Programming 2

Tutorial 8

User-Defined Types 2

Nedim Šišić

2023

Exercise 1

a) Create a type *boolEx* that represents a Boolean expression and supports the operations And, Or, and Not and values True and False.

   (note: parentheses not needed, execution order will be specified by the order of using the type's constructors)

b) Define a function *eval* that takes a *boolEx* parameter and evaluates it to a bool.

Exercise 1

a)
```
type boolEx =
  | Value of bool
  | And of boolEx * boolEx
  | Or of boolEx * boolEx
  | Not of boolEx ;;
```

b)
```
let rec eval be =
  match be with
   | Value b -> b
   | And (be1, be2) -> (eval be1) && (eval be2)
   | Or (be1, be2) -> (eval be1) || (eval be2)
   | Not be1 -> not (eval be1);;
```

Exercise 3 – Run length encoding

WWWWWWWWWWWWBWWWWWWWWWWWWBBBWWWWWWWWW
WWWWWWWWWWWWBWWWWWWWWWWWWBWWWWWWWWW

In run-length encoding:

12W 1B 12W 3B 24W 1B 14W

Exercise 3

Given the type *‘a rle*:

```
type 'a rle =
| Many of int * 'a;;
```

write a function *encode* that given a list of elements encodes it as a list of *rle* values (according to run-length encoding).

Exercise 3

```
let encode l =
  let rec encode2 l count =
    match l with
    | [] -> []
    | h1::(h2::_ as t) when h1 = h2 -> encode2 t (count + 1)
    | h::t -> (Many (count, h)) :: (encode2 t 1)
  in encode2 l 1;;

encode ['a';'a';'a';'a';'b';'c';'c';'a';'a';'d';'d';'e';'e';'e';'e'];;
```

Mutual recursion

Previously: sum type intList:

```
type intList =
| IList of int * IList
| Empty;;
```

Using both a sum and a record (product) type and **mutual recursion**:

```
type sumIntList =
| Empty
| Node of recordNode
and recordNode = { value : int; tail : sumIntList};;

let nodeVal = {value = 5; tail = Empty};;
let intListVal = Node {value = 1; tail = Node {value = 2; tail = Empty}};;
```

Exercise 4

Create a type *bTree* that represents a parameterized binary tree,
and uses a parameterized record type *Node* to represent its nodes.

The tree's node contains a value of type *'a* and two children nodes: the left subtree and the right subtree.

```
type 'a bTree =
| Empty
| Node of 'a node
and
'a node = {value : int; lTree : 'a bTree; rTree : 'a bTree};;
```

Exercise 5

A parameterized tree such that.:

- its nodes contain values of types ‘a or ‘b
- each node has multiple children nodes stored in a list

is declared by:

```
type ('a, 'b) tree =
| Nil
| NodeA of ‘a * ('a, 'b) tree list
| NodeB of ‘b * ('a, 'b) tree list;;
```

Write a function *split* that inserts all values of type *‘a* into the first and all values of type *‘b* into the second list of the returned pair:

```
split : ('a,'b) tree -> 'a list * 'b list
```

Exercise 5

```
let nA1 = NodeA (5, [Nil; Nil]);;
let nB1 = NodeB ('a', [Nil]);;
let nA2 = NodeA (1, [nA1; Nil; nB1]);;
let nB2 = NodeB ('b', [Nil]);;
let root = NodeA (3, [nA2; nB2]);;

(* helper function applySplit: apply f on each element of list and merge
results; applySplit and f should both return  two lists (just like split) *)
let rec applySplit treeList f =
  match treeList with
  | [] -> [], []
  | h::t -> (fst (f h)) @ fst (applySplit t f), (snd (f h)) @ (snd (applySplit t f));;
```

Exercise 5

```
let rec split t =
  match t with
  | Nil -> [], []
  | NodeA (e, l) ->
    e :: (fst (applySplit l split)), snd (applySplit l split)
  | NodeB (e, l) ->
    fst (applySplit l split), e :: (snd (applySplit l split));;

split root;;
```