CAS CS 320
Lec 9

References (Array) & Lazy

1. Concepts
    a. Functional programming takes advantage of using immutable values
    b. In python,
       ```
       def foo():
           res = []
           def bar():
               res = res.append(1)
       ```
       # Here, res in res.append(1) refers to the res on the bar function, but res is not defined there, implying that an error will occur
       # res is mutable data structure (makes program harder to reason)
    c. In ML, that is not the case (error does not occur and refers to the right variable, res refers to the correct res
2. References
    a. Reference is an array of size 1 (singleton array) (reference is pointer and value)
    b. val A = ref(5)
       val x = !A (get the value of referred value)
       val () = !A:= !A + 1 (update the value of A)
    c.
       ```
       fun fact(x) =
       let
           val res = ref(1)
           val i = ref(0)
           fun loop(): unit =
               if !i < x then
                   i := !i + 1
                   res := !res * !i
                   loop()
               else ()
       in
           loop(); !res
       end
       ```
    d. Two kinds of arrays
        i. Mutable (array)
        ii. Immutable (vector)
    e. val A =
       val x = Array.sub(A,i) (*subscripting*)

val _ = Array.update(A,i,v) (*update i with v *)

f. Effectful programming features → features that cause effect (example → print(x) → it prints value of x which is an effect)
g. Exceptions → one of effectful programming features
h. Pure programming features → (example → 1 + 2 → does not produce any effect to the programmer)
i. Foreach_to_iforeach (iforeach is the index, enumerate in python)
j. fun array_foreach(A: ‘a array, work: ‘a -> unit): unit =
    int1_foreach(Array.length(A), fn i => work(Array.sub(A,i)))
k. fun array_iforeach(A: ‘a array, iwork: (int * ‘a) -> unit): unit =
    foreach_to_iforeach(array_foreach)(A, iwork)

3. Lazy
    a. fun from(n: int): int list =
        n::from(n+1) (* will cause stack overflow → maximum recursion reached *)
    b. fun from(n: int) = int list =>
        n::from(n+1) (* will cause stack overflow → maximum recursion reached *)
        (* when lambda is seen, evaluation stops and it will be a value *) → never go into the body of lambda, use lambda to stop evaluation
    c. datatype ‘a strmcon =
        strmcon_nil
        | strmcon_cons of (‘a * (unit -> ‘a strmcon))

    type ‘a stream = unit -> ‘a strmcon(stream is a function that returns a stream constructor)
    (*stream constructor*) (*stream is a lazy list*)
    d. fun from(n: int) = fn() =>
        strmcon_cons(n,from(n+1))
        (* when lambda is seen, evaluation stops and it will be a value *) → never go into the body of lambda, use lambda to stop evaluation

    OR
    fun from(n: int): int stream = fn() =>
        strmcon_cons(n,from(n+1))
    val theNats = from(0)
    val fxs = theNats
    val strmcon_cons(x0, fxs) = fxs() → x0 is 0
    val strmcon_cons(x1, fxs) = fxs() → x1 is 1
    val strmcon_cons(x2, fxs) = fxs() → x2 is 2
    val strmcon_cons(x3, fxs) = fxs() → x3 is 3
    (*presents an illusion of infinite list*)