KSFUPR01KU **IT University, F2021**

Written exam, Functional Programming
Thursday May 27, 2021

Version 1.00 of May 25, 2021

These exam questions comprise 8 pages. Check immediately that you have all the pages.

The exam duration is 4 hours.

There are 4 questions. To obtain full marks you must answer all the subquestions satisfactorily.

You are allowed to use books, lecture notes, lecture slides, hand–ins, solutions to assignments, calculators, computers, software, on-line resources etc. during the examination. This includes any form of device that can execute programs written in F#.

You are allowed to use the .NET library including the modules described in the book, e.g., List, Set, Map etc.

If a subquestion requires you to define a particular function, then you may **use that function in subsequent subquestions**, even if you have not managed to define it yourself.

If a subquestion requires you to define a particular function, then you may **define as many helper functions as you want**, but in any case you must define the required function so that it has exactly the type and effect that the subquestion asks for.

The grading will favour functional solutions, i.e., solutions without side effects. Recursion is also favoured over loops. An imperative solution is of course preferred over no solution.

You should hand–in one file only, e.g., `ksfupr2021.<fsx,pdf>`. Do not use time on formatting your solution in Word or PDF.

You are welcome to use the accompanying file `may2021Snippets.fsx`. The file contains some of the code snippets included in the exam set for your convenience to copy into your solution.

You MUST include explanations and comments to support your solutions. You simply write them as comments around your code.

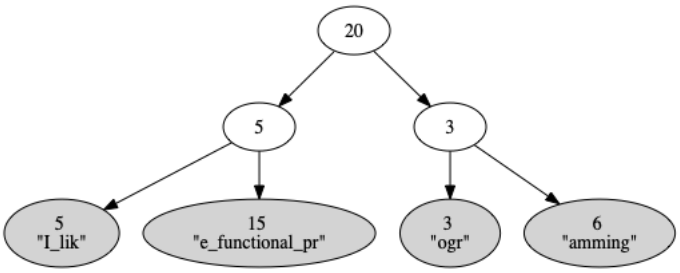
Your exam hand–in must be made by yourself and yourself only, and this holds for program code, examples, the explanations you provide for the code, and all other parts of the answers. It is illegal to make the exam answers as group work or to enlist the help of others in any way.

Your hand–in must contain the following declaration:

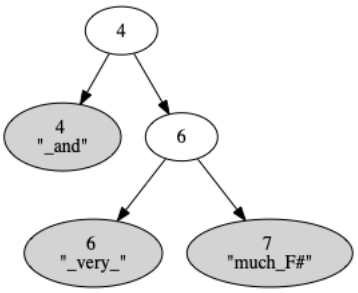
I hereby declare that I myself have created this exam hand–in in its entirety without help from anybody else.

Question 1 (25%)

The concept of finite trees is described in HR Chapter 6. We define a *rope* as a tree with nodes and leaf nodes. A node has two children and edges connect parent nodes to its children.
A rope is used to represent long strings by storing smaller sub–strings in the leaf nodes. Concatenating the sub–strings gives the longer strings. The data structure makes it possible to add and remove sub–strings by adding and removing nodes in the tree.
The example ropes **rope1** and **rope2** below show how the longer strings "I_like_functional_programming" and "_and_very_much_F#" can be represented as a rope.



Example rope1: A rope with four sub-strings.



Example rope2: A rope with three sub-strings.

Each node in the tree has a *weight*. The *weight* of a leaf node is the length of its sub-string. For instance, the weight of leaf node with sub–string "I_lik" is 5. The *weight* of a non–leaf node is the sum of the weights of all the leaf nodes its left subtree. For instance, the weight of the root node in **rope1** is 20 because the sum of the weights of leaf nodes in its left subtree is 5 ("I_lik") plus 15 ("e_functional_pr").
We represent a rope with below datatype

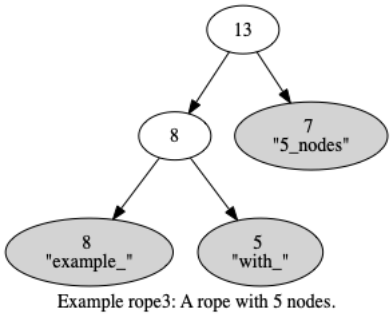
```
type Rope =
    Leaf of string * int
  | Node of Rope * int * Rope
```

The example **rope1** is represented by the below value `rope1`

```
let rope1 = Node(Node(Leaf("I_lik",5),5,Leaf("e_functional_pr",15)),
                 20,
                 Node(Leaf("ogr",3),3,Leaf("amming",6)))
```

Question 1.1

- Declare value `rope2` for the example above representing the string "_and_very_much_F#". The value must have the nodes and leaf nodes as shown in the example.
- Explain whether the type `Rope` is monomorphic or polymorphic.
- Declare a value `rope3` representing the string "example_with_5_nodes" corresponding to the tree at right.



Example rope3: A rope with 5 nodes.

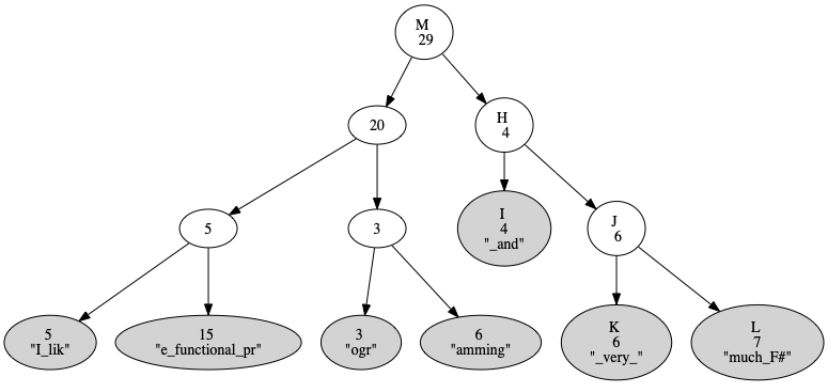
Question 1.2

- Declare an F# function `length` r of type `Rope -> int` that returns the length of the string represented by the rope r. For instance, `length rope1` returns 29.

- Declare an F# function `flatten` r of type `Rope -> string` that returns an F# string as the concatenation of all the sub–strings in the rope r. For instance, `flatten rope1` returns `"I_like_functional_programming"`.
- Declare an F# function `maxDepth` r of type `Rope -> int` that returns the maximum depth of the tree. For instance `maxDepth rope1` returns 3. The maximum depths of `rope2` and `rope3` are also 3.
- Declare an F# function `index` i r of type `int -> Rope -> char` that returns the character at index i in the string represented by rope r. The first character has index 0. The function fails, with an appropriate message, if index is outside the string (`failwith`). For instance, `index 5 rope1` returns the character `'e'`. Make at least three test cases testing the index function.

Question 1.3

- The purpose of a rope data structure is to be able to add and delete sub–strings efficiently. An example operation is concatenating two ropes r_1 and r_2 into a single rope. This can be done by creating a new parent node with r_1 as left node and r_2 as right node. The weight of the parant node is the length of r_1. The example below shows the concatenation of `rope1` and `rope2`.



Example rope4: Concatenating ropes rope1 and rope2.

Declare an F# function `concat` r_1 r_2 of type `Rope -> Rope -> Rope` that concatenates the two ropes r_1 and r_2 as described above.

- Declare an F# function `prettyPrint` r of type `Rope -> string` that pretty prints the internal representation of r. For instance, `prettyPrint rope1` may produce the output below. The readability of your layout is important, but you do not have to make it exactly as shown below.

```
Node(
     Node(
          Leaf(I_lik,5),
          5,
          Leaf(e_functional_pr,15)),
     20,
     Node(
          Leaf(ogr,3),
          3,
          Leaf(amming,6)))
```

Question 2 (30%)

The concept of lists is described in HR Chapter 5. We define an *unrolled list* as a variation of an F# list where each F# list element can store multiple values in the unrolled list. Consider the F# list value

```
let list01 = ['A'; 'B'; 'C'; 'D'; 'E'; 'F'; 'G'; 'H'; 'I'; 'J'; 'K'; 'L'; 'M';
              'N'; 'O'; 'P'; 'Q'; 'R'; 'S'; 'T'; 'U'; 'V'; 'W'; 'X'; 'Y'; 'Z']
```

The idea is to pack several elements into the same *bucket* and then represent the unrolled list as a list of buckets. Besides the elements in a bucket, we also keep track of the current number of elements in the bucket, *sizeBucket*. This leads to the following F# representation of unrolled lists.

```
type Bucket<'a> = {
  sizeBucket : int;
  elems : List<'a>
}

type UList<'a> = Bucket<'a> list
```

The elements in the unrolled list are ordered similar to elements in an F# list and first element has index 0. One possible representation of `list01`, as an unrolled list, maintaining the same order of elements, is

```
let ulist01 = [ { sizeBucket = 4; elems = ['A';'B';'C';'D'] };
                { sizeBucket = 2; elems = ['E';'F'] };
                { sizeBucket = 6; elems = ['G';'H'; 'I'; 'J'; 'K'; 'L'] };
                { sizeBucket = 6; elems = ['M'; 'N'; 'O'; 'P'; 'Q'; 'R'] };
                { sizeBucket = 4; elems = ['S'; 'T'; 'U'; 'V'] };
                { sizeBucket = 4; elems = ['W'; 'X'; 'Y'; 'Z'] } ]
```

Question 2.1

- Explain why the type of `ulist01` is `Bucket<char> list`.
- The representation of elements in an unrolled list is not unique. Declare a value, `ulist02` of type `Bucket<char> list`, that represents another valid unrolled list representation of the elements in `list01`.
- Explain why the expression `ulist02 = ulist01` returns false.
- Declare an F# function `emptyUL()` of type `unit -> UList<'a>` that returns an empty unrolled list.

Question 2.2

- Declare an F# function `sizeUL` *ul* of type `Bucket<'a> list -> int` that returns the number of elements in the unrolled list *ul*. For instance, `sizeUL ulist01` returns 26 and `sizeUL (emptyUL())` returns 0.
- Declare an F# function `isEmptyUL` *ul* of type `Bucket<'a> list -> bool` that returns true if the unrolled list has no elements and false otherwise. For instance, `isEmptyUL (emptyUL())` returns true and `isEmptyUL ulist01` returns false.

- Declare an F# function `existsUL` *e ul* of type `'a -> Bucket<'a> list -> bool when 'a:equality`, that returns true, if *e* exists in unrolled list *ul*; otherwise returns false. For instance, `existsUL 'A' (emptyUL())` returns false and `existsUL 'A' ulist01` returns true.
- Declare an F# function `itemUL` *ul i* of type `Bucket<'a> list -> int -> 'a` that returns the element at index *i* in the unrolled list *ul*. The function fails if index is outside the list (`failwith`). First element has index 0. For instance, `itemUL ulist01 5` returns `'F'`.
- Declare an F# function `filterUL` *p ul* of type `('a->bool) -> Bucket<'a> list -> Bucket<'a> list` that filters out all elements e in *ul* where the predicate *p* e returns false. For instance, `filterUL (fun e -> e < 'I') ulist01` may return the F# value

```
[{sizeBucket = 4;elems = ['A';'B';'C';'D'];};
 {sizeBucket = 2;elems = ['E'; 'F'];};
 {sizeBucket = 2;elems = ['G'; 'H'];}]
```

Question 2.3

- We now define three properties for our representation of unrolled lists:
  1. The maximum allowed number of elements in a bucket, *maxSizeBucket*, is 4.
  2. We will not allow empty buckets in our representation.
  3. The size of a bucket, `sizeBucket`, must be equal to the number of elements in the bucket (`elems`).

  Below representation of the F# list `[1;2;3;4;5;6]` breaks all three properties above.

```
let ulist03Wrong = [ { sizeBucket = 2; elems = [1] };
                     { sizeBucket = 0; elems = [] };
                     { sizeBucket = 5; elems = [2;3;4;5;6] } ]
```

  Declare an F# function `chkUL` *ul* of type `Bucket<'a> list -> bool` that returns true if all properties above are fulfilled for the unrolled list *ul*; otherwise returns false. For instance, `chkUL ulist03Wrong` returns false and `chkUL ulist01` returns true.

- Declare an F# function `map` *f ul* of type `('a->'b) -> Bucket<'a> list -> Bucket<'b> list` that returns the unrolled list resulting from applying the function f on all elements in the unrolled list *ul*. For instance, `map (int) ulist01` returns the unrolled list with ASCII values for each character in `ulist01`.

```
[{sizeBucket = 4;elems = [65; 66; 67; 68];};
 {sizeBucket = 2;elems = [69; 70];};
 {sizeBucket = 6;elems = [71; 72; 73; 74; 75; 76];};
 {sizeBucket = 6;elems = [77; 78; 79; 80; 81; 82];};
 {sizeBucket = 4;elems = [83; 84; 85; 86];};
 {sizeBucket = 4;elems = [87; 88; 89; 90];}]
```

- Declare an F# function `fold` *f a ul* of type `('a->'b->'a) -> 'a -> Bucket<'b> list -> 'a` that folds the function f over all entries in the unrolled list *ul* starting with initial accumulator a. For instance, `fold (fun a c -> a+((string)c)) "" ulist01` returns the value

```
"ABCDEFGHIJKLMNOPQRSTUVWXYZ"
```

Question 3 (20%)

Question 3.1

A recursion formula G of two variables, *m* and *n*, is defined as

$$G(m,n) = \begin{cases} n+m & \text{if } n \leq 0 \\ G(2*m, n-1)+m & \text{if } n > 0 \end{cases}$$

where m and n are integers.

- Declare an F# function `G` (m,n) of type `int*int -> int` that implements the recursion formula above. For instance, `G(10,10)` returns the value 20470.
- Explain whether your implementation of `G` above is tail–recursive.
- Implement a tail-recursive version `GA` of `G`, in case `G` is not already tail-recursive. The function `GA` must have the same function type as `G`.

Question 3.2

The concept of sequences is described in HR Chapter 11.

- Declare a finite F# sequence `mySeq` of type `seq<int*int>`, that computes the sequence of pairs

```
(1,1),   (1,2), ...,   (1,100),
(2,1),   (2,2), ...,   (2,100),
...,
(100,1), (100,2), ..., (100,100)
```

  For instance, `Seq.take 4 mySeq` returns the value

```
seq [(1, 1); (1, 2); (1, 3); (1, 4)]
```

- Declare a finite F# sequence `gSeq` of type `seq<int>`, defined by applying the function `G` (or `GA`) on each element in the sequence `mySeq` defined above. For instance, `Seq.take 4 gSeq` returns the value

```
seq [3; 7; 15; 31]
```

  because `G(1,1)` is 3, `G(1,2)` is 7 etc.

Question 4 (25%)

We consider a simple conditional calculator *CondCalc*. The calculator has instructions for addition, subtraction, constants, labels and a conditional jumps. The *instruction set*, `inst`, is defined below

```
type stack = int list
type inst =
    ADD
  | SUB
  | PUSH of int
  | LABEL of string
  | IFNZGOTO of string
  | EXIT
```

```
let insts01 =
  [PUSH 10;
   PUSH 12;
   ADD;
   EXIT]
```

```
let insts02 =
  [PUSH 10;
   LABEL "sub1";
   PUSH 1;
   SUB;
   IFNZGOTO "sub1";
   EXIT]
```

The instructions work on a *stack machine* represented as a list of integers, see type `stack` above. The execution of `ADD` on a stack $v_1\ v_2\ v_3\ \cdots$ yields the stack $(v_1 + v_2)\ v_3\ \cdots$, i.e., the top two values v_1 and v_2 are popped and the result $v_1 + v_2$ pushed on top. Similar for `SUB`, on a stack $v_1\ v_2\ v_3\ \cdots$ yields the stack $(v_2 - v_1)\ v_3\ \cdots$. The execution of `PUSH` *i*, on a stack $v\ \cdots$, yields the stack $i\ v\ \cdots$. The `EXIT` instruction stops program execution, and returns the top value from the stack, as the result of the exection. The instruction `LABEL` *lab* defines a label, representing the instruction following the `LABEL` instruction. The execution of instruction `IFNZGOTO` *lab* on a stack $v\ \cdots$ tests the value v, and if not zero, execution continues at the instruction following the label *lab*. If value v is zero, execution continues at the instruction following the `IFNZGOTO` instruction. The stack is left unchanged, i.e., $v\ \cdots$. The example `insts01` represents a program that adds the constants 10 and 12 and then exits with result 22. The example `insts02` pushes the constant 10 on the stack and then executes a loop, where it, in each iteration, subtracts 1 from the constant pushed initially. When the subtraction results in 0, then `IFNZGOTO "sub1"` is false, and program exits with result 0.

Question 4.1

- Declare an F# function `execInsts` *insts* of type `inst list -> int` that executes the instructions `insts` as defined above on an empty stack. We do not consider the instructions `LABEL` and `IFNZGOTO` in this question and the function fails (`failwith`) trying to execute those, or if values are missing on the stack. For instance, `execInsts insts01` returns the value 22 and `execInsts insts02` fails.

  Hint: This question is similar to exercise HR 6.8. You may use the template below

```
let execInsts insts =
  let rec exec insts s =
    match (insts,s) with
    | (SUB::is,v1::v2::s) -> exec is (v2-v1::s)
    | ...
    | (LABEL lab::_,s) -> failwith "LABEL not implemented"
    | (IFNZGOTO lab::_,s) -> failwith "IFNZGOTO not implemented"
    | _ -> failwith "Missing stack values for instruction"
  exec insts []
```

Question 4.2

In order to execute programs with instructions `LABEL` and `IFNZGOTO` we need to know what instruction corresponds to a given label. To achieve this, we transform the list of instructions into a map, mapping

instruction indices to instructions, see type `prog` below. The first instruction has index 0. We also remove the `LABEL` instruction and replace the reference in the `IFNZGOTO` instruction with the index of the instruction to jump to. Below shows the result of this transformation on the two example programs `insts01` and `insts02`. As we are changing the definition of the `IFNZGOTO` instruction we define a new instruction set, type `resolvedInst`.

(insts01)

Index	Instruction
0	`RPUSH 10`
1	`RPUSH 12`
2	`RADD`
3	`REXIT`

(insts02)

Index	Instruction
0	`RPUSH 10`
1	`RPUSH 1`
2	`RSUB`
3	`RIFNZGOTO 1`
4	`REXIT`

New instruction set

```
type resolvedInst =
    RADD
  | RSUB
  | RPUSH of int
  | RIFNZGOTO of int
  | REXIT
type prog = Map<int,resolvedInst>
```

- Declare an F# function `buildEnv` *insts* of type `inst list -> Map<string,int>`, that returns a *label environment*, *env*, mapping labels to the index of the label instruction holding the label. For instance, `buildEnv insts01` returns the empty map, `map []` and `buildEnv insts02` returns the map, `map [("sub1", 1)]`.

  Hint: All instructions, except `LABEL`, do not change the env, `env`, but increase the index, `idx` with 1. The `LABEL` instruction does not increase the index, as it is removed later. Template below.

```
let buildEnv insts =
  let rec build idx env = function
    [] -> env
  | LABEL lab :: insts -> build idx ...
  | _ :: insts -> build (idx+1) ...
  build 0 Map.empty insts
```

- Declare an F# function `resolveInsts` *insts env* of type `inst list -> Map<string,int> -> Map<int,resolvedInst>` that transforms the instructions *insts*, with label environment *env*, into a map, mapping indices to the new instructions, where label references are replaced with indices. For instance, `resolveInsts insts02 (buildEnv insts02)` returns

```
map [(0, RPUSH 10); (1, RPUSH 1); (2, RSUB); (3, RIFNZGOTO 1); (4, REXIT)]
```

  Hint: You can use the template below. Notice the use of `lookup`, to lookup a label in the label environment.

```
type env = Map<string,int>
let lookup l m =
  match Map.tryFind l m with
    None -> failwith "Value not in map"
  | Some v -> v

let resolveInsts insts env =
  let rec resolve idx = function
    [] -> Map.empty
  | LABEL lab :: insts -> resolve idx insts
  | ADD :: insts -> Map.add idx RADD (resolve (idx+1) insts)
  | IFNZGOTO lab :: insts -> Map.add idx (RIFNZGOTO (lookup ...)) (resolve ...)
  | ...
  resolve 0 insts
```