CS 131 Discussion 1

Li Zhang

SEASnet Environment Set Up

1. SEASnet server logon:

   ```
   ssh <username>@lnxsrv.seas.ucla.edu
   ```

2. Path set up:

   1. `vi $HOME/.profile`
   2. `export PATH=/usr/local/cs/bin:$PATH`
   3. `source $HOME/.profile`

Note: Make sure OCaml is of version 4.02.1

Introduction to OCaml

1. Getting started:

```
# print_string “Hello, world!\n”;;
```

2. Variable assignment:

```
# let x = 100;;

# x * x * x;;

# let x = 100 in x * x * x;;

# let x = 100 in (let y = x * x in y + x);;
```

Note: Parentheses are optional in OCaml.

Introduction to OCaml

3. Functions:

```
# let cube x = x * x * x;;

# let neg x = x < 0;;

# let neg x = x < 0.0;;

# let isvowel c =
    c = ‘a’ || c = ‘e’ || c = ‘i’ || c = ‘o’ ||
    c = ‘u’;;
```

4. Recursion:

```
# let rec factorial a =
     if a = 0 then 1 else
     a * factorial(a - 1);;

# let rec gcd a b =
     if b = 0 then a else
     gcd b (a mod b);;
```

Introduction to OCaml

5. Pattern Matching:

```
# let rec factorial a =
     match a with
       1 -> 1
     | _ -> a * factorial (a - 1);;

# let isvowel c =
     match c with
       ‘a’ | ‘e’ | ‘i’| ‘o’ | ‘u’ -> true
     | _ -> false;;
```

Introduction to OCaml

6. Lists:

The :: (pronounced cons) operator:

Add a single element to the front of the existing list

```
# let l = [1; 2] in 2 :: l;;
```

```
# true :: [true; false];;
```

The @ (pronounced append) operator:

Combine two existing lists together

```
# [1; 2] @ [3; 4; 5];;
```

Introduction to OCaml

6. Lists:

```
# let isempty l =

    match l with

      [] -> true

    | _ -> false;;
```

```
# let rec length l =

    match l with

      [] -> 0;

    | h::t -> 1 + length (t);;
```

```
# let rec length_helper l n =

    match l with

      [] -> n

    | h::t -> length_helper t (n + 1);;

  let length l = length_helper l 0;;
```

Recursive functions which do not build up a growing intermediate expression are called tail recursion

Introduction to OCaml

6. Lists:

```
# let rec odd_elements l =

    match l with

      [] -> []

    | [a] -> [a]

    | h::_::t -> h::odd_element (t);;
```

```
# let rec odd_elements l =

    match l with

      h::_::t -> h::odd_elements (t)

   | _ -> l;;
```

Introduction to OCaml

6. Lists:

```
# let rec append a b =
    match a with
      [] -> b
    | h::t -> h::append t b;;
# let rec reverse a =
    match a with
      [] -> []
    | h::t -> reverse (t) @ [h];;
```

Introduction to OCaml

6. Lists:

```
# let rec take n l =
    if n = 0 then []
    else
      match l with
        [] -> []
      | h::t -> h::take (n-1) t;;
```

```
# let rec drop n l =
    if n = 0 then l
    else
      match l with
        [] -> []
      | h::t -> drop (n-1) t;;
```

Introduction to OCaml

7. Sorting:

7.1 Insertion sort:

```
# let rec insert a l =

    match l with

      [] -> [a]

   | h::t ->

       if a <= h

       then a::h::t

       else h::insert a t;;
```

```
# let rec insert_sort l =

    match l with

      [] -> []

   | h::t ->

       insert h (sort t);;
```

Introduction to OCaml

7. Sorting:

7.2 Merge sort:

```
# let rec merge x y =
    match x, y with
      [], l -> l
   | l, [] -> l
   | hx::tx, hy::ty ->
       if hx <= hy
       then hx::merge tx (hy::ty)
       else hy::merge (hx::tx) ty;;
```

```
# let rec merge_sort l =
    match l with
      [] -> []
   | [a] -> [a]
   | _ ->
     let len = length l in
       let left = take (len/2) l in
         let right = drop (len/2) l in
           merge (merge_sort left)
                 (merge_sort right);;
```

Introduction to OCaml

8. Functional programming:

```
# let rec map f l =
    match l with
      [] -> []
    | h::t -> f h::map f t;;
# let double x = 2 * x;;
# map double [1; 2; 3;];;
```

Anonymous function

```
let evens =
  map (fun x -> x mod 2 = 0)
      [1; 2; 3];;
```

Introduction to OCaml

9. Dictionary:

```
# let rec lookup x l =
    match l with
      [] -> raise Not_found
    | (k,v)::t ->
     if k = x
     then v
     else lookup x t;;
```

```
# let rec add k v l =
    match l with
      [] -> [(k,v)]
    | (k’,v’)::t ->
     if k’ = k
     then (k,v)::t
     else (k’,v’)::add k v t;;
```

Introduction to OCaml

9. Dictionary:

```
# let rec remove k l =
    match l with
      [] -> []
    | (k',v')::t ->
      if k' = k
      then t
      else (k',v')::remove k t;;
```

```
# let rec key_exists_1 k l =
    try
      let _ = lookup k l in true
    with
      Not_found -> false;;
```

```
# let rec key_exists_2 k l =
    match l with
      [] -> false
    | (k',_)::t ->
      if k' = k
      then true
      else key_exists_2 t;;
```

Introduction to OCaml

10. User-defined types:

```
# type color =
    Red
  | Green
  | Blue
  | Yellow
  | RGB of int * int * int
```

```
# let components c =
    match c with
      Red -> (255, 0, 0)
    | Green -> (0, 255, 0)
    | Blue -> (0, 0, 255)
    | Yellow -> (255, 255, 0)
    | RGB (r, g, b) -> (r, g, b);;
```

1. The new type variable names start with upper case letter.

2. We use keyword `of` in our type constructor
to carry information along with values built with it.

Introduction to OCaml

10. User-defined types:

Types may contain a **type variable** like ‘a to allow the type of part of the new type to vary.

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

We can read this as “a value of type ‘a option is either nothing, or something of type ‘a”.

```
# let rec lookup_opt k l =
  match l with
    [] -> None
  | (k', v)::t -> if k' = k then Some v else lookup_opt k t;;
```

10. User-defined types:

Assume that we want to evaluate the expression 1 + 2 x 3, and we want to evaluate it without parentheses.

```
# type exp =
    Num of int
  | Add of exp * exp
  | Sub of exp * exp
  | Mul of exp * exp
  | Div of exp * exp
  | Mod of exp * exp;;
```

```
# let rec evaluate e =
    match e with
      Num x -> x
    | Add (a, b) -> evaluate a + evaluate b
    | Sub (a, b) -> evaluate a - evaluate b
    | Mul (a, b) -> evaluate a * evaluate b
    | Div (a, b) -> evaluate a / evaluate b
    | Mod (a, b) -> evaluate a mod evaluate b;;
```