# CSE216
# Foundations of Computer Science

**Instructor: Zhoulai Fu**

**State University of New York, Korea**

Many slides taken from Cornell's CS3110. Thanks!
https://www.cs.cornell.edu/courses/cs3110/2014fa/lecture_notes.php

# Plan

- A review exercise
- Syntax Sugar (review-like)
- Two additional exercises
- Recitation this afternoon: More exercises. Homework of the week before will be explained next week.

# Exercise: unzip 3-element lists

```
unzip3 : ('a * 'b * 'c) list -> 'a list * 'b list * 'c list
```

```
# unzip3 [(1,2,3);(4,5,6);(7,8,9);(10,11,12)] ;;
- : int list * int list * int list =
([1; 4; 7; 10], [2; 5; 8; 11], [3; 6; 9; 12])
```

"A language that doesn't affect the way you think about programming is not worth knowing."

–Alan J. Perlis **(Turing Laureate)**

# 1. If expressions are just matches

- **`if`** expressions exist only in the *surface syntax* of the language
- Early pass in compiler can actually replace **`if`** expression with **`match`** expression, then compile the **`match`** expression instead

```
if e0 then e1 else e2
```

becomes...

```
match e0 with true -> e1 | false -> e2
```

because...

```
type bool = false | true
```

# 2. Option is a built-in datatype

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

- **`None`** and **`Some`** are constructors
- **`'a`** means "any type"

```
let string_of_intopt(x:int option) =
  match x with
    None     -> ""
  | Some(i) -> string_of_int(i)
```

# 3. List is another builtin datatype

```
type 'a list = Nil | Cons of 'a * 'a list

let rec append (xs: ‘a list)(ys: ‘a list) =
  match xs with
[] -> ys
| x::xs’ -> x :: (append xs’ ys)
```

- Ocaml uses [], ::, @ instead

# 4. Let expressions are pattern matches

- The syntax on the LHS of = in a let expression is really a pattern

```
let p = e
```

  - (Variables are just one kind of pattern)

- Implies it's possible to do this (e.g.):

```
let [x1;x2] = lst
```

# 5. Function arguments are patterns

A function argument can also be a pattern

– Match against the argument in a function call

```
let f p = e
```

Examples:

```
let sum_triple (x, y, z) =
    x + y + z

let sum_stooges {larry=x; moe=y; curly=z} =
    x + y + z
```

# Back to 'a
# (some textbooks call it alpha)

Length of a list:

```
let rec len (xs: int list) =
    match xs with
        [] -> 0
      | _::xs' -> 1 + len xs'
```

```
let rec len (xs: string list) =
    match xs with
        [] -> 0
      | _::xs' -> 1 + len xs'
```

No algorithmic difference! Would be silly to have to write function for every kind of list type…

# Type variables ‘a to the rescue

Use *type variable* to stand in place of an arbitrary type:

```
let rec len (xs: 'a list) =
   match xs with
       [] -> 0
     | _::xs’ -> 1 + len xs’
```

– Just like we use *variables* to stand in place of arbitrary values
– Creates a *polymorphic* function (“poly”=many, “morph”=form)

**Somewhat related to Java generics and C++ templates**

# BTW, Java generics looks like this

```
public class Box<T> {
  private T content;

  public void addContent(T content) {
    this.content = content;
  }

  public T getContent() {
    return content;
  }

  public static void main(String[] args) {
    Box<String> box1 = new Box<>();
    box1.addContent("Hello, world!");
    System.out.println(box1.getContent()); // Output: Hello, world!

    Box<Integer> box2 = new Box<>();
    box2.addContent(42);
    System.out.println(box2.getContent()); // Output: 42
  }
}
```

# Extended datatype syntax

```
type 'a t = C1 [of t1] | ...
| Cn [of tn]
```

- t1..tn are now allowed to mention t and 'a

# Exercise

- Define a Polymorphic Binary Tree Type in OCaml, called **binary_tree**
  - support storing elements of any type.
  - Use constructor Empty to create an empty binary tree
  - Use constructor Node to create a node in the tree that carries data

```
let mytree = Node (2, Node (1, Empty, Empty), Node (3, Empty, Empty))
```

- Define a function “height” to get a tree’s height; “mytree” above has height 2.

# Exercise

- Write a function *drop : int -> 'a list -> 'a list* such that *drop n lst* returns all but the first n elements of lst. If lst has fewer than n elements, return the empty list. Here, n can be any integer including negative number.