CAS CS 320
Lec 4

Polymorphic datatype && Python Version of List

1. Polymorphic datatype
    a. datatype ‘a list = nil | cons of ‘a * ‘a list (* it is a tuple*)
    (* it is a recursive call of lists *)
    b. Functional programming is about building new object (list)
    c. fun list_length (xs: ‘a list): int = (* the tail recursive version of finding length *)
    (* same runtime but uses different space, does not require lot of space *)

```
let
        fun loop (xs: ‘a list, res: int): int =
        case xs of
                nil => res
                | _ :: xs => loop(xs, res + 1)
in
        loop(xs, 0)
end
```

OR

```
fun list_length (xs: ‘a list): int =
(* requires lots of space → useless library function*)
        (
        case xs of
                nil => 0 |
                _ :: xs => 1 + list_length(xs)
        )
```

    d. fun list_head (xs: ‘a list): ‘a =

```
        (
        case xs of
                nil => raise Empty |
                x1 :: _ => x1
        )
```

OR

e.
```
fun list_headopt (xs: ‘a list): ‘a optn=
        (
        case xs of
                nil => None|
                x1 :: _ => SOME(x1)
        )
```
f.
```
fun list_tail (xs: ‘a list): ‘a =
        (
        case xs of
                nil => raise Empty |
                _ :: xs => xs
        )
```

OR

```
fun list_tailopt (xs: ‘a list): ‘a optn =
        (
        case xs of
                nil => None |
                _ :: xs => SOME(xs)
        )
```
g.
```
fun list_last(xs: ‘a list): ‘a =
        case xs of
                nil => raise Empty |
                x1 :: xs =>
                        (
                        case xs of
                                nil => x1 |
                                _ => list_last(xs)
                        )
```

OR

```
fun list_last(xs: ‘a list): ‘a =
let
        fun loop(x1, xs) =
        (
        case xs of
                nil => x1 |
                x2 :: xs => loop(x2, xs)
```

```
        )
    in
        case xs of
            nil => raise Empty |
            x1 :: xs => loop (x1, xs)
    end
```

h.
```
fun list_append(xs: ‘a list, ys: ‘a list): ‘a list =
        (
        case xs of
            nil => ys |
            x1 :: xs => x1 :: list_append(xs, ys)
        )
```

i.
```
fun list_reverse(xs: ‘a list): ‘a list =
(* runtime is O(n^2) *)
        (
        case xs of
        nil => nil |
        x1 :: xs => list_reverse(xs) @ [x1]
 (* @ is a concatenation operator and is used to concatenate two lists together*)
        )
```

j.
```
fun list_fromto (start: int, finish: int): int list =
        if start < finish then start :: list_fromto(start + 1, finish) else []
(* similar to range in python *)
```

k.
```
fun list_rappend(xs: ‘a list, ys: ‘a list): ‘a list =
        (
        case xs of
        nil => ys |
        x1 :: xs => list_rappend(xs, x1::ys)
        )
```

l.
```
fun list_reverse (xs: ‘a list): ‘a list = list_rappend(xs, [])
(* reversing items one by one *) (* runtime is O(n) *)
```

m.
```
fun list_append(xs: ‘a list, ys: ‘a list) ‘a list =
        list_rappend(list_reverse(xs),ys)
(*This is an inefficient tail-recursive implementation of list_append as it traverses
the first argument twice *)
```

2. Python (Datalist in python)

a.
```
#datatype ‘a list = nil | cons of (‘a * ‘a list)
class fnlist:
        ctag = -1
Def get_ctag(self):
```

```
        return self.ctag
    def __iter__(self):
        return fnlist_iter(self)
```

b. Each has a sub class (0 indicate it is nil), (1 indicates it is not nil and there are elements)

c.
```
def _iter__(self):
        Return self
    def __next__(self):
        if (self.itms.ctag == 0):
            raise StopIteration
        else:
            itm1 = self.itms.cons1
            self.itms = self.itms.cons2
            return itm1
```

d.
```
class fnlist_nil (fnlist):
        def __init__(self):
            self.ctag = 0
            return None
```

e.
```
class fnlist_cons(fnlist):
        def __init__ (self, cons1, cons2):
            self.ctag = 1
            self.cons1 = cons1
            self.cons2 = cons2
            return None
        def get_cons1(self):
            return self.cons1
        def get_cons2(self):
            return self.cons2
    #end of class fnlist_cons
```

f.
```
def fnlist_length(xs):
        res = 0
        while xs.ctag > 0:
            res += 1
            xs = xs.cons2
        return res
```

g.
```
def fnlist_fromto(start, finish):
        if start < finish:
            return fnlist_cons(start, fnlist_fromto(start+1,finish))
        else:
            return fnlist_nil()
```

OR

```
def fnlist_fromto(start,finish):
        res =fnlist_nil()
        while start < finish:
                res = fnlist_cons(finish-1, res)
                finish --
```