COSE212: Programming Languages

Lecture 4 — Recursive and Higher-Order Programming

Hakjoo Oh
2016 Fall

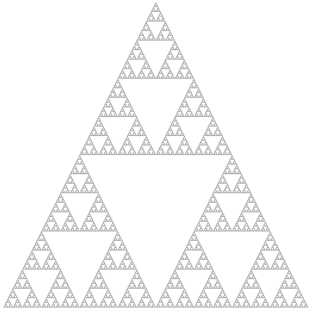
Recursive and Higher-Order Programming

Heavily used in functional programming languages:

- Recursive programming provides a powerful problem-solving method.
- Higher-order programming provides expressiveness.

Why Recursive Thinking?

Describe an algorithm to draw the following:



Recursive Problem-Solving Strategy

- If the problem is sufficiently small, directly solve the problem.
- Otherwise,
  1. Split the problem up into smaller problems with the same structure as original.
  2. Solve each of those smaller problems.
  3. Combine the results to get the overall solution.

Example: list length

- If the list is empty, the length is 0.
- Otherwise,
  1. The list can be split into its head and tail.
  2. Compute the "lengths" of the tail.
  3. The overall solution is the length of the tail plus one.

In OCaml:

```
let rec length l =
  match l with
  | [] -> 0
  | hd::tl -> 1 + length tl
```

Exercise 1: append

Write a function that appends two lists:

```
# append [1; 2; 3] [4; 5; 6; 7];;
- : int list = [1; 2; 3; 4; 5; 6; 7]
# append [2; 4; 6] [8; 10];;
- : int list = [2; 4; 6; 8; 10]
```

Exercise 2: reverse

Write a function that reverses a given list:

```
val reverse : 'a list -> 'a list = <fun>
# reverse [1; 2; 3];;
- : int list = [3; 2; 1]
# reverse ["C"; "Java"; "OCaml"];;
- : string list = ["OCaml"; "Java"; "C"]
```

Exercise 3: nth-element

Write a function that computes nth element of a list:

```
# nth [1;2;3] 0;;
- : int = 1
# nth [1;2;3] 1;;
- : int = 2
# nth [1;2;3] 2;;
- : int = 3
# nth [1;2;3] 3;;
Exception: Failure "list is too short".
```

```
let rec nth l n =
  match l with
  | [] -> raise (Failure "list is too short")
  | hd::tl -> (* ... *)
```

Exercise 4: remove-first

Write a function that removes the first occurrence of an element from a list:

```
# remove_first 2 [1; 2; 3];;
- : int list = [1; 3]
# remove_first 2 [1; 2; 3; 2];;
- : int list = [1; 3; 2]
# remove_first 4 [1;2;3];;
- : int list = [1; 2; 3]
# remove_first [1; 2] [[1; 2; 3]; [1; 2]; [2; 3]];;
- : int list list = [[1; 2; 3]; [2; 3]]
```

```
let rec remove_first a l =
  match l with
  | [] -> []
  | hd::tl -> (* ... *)
```

Exercise 5: insert

Write a function that inserts an element to a sorted list:

```
# insert 2 [1;3];;
- : int list = [1; 2; 3]
# insert 1 [2;3];;
- : int list = [1; 2; 3]
# insert 3 [1;2];;
- : int list = [1; 2; 3]
# insert 4 [];;
- : int list = [4]
```

```
let rec insert a l =
  match l with
  | [] -> [a]
  | hd::tl -> (* ... *)
```

Exercise 6: insertion sort

Write a function that performs insertion sort:

```
let rec sort l =
  match l with
  | [] -> []
  | hd::tl -> insert hd (sort tl)
```

cf) Compare with “C-style” non-recursive version:

```
for (c = 1 ; c <= n - 1; c++) {
 d = c;
 while ( d > 0 && array[d] < array[d-1]) {
   t          = array[d];
   array[d]   = array[d-1];
   array[d-1] = t;
   d--;
 }
}
```

Recursion in ML is Not Expensive

In languages like C, recursion should be avoided because function call consumes additional memory:

```
void f() { f(); }        /* stack overflow */
```

The same program in ML iterates forever:

```
let rec f () = f ()
```

Tail-Recursive Functions

More precisely, *tail-recursive functions* are not expensive in ML. A recursive call is a tail call if there is nothing to do after the function returns.

-
  ```
  let rec last l =
    match l with
    | [a] -> a
    | _::tl -> last tl
  ```

-
  ```
  let rec factorial a =
    if a = 1 then 1
    else a * factorial (a - 1)
  ```

Languages like ML, Scheme, Scala, and Haskell do *tail-call optimization*, so that tail-recursive calls do not consume additional amount of memory.

Transforming to Tail-Recursive Functions

Non-tail-recursive factorial:

```
let rec factorial a =
  if a = 1 then 1
  else a * factorial (a - 1)
```

Tail-recursive version:

```
let rec fact product counter maxcounter =
  if counter > maxcounter then product
  else fact (product * counter) (counter + 1) maxcounter

let factorial n = fact 1 1 n
```

Higher-Order Functions

Higher-order functions:

- functions that take other functions or return functions as results
- a powerful tool for code reuse

Example 1: map

Three similar functions:

```
let rec inc_all l =
  match l with
  | [] -> []
  | hd::tl -> (hd+1)::(inc_all tl)

let rec square_all l =
  match l with
  | [] -> []
  | hd::tl -> (hd*hd)::(square_all tl)

let rec cube_all l =
  match l with
  | [] -> []
  | hd::tl -> (hd*hd*hd)::(cube_all tl)
```

Example 1: map

The code pattern can be captured by the higher-order function `map`:

```
let rec map f l =
  match l with
  | [] -> []
  | hd::tl -> (f hd)::(map f tl)
```

With `map`, the functions can be defined as follows:

```
let inc x = x + 1
let inc_all l = map inc l

let square x = x * x
let square_all l = map square l

let cube x = x * x * x
let cube_all l = map cube l
```

Or, using nameless functions:

```
let inc_all l = map (fun x -> x + 1) l
let square_all l = map (fun x -> x * x) l
let cub_all l = map (fun x -> x * x * x) l
```

Example 2: fold

Two similar functions:

```
let rec sum l =
  match l with
  | [] -> 0
  | hd::tl -> hd + (sum tl)

let rec prod l =
  match l with
  | [] -> 1
  | hd::tl -> hd * (prod tl)

# sum [1; 2; 3; 4];;
- : int = 10
# prod [1; 2; 3; 4];;
- : int = 24
```

Example 2: fold

The two functions have the following form:

```
sum [x1; x2; ...; xn] = x1 + (x2 + (... + (xn + 0)))
```

```
prod [x1; x2; ...; xn] = x1 * (x2 * (... * (xn * 1)))
```

This pattern is captured by `fold`:

```
fold f [x1; x2; ...; xn] a = f x1 (f x2 (... (f xn a)))
```

```
let sum  lst = fold (fun x y -> x + y) lst 0
let prod lst = fold (fun x y -> x * y) lst 1
```

or,

```
let sum  l = fold (+) l 0
let prod l = fold (*) l 1
```

Example 2: fold

The definition of fold:

```
let rec fold f l a =
  match l with
  | [] -> a
  | hd::tl -> f hd (fold f tl a)
```

Exercises

Re-write the following functions in one-line using `fold`:

-
  ```
  let rec length l =
    match l with
    | [] -> 0
    | hd::tl -> 1 + length tl
  ```

-
  ```
  let rec reverse l =
    match l with
    | [] -> []
    | hd::tl -> (reverse tl) @ [hd]
  ```

-
  ```
  let rec is_all_pos l =
    match l with
    | [] -> true
    | hd::tl -> (hd > 0) && (is_all_pos tl)
  ```