CS 321 Programming Languages

Intro to OCaml – User-Defined Data Types

Baris Aktemur

Özyeğin University

Last update made on Monday 16th October, 2017 at 15:34.

Much of the contents here are taken from Elsa Gunter and Sam Kamin's OCaml notes available at http://courses.engr.illinois.edu/cs421

Defining your own data types

Users can define custom data types by specifying the *constructors.*

```
# type weekday = Monday | Tuesday | Wednesday
               | Thursday | Friday | Saturday | Sunday;;

# Monday;;
- : weekday = Monday
# let today = Thursday;;
val today : weekday = Thursday
```


Defining your own data types

- Similar to defining your own classes in Java.
- Data type c'tor names start with Uppercase letter.
- Can do pattern matching.

```
# let day_after day =
    match day with
    | Monday -> Tuesday | Tuesday -> Wednesday | Wednesday -> Thursday
    | Thursday -> Friday | Friday -> Saturday | Saturday -> Sunday
    | Sunday -> Monday;;

val day_after : weekday -> weekday

# day_after Sunday;;
- : weekday = Monday
# day_after today;;
- : weekday = Friday
```


Function over enumerations

```
# let rec days_later n day =
    match n with
    | 0 -> day
    | _ -> if n > 0 then day_after(days_later (n-1) day)
           else days_later (n+7) day;;

val days_later : int -> weekday -> weekday

# days_later 2 Tuesday;;
- : weekday = Thursday
# days_later (-1) Wednesday;;
- : weekday = Tuesday
# days_later (-4) Monday;;
- : weekday = Thursday
```

Constructors with parameters

```
# type id = DriversLicense of int
          | SocialSecurity of int
          | Name of string;;

# let check_id id =
    match id with
    | DriversLicense num ->
        not (List.exists (fun n -> n = num) [13570; 99999])
    | SocialSecurity num -> num < 900000000
    | Name str -> not(str = "John Doe");;

val check_id : id -> bool

# check_id (Name "John Doe");;
- : bool = false
# check_id (Name "Obi Wan Kenobi");;
- : bool = true
# check_id (DriversLicense 12345);;
- : bool = true
```


Exercise

Define a data type shape. A shape can be a circle, square or a triangle. Circle has a radius, square has a side length, triangle has three sides.

```
# type shape =



# let c = Circle 5.7
  and t = Triangle (2.0, 3.0, 4.0);;

val c : shape = Circle 5.7
val t : shape = Triangle (2.0,3.0,4.0)
```


Exercise

Define a data type shape. A shape can be a circle, square or a triangle. Circle has a radius, square has a side length, triangle has three sides.

```
# type shape =

               Circle of float
             | Square of float
             | Triangle of float * float * float;;

# let c = Circle 5.7
  and t = Triangle (2.0, 3.0, 4.0);;

val c : shape = Circle 5.7
val t : shape = Triangle (2.0,3.0,4.0)
```


Constructors with parameters

```
# let area s =
    match s with
    | Circle r -> 3.14 * r * r
    | Square d -> d * d
    | Triangle (a,b,c) -> let s = (a+b+c)/2.0
                          in sqrt(s*(s-a)*(s-b)*(s-c));;

val area : shape -> float

# area c;;
- : float = 102.0186
# area t;;
- : float = 2.90473751
# area (Triangle(3.0,4.0,5.0));;
- : float = 6.0
```

Recursive data types

```
# type tree = Leaf of int
          | Node of (tree * tree);;

# let myTree = Node(Node(Leaf 3,
                         Node(Leaf 5, Leaf 8)),
                    Node(Leaf 9, Leaf 11));;

# let rec contains t n =
    match t with
    | Leaf i -> i = n
    | Node (t1,t2) -> contains t1 n || contains t2 n;;

val contains : tree -> int -> bool

# contains myTree 8;;
- : bool = true
# contains myTree 6;;
- : bool = false
```


Functions on recursive data types

```
# let rec flatten t =
    match t with
    | Leaf num -> [num]
    | Node(t1, t2) -> flatten t1 @ flatten t2;;

val flatten : tree -> int list

# flatten myTree;;
- : int list = [3; 5; 8; 9; 11]
```

See a better implementation

Exercise

```
# let rec mirror t =
    ???

val mirror : tree -> tree

# mirror myTree;;
- : tree =
  Node (Node (Leaf 11, Leaf 9), Node (Node (Leaf 8, Leaf 5), Leaf 3))
# flatten(mirror myTree);;
- : int list = [11; 9; 8; 5; 3]
```


Mapping a function on int binary tree

```
# let rec treeMap f t =
    match t with
    | Leaf num -> Leaf(f(num))
    | Node(t1,t2) -> Node(treeMap f t1, treeMap f t2);;

val treeMap : (int -> int) -> tree -> tree

# treeMap (fun n -> n*2) myTree;;
- : tree =
  Node (Node (Leaf 6, Node (Leaf 10, Leaf 16)),
        Node (Leaf 18, Leaf 22))
```

Exercise

```
# let rec tally t =
    ???

val tally : tree -> int

# tally mytree;;
- : int = 36
```

▸ See a better implementation

Polymorphic data types

```
# type 'a tree = Leaf of 'a
               | Node of ('a * 'a tree * 'a tree);;

# let myIntTree = Node(4, Node(8, Leaf 5, Leaf 2),
                          Node(3, Leaf 7, Node(9, Leaf 12,
                                                  Leaf 6)));;
# let myCharTree = Node('a', Leaf 'b',
                             Node('c', Leaf 'd', Leaf 'e'));;
# let rec size t =
    match t with
    | Leaf n -> 1
    | Node(_,t1,t2) -> 1 + size t1 + size t2;;
val size : 'a tree -> int

# size myCharTree;;
- : int = 5
# size myIntTree;;
- : int = 9
```

▸ See a better implementation

Exercise

```
# let rec flatten t =
    ???

val flatten : 'a tree -> 'a list

# flatten myIntTree;;
- : int list = [5; 8; 2; 4; 7; 3; 12; 9; 6]
# flatten myCharTree;;
- : char list = ['b'; 'a'; 'd'; 'c'; 'e']
```


Exercise

```
# let rec flatten t =
    ???

val flatten : 'a tree -> 'a list

# flatten myIntTree;;
- : int list = [5; 8; 2; 4; 7; 3; 12; 9; 6]
# flatten myCharTree;;
- : char list = ['b'; 'a'; 'd'; 'c'; 'e']
```

▸ See a better implementation

Note: Polymorphic data types are homogeneous. (e.g. Node('a', Leaf 1, Leaf 'b') gives error.

Exercise

```
# let rec contains t x =
    ???

val contains : 'a tree -> 'a -> bool

# contains myCharTree 'c';;
- : bool = true
# contains myIntTree 0;;
- : bool = false
```


“option” type

Useful for partial functions that cannot calculate a result for every input. Often replaces exceptions.

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

(* Return the first element that satisfies p *)
# let rec first p lst =
    match lst with
    | [] -> None
    | x::xs -> if p x then Some(x) else first p xs;;

val first : ('a -> bool) -> 'a list -> 'a option

# first (fun x -> x > 3) [1;3;4;5;2];;
- : int option = Some 4
# first (fun x -> x > 5) [1;3;4;5;2];;
- : int option = None
```


“option” type

```
(* Return the last element that satisfies p *)
# let rec last p lst =
    match lst with
    | [] -> None
    | x::xs -> (match last p xs with
                | None -> if p x then Some x else None
                | Some y -> Some y);;

val last : p:('a -> bool) -> lst:'a list -> 'a option

# last (fun n -> n%2 = 0) [3;6;2;9;8;12;15];;
- : int option = Some 12
# last (fun n -> n%7 = 0) [3;6;2;9;8;12;15];;
- : int option = None
```

See an implementation with fold_left.

option type is defined in the pervasive environment.

Arbitrary trees (not just binary)

```
# type 'a tree = Node of ('a * 'a tree list);;

# let singleNode = Node(3, []);;

# let mytree = Node(3, [Node(5, []);
                       Node(8, []);
                       Node(11, [Node(6, [])])]);;
```

Functions on arbitrary trees

```
# let rec size t =
    match t with
    | Node(v,children) ->
       1 + List.fold_left (fun acc n -> acc + size n) 0 children;;

val size : 'a tree -> int

# size singleNode;;
- : int = 1
# size mytree;;
- : int = 5
```

See a better implementation

Functions on arbitrary trees

```
# let rec sum t =
    match t with
    | Node(v,children) ->
      v + List.fold_left (fun acc n -> acc + sum n) 0 children;;

val sum : int tree -> int

# sum mytree;;
- : int = 33
# sum singleNode;;
- : int = 3
```


Exercise

Manually define a data type to represent lists.

```
# type mylist = Empty | Cons of int * mylist;;

# let list1 = Cons (3, Cons (4, Cons(5, Empty)));;
val list1 : mylist = Cons (3,Cons (4,Cons (5,Empty)))
```

Write the function sum: `mylist -> int`.

Efficiency

Functions we have defined in this lecture are inefficient. We focused on correctness, rather than efficiency. Now implement improved versions of the functions. Good luck.

Functions on recursive data types

```
# let rec flatten t =
    (* Auxiliary function with accumulator arg. *)
    let rec aux t acc =
      match t with
      | Leaf num -> num::acc
      | Node(t1, t2) -> aux t1 (aux t2 acc)
    in aux t [];;

val flatten : tree -> int list

# flatten myTree;;
- : int list = [3; 5; 8; 9; 11]
```

Click me!

Exercise

```
# let rec tally t =
    let rec aux t acc =
      match t with
      | Leaf(num) -> acc + num
      | Node(t1,t2) -> aux t2 (aux t1 acc)
    in aux t 0;;

val tally : tree -> int

# tally mytree;;
- : int = 36
```

See the original version

Polymorphic data types

```
# type 'a tree = Leaf of 'a
               | Node of ('a * 'a tree * 'a tree);;

# let myIntTree = Node(4, Node(8, Leaf 5, Leaf 2),
                          Node(3, Leaf 7, Node(9, Leaf 12,
                                                 Leaf 6)));;
# let rec size t =
    let rec aux t acc =
      match t with
      | Leaf n -> acc + 1
      | Node(_,t1,t2) -> aux t1 (aux t2 (acc+1))
    in aux t 0;;

val size : 'a tree -> int

# size myIntTree;;
- : int = 9
```

See the original version

Exercise

```
# let rec flatten t =
    let rec aux t acc =
      match t with
      | Leaf n -> n::acc
      | Node(v,t1,t2) -> aux t1 (v::(aux t2 acc))
    in aux t [];;

val flatten : 'a tree -> 'a list

# flatten myIntTree;;
- : int list = [5; 8; 2; 4; 7; 3; 12; 9; 6]
# flatten myCharTree;;
- : char list = ['b'; 'a'; 'd'; 'c'; 'e']
```

See the original version

How can you define other orderings?

"option" type

```
# let last p lst =
    List.fold_left (fun acc x -> if p x then Some x else acc) None lst;;

val last : p:('a -> bool) -> lst:'a list -> 'a option

# last (fun n -> n%2 = 0) [3;6;2;9;8;12;15];;
- : int option = Some 12
# last (fun n -> n%7 = 0) [3;6;2;9;8;12;15];;
- : int option = None
```

See the original version

Functions on arbitrary trees

```
# let rec size t =
    let rec aux (Node(v,children)) acc =
      List.fold_left (fun a x -> aux x a) (acc+1) children
    in aux t 0;;

val size : 'a tree -> int

# size singleNode;;
- : int = 1
# size mytree;;
- : int = 5
```

See the original version