COSE212: Programming Languages

Lecture 4 — Recursive and Higher-Order Programming

Hakjoo Oh
2019 Fall

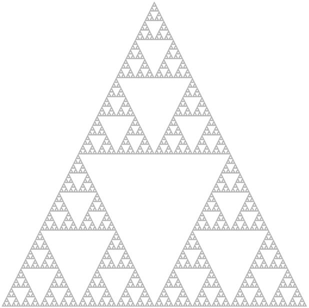
Why Recursive and Higher-Order Programming?

Recursion and higher-order functions are essential in functional programming:

- Recursion is used instead of loops and provides a powerful problem-solving method.
- Higher-order functions provide a powerful means for abstractions (i.e. the capability of combining simple ideas to form more complex ideas).

The Power of Recursive Thinking

Quiz) Describe an algorithm to draw the following pattern:



Recursive Problem-Solving Strategy

- If the problem is sufficiently small, directly solve the problem.
- Otherwise,
  1. Decompose the problem to smaller ones 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 length of the tail.
  3. The overall solution is the length of the tail plus one.

```
# length [];;
- : int = 0
# length [1;2;3];;
- : int = 3
```

```
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]
```

```
let rec append l1 l2 =
```

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"]
```

```
let rec reverse l =
```

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--;
 }
}
```

cf) Imperative vs. Functional Programming

- Imperative programming focuses on describing **how** to accomplish the given task:

  ```
  int factorial (int n) {
    int i; int r = 1;
    for (i = 0; i < n; i++)
      r = r * i;
    return r;
  }
  ```

  Imperative languages encourage to use statements and loops.

- Functional programming focuses on describing **what** the program must accomplish:

  ```
  let rec factorial n =
    if n = 0 then 1 else n * factorial (n-1)
  ```

  Functional languages encourage to use expressions and recursion.

Is Recursion Expensive?

- In C and Java, we are encouraged to avoid recursion because function calls consume additional memory.

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

- This is not true in functional languages. 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.

cf) 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 are functions that manipulate procedures; they take other functions or return functions as results.
- Higher-order functions provide a powerful tool for building abstractions and allow code reuse.

Abstractions

- A good programming language provides powerful abstraction mechanisms (i.e. the means for combining simple ideas to form more complex ideas). E.g.,
  - variables: the means for using names to refer to values
  - functions: the means for using names to refer to compound operations
- For example, suppose we write a program that computes $\mathbf{2^3 + 3^3 + 4^3}$.
  - Without functions, we have to work at the low-level:
    ```
    2*2*2 + 3*3*3 + 4*4*4
    ```
  - Functions allow use to express the concept of cubing and write a high-level program.
    ```
    let cube n = n * n * n
    in cube 2 + cube 3 + cube 4
    ```
- Every programming language provides variables and functions.
- Not all programming languages provide mechanisms for abstracting same programming patterns.
- Higher-order functions serve as powerful mechanisms for this.

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
```

Exercise

1. What is the type of `map`?
2. What does

```
map (fun x mod 2 = 1) [1;2;3;4]
```

   evaluate to?

Example 2: filter

```
let rec even l =
  match l with
  | [] -> []
  | hd::tl ->
    if hd mod 2 = 0 then hd::(even tl)
    else even tl

let rec greater_than_five l =
  match l with
  | [] -> []
  | hd::tl ->
    if hd > 5 then hd::(greater_than_five tl)
    else greater_than_five tl
```

Example 2: filter

```
filter : ('a -> bool) -> 'a list -> 'a list
```

- `even =`
- `greater_than_five`

Example 3: fold_right

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 3: fold_right

The code pattern can be captured by the higher-oder function `fold`:

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

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

fold_right vs. fold_left

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

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

fold_right vs. fold_left

- Direction:
  - `fold_right` works from left to right:

    `fold_right f [x;y;z] init = f x (f y (f z init))`

  - `fold_left` works from right to left

    `fold_left f init [x;y;z] = f (f (f init x) y) z`

  - They may produce different results if `f` is not associative
- Types:

```
fold_right : ('a -> 'b -> 'b) -> 'a list -> 'b -> 'b
fold_left  : ('a -> 'b -> 'a) -> 'a -> 'b list -> 'a
```

- `fold_left` is tail-recursion

Exercises

- ```
  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)
  ```
- `map f l =`
- `filter f l =`
  ```

Functions as Returned Values

Functions can be returned from the other functions. For example, let f and g be two one-argument functions. The composition of f after g is defined to be the function $x \mapsto f(g(x))$.
In OCaml:

```
let compose f g = fun x -> f(g(x))
```

What is the value of the expression?

```
((compose square inc) 6)
```

Summary

Two mechanisms play key roles for writing concise and readable code in programming:

- Recursion provides a powerful problem-solving strategy.
- Higher-order functions provide a powerful means for abstractions.