CAS CS 320
Lec 5

Datatypes & Tree & Sort

1. List Processing Functions

   a. The database is (“a”, 0), (“b”, 1), (“c”,2), …

   b. exception NotFound

   c.
   ```
   fun dbase_search (dbase: (string * ‘a) list, key: string): ‘a = (* tail recursive function *)
   case dbase of
           nil => raise NotFound
           | kx1 :: dbase =>
                   if key = #1(kx1) then #2(kx1) else dbase_search(dbase,key)
   ```

   d. Testing the function
   ```
   val mydbase = [(“a”,0), (“b”,1), (“c”,2)]
   val x0 = dbase_search(mydbase, “a”)
   val x1 = dbase_search(mydbase, “b”)
   val x2 = dbase_search(mydbase, “c”)
   val x3 = dbase_search(mydbase, “d”) handle NotFound => ~1
   ```

   e. Insertion sort takes O(n^2) run-time

   f.
   ```
   fun insertion_sort(xs: intt list): int list =
   (* monomorphic function → only works for integers *)
   (
           case xs of
                   nil => nil
                   | x1::xs => insert_order(x1, insertion_sort(xs))
   )
   and insert_order(x1: int, xs: int list): int list =
           case xs of
                   nil => [x1]
                   | x2::xs2 =>
                           if x1 <= x2 then x1 :: xs else x2 :: insert_order(x1,xs2)
   ```

   g. Testing the function
   ```
   val xs = [1,3,5,2,4,0]
   val ys = insertion_sort(xs)
   ```

   h. Sorting the types of generic items (not necessarily integers) → need to create another function (called high order function)

   i.
   ```
   fun insertion_sort
           (lte: ‘a * ‘a -> bool) (xs: ‘a list): ‘a list =
   ```

```
        (
        case xs of
        nil => nil
        | x1 :: xs => insert_order lte(x1, insertion_sort lte(xs))
        )
and insert_order
    (lte: ‘a * ‘a -> bool) (x1: ‘a, xs: ‘a list): ‘a list =
        (
        case xs of
        nil => [x1]
        | x2::xs2 =>
            if lte(x1,x2) then x1 :: xs else x2 :: insert_order lte (x1,xs2)
```

j. Testing

```
val xs = [1,3,5,0,2,4]
val ys = insertion_sort(fn(x,y) => x <= y) (xs)
val zs = insertion_sort(fn(x,y) => x >= y) (xs)
(* reverse order *)
val evenodds = insertion_sort(fn(x,y) => (x mod 2) <= (y mod 2)) (xs)
(* even numbers come out first *)
val oddevens = insertion_sort(fn(x,y) => (x mod 2) >= (y mod 2)) (xs)
```

k. Stable sorting algorithm: do not change the order of the values that are equal (insertion sort)

l. Unstable sorting algorithm: change the order of the values even though they are equal (quick sort)

2. Tree

a.

```
datatype ‘a tree =
tree_nil (* empty *)
| tree_cons of ‘a tree * ‘a * ‘a tree (* this is a binary tree *)
```

b.

```
fun tree_size tree_nil = 0
| tree_size (tree_cons(tl, _, tr)) = tree_size(tl) + 1 + tree_size(tr)
(*clausal form *)
```

OR

```
fun tree_size(xs: ‘a tree): int =
case xs of
tree_nil => 0
| tree_cons(tl, _, tr) => tree_size(tl) + 1 + tree_size(tr)
```

c.

```
fun tree_height (xs: ‘a tree): int =
    case xs of
    tree_nil => 0
```

```
        | tree_cons(t1,_,tr) => 1 + int_max(tree_height(tl), tree_height(tr))
```

d. Testing

```
val xs = tree_nil
val xs = tree_cons(xs, 1, xs) (* This is a Leaf *)
val xs = tree_cons(xs, 2, xs)
val xs = tree_cons(xs, 3, xs)
val xs3 = xs
val xs = tree_cons(xs, 4, xs)
val xs4 = xs
val xs = tree_cons(xs, 5, xs)
val xs = tree_cons(xs, 6, xs)
val xs = tree_cons(xs, 7, xs)
val xs = tree_cons(xs, 8, xs)
val xs = tree_cons(xs, 9, xs) (* This is the root node *)
val size_of_xs = tree_size(xs)
(* it is 2^9 - 1 *) (* it is 511 *)
val height_of_xs = tree_height(xs)
(* it is 9 *) (* size = 2^height -1 *)
```

e.

```
fun tree_flatten(xs: ‘a tree): ‘a list =
        case xs of
        tree_nil => []
        | tree_cons(xs1, x0, xs2) => tree_flatten(xs1) @ [x0] @ tree_flatten(xs2)
```

f. Testing

```
val xs3 = tree_flatten(xs3)
(* result is [1,2,1,3,1,2,1] *)
```

g.

```
fun tree_reverse(xs: ‘a tree): ‘a tree =
case xs of
        tree_nil => tree_nil
        | tree_cons(xs1, x0, xs2) => tree_cons(tree_reverse xs2, x0, tree_reverse
         xs1)
```

h. Testing

```
val xs3_reverse = tree_reverse(xs3)
```

3. ylist (tree)

a.

```
datatype ‘a ylist =
        ylist_nil
        | ylist_cons of ‘a * ‘a ylist (* put at the front of the list *)
        | ylist_snoc of ‘a ylist * ‘a (* put at the end of list *)
```

b.

```
fun ylist_length(ys: ‘a ylist): int =
        case ys of
                ylist_nil => 0
        | ylist_cons(_, ys) => 1 + ylist_length(ys)
        | ylist_snoc(ys, _) => ylist_length(ys) + 1
```

   c. fun ylist_last(ys: ‘a ylist): ‘a =

      ```
      case ys of
              ylist_nil => raise Empty
              | ylist_snoc(ys, y1) => y1
              | ylist_cons(y1, ys) => ???
      ```

4. Back to function
   a. fun list_nth(xs: ‘a list, n: int): ‘a =

      ```
      case xs of
              nil => raise Subscript
              | x1::xs => if n <=0 then x1 else list_nth(xs, n-1)
      ```

5. Quicksort → faster than insertion sort
   a. fun list_quicksort(lte: ‘a * ‘a -> bool) (xs: ‘a list): ‘a list =

      ```
              case xs of
                      nil => nil
                      | x1 :: xs => list_partition lte (x1, xs, [], [])
      and
      list_partition (lte: ‘a * ‘a -> bool)
      (p0: ‘a, xs: ‘a list, ys: ‘a list, zs: ‘a list): ‘a list =
              (
              case xs of
              nil => (list_quicksort lte ys) @ (p0 :: (list_quicksort lte zs))
              | x1 :: xs =>
                      if lte(x1, p0)
                      then list_partition lte (p0, xs, x1::ys, zs)
                      else list_partition lte (p0, xs, ys, x1::zs)
      ```