CAS CS 320
Lec 6

Higher Order Functions

1. List Comprehension
   a. In Python, we define lists as []
   b. List Comprehension in Python
      i. Square of xs in the list →
         1. [x * x for x in range(10)]
   c. fun list_map(xs: ‘a list, f0: ‘a -> ‘b): ‘b list =
```
(* non tail recursive *)
        case xs of
        nil => nil
        | x1 :: xs => f0(x1): list_map(xs, f0)
```
   d. fun list_sum(xs: int list): int =
      (* sum up all the ints in a given list xs *)
   e. list_sum(fun list_map(xs, fn _ => 1)) → compute the length of the list
   f. fun list_foldl(xs: ‘a list, r0 = ‘r, f0: ‘r * ‘a -> ‘r): ‘r =
      (* basically list sum by using combinators*)
      i. Example
```
        r0 = 0      1     3     6     10    15
        xs = 1      2     3     4     5
        f0 = +

        nil => r0
        | x1 :: xs => list_foldl(xs, f0(r0, x1), f0)
```
   g. list_length(xs) = list_foldl(xs, 0, fn(r,x) => r +1)
   h. fun list_foldr(xs: ‘a list, r0 = ‘r, f0: ‘a * ‘r -> ‘r): ‘r =
      i. Example
```
        r0 = 15     14          12    9     5     0
        xs =        1           2     3     4     5
        f0 = +

        case xs of
        nil => r0
        | x1 :: xs => f0(x1, list_foldr(xs, r0, f0))
```
   i. list_length(xs) = list_foldr(xs, 0, fn(x,r) => 1 + r)
   j. list_append(xs, ys) =
```
        list_foldr(xs, ys, fn(x,r) => x :: r)
```

```
list_rappend(xs, ys) =
        list_foldl(xs, ys, fn(r,x) => x :: r)
```

2. Building on SML library for lists

a.

```
fun list_foldl(xs: ‘a list, r0: ‘r, fopr: ‘r * ‘a -> ‘r): ‘r =
(* high order function → because it takes another function *)
(* more specifically, it is second order function because it takes a first order function *)
        case xs of
        nil => nil
        | x1 :: xs => list_foldl(xs, fopr(r0, x1), fopr)
```

b.

```
fun list_foldr(xs: ‘alist, r0: ‘r, fopr: ‘a * ‘r -> ‘r): ‘r =
        case xs of
        nil => nil
        | x1 :: xs => fopr(x1, list_foldr(xs, r0, fopr))
OR
fun list_foldr(xs: ‘a list, r0: ‘r, fopr: ‘a * ‘r -> ‘r): ‘r =
(* tail recursive version *)
        list_foldl(list_reverse(xs), r0, fn(r,x) => fopr(x,r))
```

c.

```
fun list_extend(xs: ‘a list, x0: ‘a): ‘a list =
        list_append(xs, [x0])
```

d.

```
fun list_append(xs: ‘a list, ys: ‘a list): ‘a list =
        list_foldr(xs, ys, fn(x,r) => x :: r)
```

e.

```
fun list_length (xs: ‘a list): int =
        list_foldl(xs,0,fn(r,x) => r + 1)
```

f.

```
fun list_rappend(xs: ‘a list, ys: ‘a list): ‘a list =
        list_foldl(xs, ys, fn(r,x) => x :: r)
```

g.

```
fun list_map(xs: ‘apple list, fopr: ‘apple -> ‘banana): ‘banana list =
        list_foldr(xs, [], fn(x, r) => fopr(x) :: r)
```

3. Building on SML library for strings

a.

```
fun string_foldl (xs: string, r0: ‘r, fopr: ‘r * char -> ‘r): ‘r =
        let
                val ln = String.size(xs)
                fun loop(i0: int, r0: ‘r): ‘r =
                        if i0 < ln then loop(i0 + 1, fopr(String.sub(xs,i0), r0))
                        else r0
        in
                loop(0, r0)
        end
```

   b. fun string_length(xs) =
      string_foldl(xs, 0, fn(r,x) => r + 1)

4. SML library Trees
   a. fun tree_max_height(xs: ‘a tree): int =
      case xs of
      tree_nil => 0
      | tree_cons(tl, _, tr) => 1 + int_max(tree_max_height(tl), tree_max_height(tr))
   b. fun tree_min_height(xs: ‘a tree): int =
      case xs of
      tree_nil => 0
      | tree_cons(t1, _, tr) => 1 + int_min(tree_min_height(tl), tree_min_height(tr))
   c. fun tree_fold(xs: ‘a tree, r0, fopr: (‘r * ‘a * ‘r) -> ‘r): ‘r =
      case xs of
      tree_nil => r0
      | tree_cons(tl, x0, tr) => fopr(tree_fold(t1, r0, fopr), x0, tree_fold(tr, r0, fopr))
   d. fun tree_size (xs: ‘a tree): int =
      tree_fold(xs, 0, fn(tl, _, tr) => tl + 1 + tr)
   e. fun tree_height(xs: ‘a tree): int =
      tree_fold(xs, 0, fn(t1, _, tr) => 1 + int_max(t1,tr))
   f. fun tree_sizeheight(xs: ‘a tree): int * int =
      tree_fold(xs, (0,0), fn(t1, _, tr) =>
         (#1(tl) + 1 + #1(tr), 1 + int_max(#2(tl), #2(tr))))

5. Python
   a. xs = (x * x for x in range(10)) → generator object
   b. xs = [x * x for x in range(10)] → list comprehension
   c. xs = tuple(x * x for x in range(10)) → tuple
   d. Translation of syntax
      list_map(list_fromto(0,10), fn x => x * x)
   e. xs = [(i, x * x) for (i,x) in enumerate(range(10))] → gives index and the value x by using enumerate