CSE216
Foundations of Computer Science

Instructor: Zhoulai Fu

State University of New York, Korea

Many slides taken from Prof. YoungMin Kwon. Thanks!

Procedural Abstraction

Computing pi with Nilakantha series

$$\pi = 3 + \frac{4}{2\times3\times4} - \frac{4}{4\times5\times6} + \frac{4}{6\times7\times8} - \frac{4}{8\times9\times10} + \cdots$$

Pseudo-code

$$\pi = 3 + \frac{4}{2\times3\times4} - \frac{4}{4\times5\times6} + \frac{4}{6\times7\times8} - \frac{4}{8\times9\times10} + \cdots$$

- guess = 3
- Loop
    - If guess is good return guess
    - else guess = guess + next_item
- end loop

pseudocode -> code v0.1

$$\pi = 3 + \frac{4}{2\times3\times4} - \frac{4}{4\times5\times6} + \frac{4}{6\times7\times8} - \frac{4}{8\times9\times10} + \cdots$$

- guess = 3
- Loop
    - If guess is good return guess
    - else guess = guess + next_item
- end loop

- let pi_iter guess =
    - if good_enough guess then guess
    - else pi_iter guess+next_item

code v0.1 -> v0.2

$$\pi = 3 + \frac{4}{2\times3\times4} - \frac{4}{4\times5\times6} + \frac{4}{6\times7\times8} - \frac{4}{8\times9\times10} + \cdots$$

- let pi_iter guess =
  - if good_enough guess then guess
  - else pi_iter guess+next_item

- let pi_iter guess old_guess tol=
  - if abs(guess-old guess)<tol then guess
  - else pi_iter guess+next_item guess tol

code v0.2 -> v0.3

$$\pi = 3 + \frac{4}{2\times3\times4} - \frac{4}{4\times5\times6} + \frac{4}{6\times7\times8} - \frac{4}{8\times9\times10} + \dots$$

- let pi_iter guess old_guess tol=
    - if abs(guess-old guess)<tol then guess
    - else pi_iter guess+next_item guess tol

- let pi_iter guess old_guess x sign tol=
    - if abs(guess-old guess)<tol then guess
    - else pi_iter guess+ sign*4/(x(x+1)(x+2)) guess tol

code v0.3 -> v0.4

$$\pi = 3 + \frac{4}{2\times3\times4} - \frac{4}{4\times5\times6} + \frac{4}{6\times7\times8} - \frac{4}{8\times9\times10} + \cdots$$

- let pi_iter guess old_guess x sign tol=
  - if abs(guess-old guess)<tol then guess
  - else pi_iter guess+ sign*4/(x(x+1)(x+2)) guess tol

```
let good_enough guess old_guess tol =
    (abs (guess -. old_guess)) <= tol;;

let term x sign =
    sign *. 4. /. (x *. (x +. 1.) *. (x +. 2.))

let rec pi_iter guess old_guess x sign tol =
```

code v0.3 -> v0.4 (cont.)

$$\pi = 3 + \frac{4}{2\times3\times4} - \frac{4}{4\times5\times6} + \frac{4}{6\times7\times8} - \frac{4}{8\times9\times10} + \cdots$$

- let pi_iter guess old_guess x sign tol=
  - if abs(guess-old guess)<tol then guess
  - else pi_iter guess+ sign*4/(x(x+1)(x+2)) guess tol

```
let good_enough guess old_guess tol =
    (abs (guess -. old_guess)) <= tol;;

let term x sign =
    sign *. 4. /. (x *. (x +. 1.) *. (x +. 2.))

let rec pi_iter guess old_guess x sign tol =
    if good_enough guess old_guess tol
    then guess
    else pi_iter (guess +. (term x sign))
                 guess
                 (x +. 2.)
                 (-. sign)
                 tol
```

Running the code

```
let rec pi_iter guess old_guess x sign tol =
    if good_enough guess old_guess tol
    then guess
    else pi_iter (guess +. (term x sign))
                 guess
                 (x +. 2.)
                 (-. sign)
                 tol
let pi tol =
    pi_iter 3. 0. 2. 1. tol

let _ = pi 1e-10
```

$$\pi = 3 + \frac{4}{2\times3\times4} - \frac{4}{4\times5\times6} + \frac{4}{6\times7\times8} - \frac{4}{8\times9\times10} + \dots$$

```
# #use "pi_iter.ml";;

val good_enough : float -> float -> float -> bool = <fun>
val term : float -> float -> float = <fun>
val pi_iter : float -> float -> float -> float -> float -> float…
val pi : float -> float = <fun>
- : float = 3.141592653589846
```

Expression let…in…

```
let <variable> = <expr1> in <expr2>
```

- let binding is equivalent to

```
( fun <variable> -> <expr2> ) <expr1>
```

```
let foo () =
    let x = 1 in
    let y = x + 1 in
    let z = y + 1 in
    z + 3
```

A nicer example

```
let calculate_discounted_price price customer_type =
  let standard_discount = 0.10 in
  let premium_discount = 0.20 in
  let gold_discount = 0.30 in
  let discount =
    match customer_type with
    | "standard" -> standard_discount
    | "premium" -> premium_discount
    | "gold" -> gold_discount
    | _ -> 0.0
  in
  let final_price = price -. (price *. discount) in
  (* Your code to round the final_price to two decimal places *)
  final_price
;;
```

Lab

- Exercise: Calculate the Area of a Circle

- Define a procedure to calculate the area of a circle. The procedure should take the radius of the circle as an argument and return the area. Use the **let ... in** construct to define a local variable for pi within the procedure.

Consider these functions `double` and `square` on integers:

```
let double x = 2 * x
let square x = x * x
```

```
val double : int -> int = <fun>
```

```
val square : int -> int = <fun>
```

Let's use these functions to write other functions that quadruple and raise a number to the fourth power:

- let quad = (*todo*)
- let fourth = (*todo*)

There is an obvious similarity between these two functions: what they do is apply a given function twice to a value. By passing in the function to another function `twice` as an argument, we can abstract this functionality:

```
let twice f x = f (f x)
```

```
val twice : ('a -> 'a) -> 'a -> 'a = <fun>
```

The function `twice` is higher-order: its input `f` is a function. And—recalling that all OCaml functions really take only a single argument—its output is technically `fun x -> f (f x)`, so `twice` returns a function hence is also higher-order in that way.

Using `twice`, we can implement `quad` and `fourth` in a uniform way:

- let quad = (*todo*)
- let fourth = (*todo*)

Exercise #9

- Pack consecutive duplicates of list elements into sublists.

```
# pack ["a"; "a"; "a"; "a"; "b"; "c"; "c"; "a"; "a"; "d";
"d"; "e"; "e"; "e"; "e"];;
- : string list list =
[["a"; "a"; "a"; "a"]; ["b"]; ["c"; "c"]; ["a"; "a"];
["d"; "d"];
 ["e"; "e"; "e"; "e"]]
```