CISS445 Lecture 15: Arrays and hashtables 5

Yihsiang Liow

February 19, 2020

Table of contents I

Arrays I

- This set of notes are on non-functional features of OCAML. OCAML is a multi-paradigm languages: it supports procedural, object- oriented, functional, non-functional thinking.
- Try this:

```
let a = [| 2;3;5;7 |];;
let length = Array.length a;;
let b = a.(2);;
a.(2) <- 999;;
a;;
b;;
```

Arrays II

- Watch out! An array is mutable, i.e., its value can change.
- This breaks from other OCAML features where the code has no **"side-effects"** where once a name is bound to a value then this value cannot change. Of course there might be multiple similar names in the environment:

$$\rho = \{a \to 1, b \to 2, a \to 3\}$$

But the earliest a is always bound to 3. We say that a is **immutable**.

Arrays III

- **Pure functional programming** has no side effects. For instance if `(f a)` is 42 now, then `(f a)` is still 42 later. You cannot possible have

```
... (f a) ... (* (f a) is 42 *)
...
...              (* some OCAML expression ... *)
...
... (f a) ... (* causing a side effect: (f a) is 0 *)
```

Arrays IV

- Try this:

```
let a = [| 2;3;5;7 |];;
let b = Array.make 5 (-1);;
let c = Array.concat [a;b;a];;
let d = Array.append a b;;
let e = Array.sub c 1 5;;
```

- You can also create an array using a formula. Try this:

```
let a = Array.init 5 (fun n -> n * n);;
```

Arrays V

- Watch out!!! First try this ...

```
let a = [| 2;3;5;7 |];;
let b = a;;
a.(0) <- 42;
a;;
b;;
```

- ... and then try this:

```
let a = [| 2;3;5;7 |];;
let b = Array.copy a;;
a.(0) <- 42;
a;;
b;;
```

Arrays VI

- Between lists and arrays:

```
let xs = [1;2;3];;
let ys = Array.of_list xs;;
let zs = Array.to_list ys;;
xs = zs;;
```

Arrays VII

- The OCAML array is not a recursive data structure like the OCAML list. So you cannot write an array function using structure recursion on the array.
- But it's a simple exercise to write a tail recursion to iterate through the values of the array. Try this:

```
let rec arr_loop f arr i n acc =
    if i = n then
        acc
    else
        arr_loop f arr (i + 1) n (g arr.(i) acc)
;;
let s = arr_loop (+) [|1;2;3|] 0 3 0;;
```

Arrays VIII

- Here's one that's easier to use where the index runs from 0 to the last index value:

```
let rec arr_loop2 f arr acc =
    arr_loop f arr 0 (Array.length arr) acc
;;
let s2 = arr_loop2 (+) [|1;2;3|] 0;;
```

Arrays IX

- Exercise. Write a map function on array. Call it `arr_map`. In other words `(arr_map f a)` applies function `f` to all values of the array `a`. (OCAML's `Array.iter` does exactly that.)
- Exercise. Write a filter function on array. Call it `arr_filter`. In other words `(arr_filter f arr)` create an array of values from `arr` that passes the boolean function `f`.
- Exercise. Write a fold left function on array. Call it `arr_foldleft`. It behaves just like fold left on lists.
- Exercise. Write a fold right function on array. Call it `arr_foldright`. It behaves just like fold right on lists.

Arrays X

- Exercise. Write a function `arr_swap` such that `(arr_swap arr0 arr1 i j)` will swap the index i value of `arr0` with the index j value of `arr1`.
- Exercise. Implement the following sorting algorithms on arrays:
  - bubble sort
  - selection sort
  - insertion sort
  - mergesort
  - quicksort

  Of course you have to modify these algorithms from CISS350 to use recursion instead of for- and while-loops.

Arrays XI

- Exercise. Implement the maxheap operations on an array so that you have a (max) priority queue. Implement the heapsort algorithm.
- Exercise. Implement binary search on a sorted array.

Arrays XII

- Of course you can create a 2d array:

```
let a = [| [| 1;2;3 |];
           [| 4;5;6 |];
           [| 7;8;9 |] |];;
let x = a.(2).(0);;
```

- Here's a convenient function to create a 2d array (matrix) that is filled with the same value:

```
let a = Array.make_matrix 2 3 42;;
```

Arrays XIII

- Exercise. A **directed graph**, which is a bunch of dots (called **nodes** or **vertices**) and lines with arrowheads (called **directed edges** or **arcs**). Suppose the nodes are v_0, ..., v_{n-1}. We can represent a directed graph with a matrix m: If v_i is connected to v_j, then the (i, j) entry is 1; otherwise it's 0. This matrix is called the **adjacency matrix**. The number of nodes that node v_i points to is called the **out degree** of v_i. The number of nodes pointing to v_i is called the **in degree**. Write a function, `indeg` that accepts an adjacency matrix and an integer i (for node v_i) and computes the in degree of v_i. Do the same for the out degree.

Hashtable I

- Recall from CISS350 that a hash table is a data structure that allows you to add key-value pairs, delete key-value pairs, find a key-value pair by key, all with an average runtime of $O(1)$.
- Try this:

```
# let ht = Hashtbl.create 10;;
val ht : ('_a, '_b) Hashtbl.t = <abstr>
```

  At this point the key type is `'_a` and the value type is `'_b`. This hashtable has an initial size of approximate 10.

- Continue with this:

```
Hashtbl.add ht "john" 5;;
Hashtbl.add ht "sue" 6;;
Hashtbl.add ht "tom" 7;;
Hashtbl.find ht "john";;
```

Hashtable II

- You can remove an entry:

```
Hashtbl.remove ht "john";;
Hashtbl.find ht "john";;
```

You should get an error.

- You can modify the value for a key:

```
Hashtbl.add ht "tom" 123;
Hashtbl.find ht "tom";
Hashtbl.replace ht "tom" 456;
Hashtbl.find ht "tom";
```

- You can check if a key is in the hashtable:

```
Hashtbl.mem ht "tom";
Hashtbl.mem ht "zaphod";
```

Hashtable III

- Exercise. Is the OCAML hashtable homogeneous?
- Exercise. What types are allowed for keys? What about values?
- Exercise. How do you find the number of entries in your hashtable?

Hashtable IV

- You can iterate through your hashtable using

```
Hashtbl.iter f ht;;
```

where `f` is a curried function that takes the key and value of an entry in the ht. For instance to print the entries in your hashtable do this:

```
let f = fun k -> fun v ->
    Printf.printf "%s:%s\n" k v;;
Hashtbl.iter f ht;;
```

Hashtable V

- You can clear your ht by doing this:

```
Hashtbl.clear ht;;
```

If you do this:

```
Hashtbl.reset ht;;
```

then you are clearing the `ht` and resetting the size to the intial size.

Hashtable VI

- Exercise.
  - Write a function `ht_keys` that accepts a hashtable `ht` and return all the keys in `ht` as a list.
  - Write a function `ht_values` that accepts a hashtable `ht` and return all the values in `ht` as a list.
  - Write a function `ht_items` that accepts a hashtable `ht` and returns a list of key-values pairs (as tuples) in `ht`.
  - Write a function `ht_add` that accepts hashtable and a list of key-value pairs (tuples) and add the entries into the hashtable.
  - Write a function `ht_update` that accepts hashtables `ht0` and `ht1` and add the key-values in `ht1` to `ht0`.

Hashtable VII

- Exercise.
  - Perform a character frequency analysis on a text file. (Go to gutenberg.net and find a large ebook.) For each character, print the frequency (the probabilty).
  - Perform character frequency analysis on 10 books. Do they all have the same top three frequency characters?
  - What is the most common character to follow e?
  - What is the most common character to precede e?
  - Perform a word frequency analysis between two authors – can they be identified by their word frequency?

Hashtable VIII

- Exercise.
  - Perform a word frequency analysis on a text file. (Go to gutenberg.net and find a large ebook.) For each word, print the frequency (the probabilty). (Of course there will be a huge number of words!)
  - Perform word frequency analysis on 10 books. Do they all have the same top three frequencies?
  - What is the average distance between two pairs of words?
  - Perform a word frequency analysis between two authors – can they be identified by their word frequency? What about word distance?

Hashtable IX

- Exercise. Continuing the graph exercise: Given the graph (as an adjacency matrix), get integers i and j from the user and then compute the path for node v_i to v_j using (a) breadth first search and (b) depth first search. Note that this is a graph – you need to avoid going into an infinite loop. You can do that by remembering the the nodes which are already visited. (There are many graph search algorithms. Feel free to use google to find these algorithms and then implement them using OCAML.)

Refs I

- First try this:

```
let x = 1;;
let f a = a + x;;
f 2;;                    (* 3 *)
let x = 2;;
f 2;;                    (* still 3 *)
```

- Now try this:

```
let x = ref 1;;      (* x bound to int ref cell *)
let f a = a + !x;; (* !x is the value of cell *)
f 2;;                    (* 3 *)
x := 1000;;
f 2;;                    (* 1002 *)
```

See it? Get it?

Refs II

- Reference cells are pointers. In the above `ref 1` creates a reference cell (pointer) that points to an integer 1 in the heap. After binding `x` to `ref 1`, `!x` is the value that the reference cell (the pointer) is pointing to. In C++–speak, roughly speaking:

```
let x = ref 1;;          int * x = new int; *x = 1;
let y = !x;;             int y = *x;
```

  (That's why C++ is important because if you don't have knowledge and language of pointers, the above would be hard to explain.)
- Suppose `x` is bound to a ref cell. Then
  - `!x` is the value that `x` points to
  - `x := v` means set the value `x` points to to `v`

Refs III

- Therefore a name that is bound to a ref cell is like a variable in the sense of C++ and Python.

Refs IV

- Exercise.

```
let x = ref 1;;

let a0 = !x;;    (* what's the diff between a0, a1? *)
let a1 = x;;
a0;;             (* what do you see? *)
a1;;             (* what do you see? *)
let a0 = 1000;; (* does this change !x ? *)
x;;
a1 := 1000;;     (* does this change !x ? *)
x;;
```

Refs V

- Exercise.

```
let x = ref 1;;

let a0 = !x;;    (* what is a0? *)
x := 2;;
let a0 = !x;;    (* what is a0? *)
```

- Exercise. In the above `x` is bound to the `ref` of an int. After that can x be bound to a ref of another type?
- Exercise. Can you bind `x` is a user defined type? Such as a variant type? After that can you change the ref?

Refs I

- You can of course do loops in OCAML using recursion.

```
let sum n = (* Evaluates 1 + 2 + 3 + ... + n *)
  let rec sum2 i n acc =
    if i > n then
      acc
    else
      sum2 (i + 1) n (acc + i)
  in
    sum2 0 n 0
;;
```

- But you can do for-loops too ...

Refs II

- Try:

```
for i = 1 to 10 do
    Printf.printf "%d\n" i
done;
;;
```

- If you want to compute something in the loop:

```
let result = ref 0;;
for i = 1 to 10 do
    result := !result + i
done;
print_int !result
;;
```

- ... and you can put the above in a function:

Refs III

```
let sum n =
  let result = ref 0 in
    for i = 1 to n do
       result := !result + i
    done;
  !result
;;
```

- Of course if OCAML has for-loop, then it's not too shocking to see this:

Refs IV

```
let sum n =
  let result = ref 0 in
  let i = ref 0 in
    while !i <= n do
      result := !result + !i;
      i := !i + 1;
    done;
  !result
;;
```

- Exercise. Implement bubblesort, selection sort, insertion sort. The input in all cases are arrays.
- Exercise. Implement mergesort. The input is in arrays. Use for- or while-loops for merge.

Refs V

- Exercise. Implement quicksort. The input is in arrays. Use for- or while-loop for partition.
- In the following, let minheaps be implemented using arrays. Implement heapify up and down and extract root for minheap; use for-loops. Implement heapsort. Use for- or while-loops.