exp;;  
- : float = 1.
```

'a is a *type variable*, and stands for any given type

# Lab exercise 1

- Define a function **square** that squares an integer.
- Find different ways to define the function
- Use the `|>` operator to square the number 123456789.

# Lab exercise 2

- Write a function `hypotenuse` of type `float->float->float`, that takes the lengths of the two shorter sides of a right triangle and returns the length of the hypotenuse.
- Write the function in 3+ different ways
- Calculate the hypotenuse for sides of lengths 3.1 and 4.2
-

# Boolean expression

```
# true;;
```

```
- : bool = true
```

```
# false;;
```

```
- _ : bool = false
```



# Structure Comparison

```
# (==);;  
- : 'a -> 'a -> bool = <fun>  
# (<);;  
- : 'a -> 'a -> bool = <fun>  
# (>);;  
- : 'a -> 'a -> bool = <fun>
```

```
# 2 > 1;;  
- : bool = true
```

```
# 2. > 1.;;  
- : bool = true
```

```
# 2 > 1.;;  
Characters 4-6:  
  2 > 1.;;  
    ^^
```

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

```
# int_of_float 1. > 2;;  
- : bool = false
```

```
# float_of_int 1 > 2.;;  
- : bool = false
```

```
# float 1 > 2.;;  
- : bool = false
```

```
# "abc" = "abc";;  
- : bool = true
```

```
# "abc" <> "abc";;  
- : bool = false
```

```
# "abc" < "def";;  
- : bool = true
```

# Structure/address Comparison

```
(* =, <>: comp. structures,  
    ==, !=: comp. addresses *)
```

```
# (==);;  
- : 'a -> 'a -> bool = <fun>  
# (!=);;  
- : 'a -> 'a -> bool = <fun>
```

```
# "hello" = "hello";;  
- : bool = true
```

```
# "hello" <> "hello";;  
- : bool = false
```

```
# "hello" == "hello";;  
- : bool = false
```

```
# "hello" != "hello";;  
- : bool = true
```

```
# let v = "hello";;  
val v : string = "hello"  
# v = v;;  
- : bool = true  
# v <> v;;  
- : bool = false  
# v == v;;  
- : bool = true  
# v != v;;  
- : bool = false
```

```
# let u = v;;  
val u : string = "hello"  
# u == v;;  
- : bool = true  
# u != v;;  
- : bool = false
```

# Logical connectives: &&, ||, not

```
# (&&);;
```

```
- : bool -> bool -> bool = <fun>
```

```
# let inside lb ub x = lb <= x && x <= ub;;
```

```
val inside : 'a -> 'a -> 'a -> bool = <fun>
```

```
# inside 0 10 5;;
```

```
- : bool = true
```

```
# let outside lb ub x = not (inside lb ub x);;
```

```
val outside : 'a -> 'a -> 'a -> bool = <fun>
```

```
# outside 0 10 5;;
```

```
- : bool = false
```

# Conditional Expression

*if* *<predicate>* *then* *<consequent>*  
*else* *<alternative>*

```
# let abs x = if x >= 0 then x else - x;;  
val abs : int -> int = <fun>
```

```
# abs (-3);;  
- : int = 3
```

# Conditional Expression (2)

- Example: factorial

```
# let rec factorial x =  
    if x = 0  
    then 1  
    else x * factorial (x - 1);;  
val factorial : int -> int = <fun>  
  
# factorial 4;;  
- : int = 24
```

- To define a *recursive function*, use **let rec**

# Exercises

# Exercises 1

- Write down the types of the defined functions in OCaml:
  - a) `let double x = 2*x;;`
  - b) `let square x = x*x;;`
  - c) `let twice f x = f (f x);;`
  - d) `let quad = twice double;;`
  - e) `let fourth = twice square;;`