# CSE216
# Foundations of Computer Science

## Instructor: Zhoulai Fu

### State University of New York, Korea

# Some Ocaml details

# Running a program with Ocaml Interpreter

- By convention, end your program with .ml

- Simple way to run your program is

  - **ocaml your_program.ml**

- Another good way is to run your program in the toplevel

  - **# #use "your_prgram.ml";;**

- Note: "#" after prompt, quotation marks, and the double semicolon.

- The toplevel approach gives you a chance to get the types.

# We will not need to use "ocamlc" for now

- Interpreter "ocaml" translates the source code into machine language one line at a time, and then executes that line before moving on to the next one.

- Compiler "ocamlc" reads the source code of a program and translates it into machine language all at once.

- Use Interpreter for quick testing.

- Use compiler for production.

# Double semicolon

- In general, not necessary in your Ocaml code, except

- Use it in Ocaml directives like #use "file.ml";;

# Single semicolon

- Semicolon is an expression separator, not a statement terminator

- Syntax for expression: e1; e2 ;…; e_n;

- Evaluation: Evaluate e1,…e_n, where e_n's value decides the type of the whole expression

- Typing: e1…e_{n-1} need to be of type unit. The type of the e_n is the type of the expression

# Single semicolon example

This code

```
let foo x y =
    print_endline "Now I'll add up two numbers";
    print_endline "Yes, seriously";
    x + y
```

is semantically equivalent to

```
let foo x y =
    let _ = print_endline "Now I'll add up two numbers" in
    let _ = print_endline "Yes, seriously" in
    x + y
```

# Single semicolon example (2)

- In "if-then-else", we need begin…end to enclose *e1;… e_n;* to avoid unexpected parsing

- begin…end is a more readable alternative to parentheses

# Shadowing is not mutation

Some uses of top level let bindings may look like mutation, but they actually aren't.

```
let x = 10
let x = x + 10
let () = Printf.printf "%d\n" x
```

In this example, the *x* in the printf expression is 20. Did we redefine *x* for the *whole program*?

```
let x = 10
let print_old_x () = Printf.printf "%d\n" x

let x = x + 10
let () = Printf.printf "%d\n" x   (* prints 20 *)
let () = print_old_x ()           (* prints 10 *)
```

# let _ = … and let () = …

- Both are definitions

- Syntax for let definitions is *let pattern = expression*

- Therefore, *let (x,y,z)= (4,5,6)* makes sense

- _ is a pattern for anything, or wildcard; () is a pattern for the single element of unit type

- The two are used to enforce evaluation

- Use let () = … instead of let _ = … if what follows is of type unit

# Ocaml debug 1 — typing

- Always fix typing errors first

- You can add your own types to make sure the your implementation corresponds to your thought

```
# let avg x y: float = (x+y)/2;;

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

# Ocam debug 2 — assert

```
# let avg x y = x +. y /.2.;;
val avg : float -> float -> float = <fun>

# assert (avg 2.0 3.0 = 2.5) ;;
Exception: Assert_failure ("//toplevel//", 1,
0).
```

# Ocaml debug 3 —print

- **Print statements.** Insert a *print statement* to ascertain the value of a variable. Suppose you want to know what the value of **x** is in the following function:

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

Just add the line below to print that value:

```
let inc x =
   let () = print_int(x) in   (* added *)
      x+1
```

# Ocaml debug 4— trace

```
# let rec fact x = if x = 1 then 1 else x * fact (x-1)
  ;;
val fact : int -> int = <fun>
# fact 5
  ;;
- : int = 120
# #trace fact;;
fact is now traced.
# fact 5;;
fact <-- 5
fact <-- 4
fact <-- 3
fact <-- 2
fact <-- 1
fact --> 1
fact --> 2
fact --> 6
fact --> 24
fact --> 120
- : int = 120
```