BSWU BFNP–1413003U **IT University, F2018**

Written exam, Functional Programming
Thursday May 31, 2018

Version 1.00 of 2018-05-28

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 ASCII file only, e.g., `bfnp2018.fsx`. Do not use time on formatting your solution in Word or PDF.

You are welcome to download the accompanying file `bfnp2018Snippets.fsx` from the course homepage in LearnIT: `https://learnit.itu.dk/course/view.php?id=3017414`. 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.

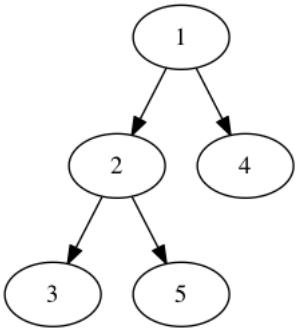
Your exam hand-in must be made by yourself and yourself only, and this holds for program code, examples, the explanation you provide for the code, and all other parts of the answers. It is illegal to make the exam answers as group work or to enlist the help of others in any way.

Your hand-in must contain the following declaration:

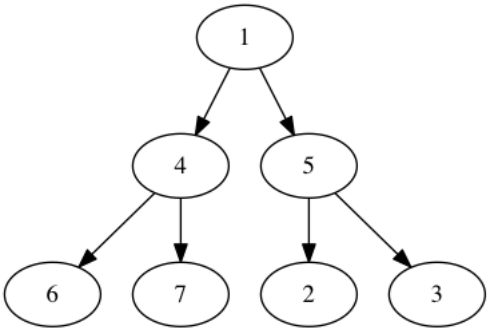
I hereby declare that I myself have created this exam hand-in in its entirety without help from anybody else.

Question 1 (25%)

We define a *heap* as a binary tree whose nodes fulfil the *heap property*. The heap property means that the value stored in a node must be less than or equal to the values stored in the child nodes. The heap property is fulfilled in example 1 below, but **not** in example 2 where 5 is greater than 2 and 3.

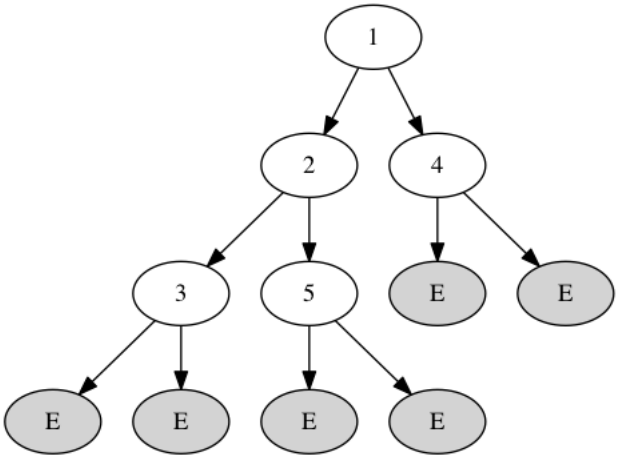


Example 1: The heap property is fulfilled.



Example 2: The heap property is not fulfilled.

In this assignment you will implement a heap based on a binary tree structure. Example 3 below shows the tree from Example 1 with empty nodes E added. The empty nodes are grey.



Example 3: The tree from example 1 with empty nodes added.

We represent the heap with the below polymorphic datatype where empty nodes are represented by `EmptyHP`.

```
type Heap<'a when 'a: equality> =
  | EmptyHP
  | HP of 'a * Heap<'a> * Heap<'a>
```

Question 1.1

- Declare a value `ex3` representing the binary tree shown in example 3 above. You may use the template:

```
let ex3 = HP(1,HP(2,HP(...
              HP(4,...
```

- Write the type of the value `ex3`. Explain why the type is either monomorphic or polymorphic.
- Declare a value `empty` representing an empty heap, i.e. a binary tree with only one empty root node. The type of the empty value is `empty : Heap<'a> when 'a : equality`.

- Declare an F# exception named `HeapError` that can be used to signal an error condition from a function on heaps. The exception should carry a string to be used to describe the error.

Question 1.2

- Declare a function

```
isEmpty : Heap<'a> -> bool when 'a : equality
```

  that returns true if a heap is the empty heap. For instance `isEmpty empty` returns true. The value `empty` is defined above.

- The *size h* of a heap *h* is the number of non–empty nodes in the binary tree. Declare a function

```
size : Heap<'a> -> int when 'a : equality
```

  that returns the size of a heap. For instance, `size ex3` returns 5.

- Declare a function `find` h of type

```
find : Heap<'a> -> 'a when 'a : equality
```

  that returns the minimum value in a non–empty heap, i.e. the root value. For instance `find ex3` returns 1.

- Declare a function `chkHeapProperty` h of type

```
chkHeapProperty : Heap<'a> -> bool when 'a : comparison
```

  that returns true if the heap h fulfils the heap property and otherwise false. The empty heap by definition fulfils the heap property. For instance `chkHeapProperty ex3` returns true.

Question 1.3

- Declare a function `map` *f h* of type

```
map : ('a -> 'b) -> Heap<'a> -> Heap<'b>
  when 'a : equality and 'b : equality
```

  where `map` *f h* returns the heap where the function *f* has been applied on all values in the heap *h*. You decide, but must explain, what order the function *f* is applied to the values in the heap. For instance `map ((+)1) ex3` returns the heap with all values in `ex3` increased by one.

- The heap `ex3` fulfils the heap property. Give an example of a function *f* such that mapping *f* on all values in `ex3` gives a new heap that does not fulfil the heap property. Given your definition of *f*, show that `chkHeapProperty (map f ex3)` returns false.

Question 2 (30%)

We shall now consider a binary *divide–and–conquer* algorithm. We will use the algorithm to implement mergesort. You do not need to know how mergesort works to do this.

Question 2.1

- Declare a function `genRandoms` *n* of type `int -> int[]` that returns an array of *n* random integers. The random integers are larger than or equal to 1 and less than 10000. For instance, `genRandoms 4` may return `[|8803;8686;2936;2521|]`.

  Hint: You can use below to define a generator `random` of type `unit -> int` to generate the random numbers.

```
let random =
  let rnd = System.Random()
  fun () -> rnd.Next(1,10000)
```

- Declare a function `genRandomsP` *n* of type `int -> int[]` that is similar to `genRandom` except that the numbers are generated in parallel to speed up the process.

  Hint: You may use the `Array.Parallel` library as explained in Section 13.6 in the book HR.

Question 2.2

Mergesort consists of three separate steps: splitting the remaining unsorted elements in two halves (`split`), identifying when the list has at most one element, and thus is trivially sorted (`indivisible`) and merging two already sorted lists together (`merge`). We implement each step below.

- Declare a function `split` *xs* of type `'a list -> 'a list * 'a list` which takes a list *xs*, say $[e_1, \ldots, e_n]$ and returns two lists with half elements in each: $([e_1, \ldots, e_{n/2}], [e_{n/2+1}, \ldots, e_n])$. For instance `split [22;746;931;975;200]` returns `([22;746],[931;975;200])`. Define and explain at least three relevant test cases.

- Declare a function `indivisible` *xs* of type `'a list -> bool`. The function returns true if the list is either empty or contains one element only, i.e. the list is trivially sorted; otherwise the function returns false. For instance `indivisible [23;34;45]` returns false.

- Declare a function `merge` *xs ys* of type

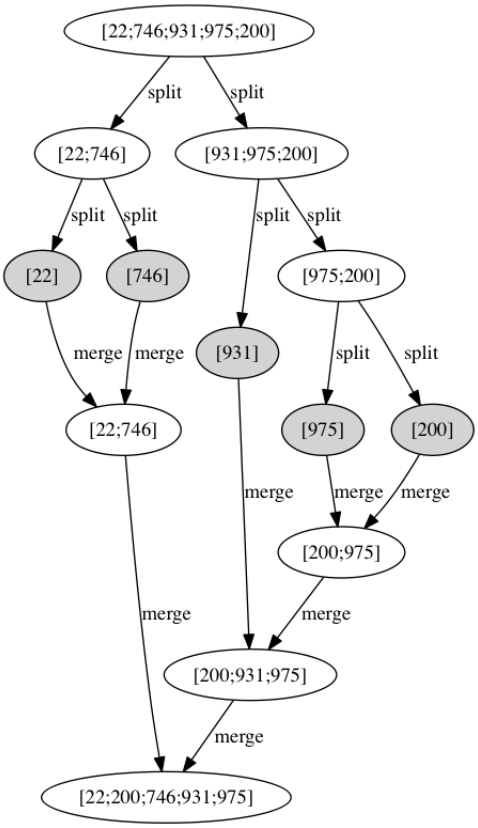
```
merge : 'a list * 'a list -> 'a list when 'a : comparison
```

  that returns the sorted merged list of *xs* and *ys*. The function `merge` can assume the two lists are sorted and does not have to check for that. For instance `merge ([1;3;4;5],[1;2;7;9])` returns `[1;1;2;3;4;5;7;9]`. Define and explain at least three relevant test cases.

Question 2.3

The process of solving a problem *p* using binary divide–and–conquer is to repeatedly divide the problem *p* into problems of half size until the divided problems are indivisible and trivially solved. The divided solutions are then merged together until the entire problem *p* is solved.

In the case of mergesort the problem *p* is the list of elements to sort. Dividing the problem is to use `split` to make two new sublists to sort. Merging two solved problems, i.e. sorted lists, is done by `merge`. Example 4 below shows the process for sorting the list `[22;746;931;975;200]`.



Example 4: Mergesort on [22;746;931;975;200]. Grey nodes are indivisible.

- Declare a function `divideAndConquer` *split merge indivisible p* of type

```
divideAndConquer : ('a -> 'a * 'a) -> ('a * 'a -> 'a)
                     -> ('a -> bool) -> 'a -> 'a
```

that implements the divide and conquer process. You may, but don't have to, use the template below.

```
let divideAndConquer split merge indivisible p =
  let rec dc p =
    if indivisible p
      then ...
      else ...
  dc p
```

Executing `divideAndConquer split merge indivisible [22;746;931;975;200]` should give give the result `[22;200;746;931;975]`.

Question 3 (20%)

In this question we work with sequences as covered in Chapter 11 in HR.

Question 3.1

- Declare the infinite sequence `triNum` of *triangular numbers*. The type of `triNum` is `seq<int>`. The triangular numbers are defined as $x_n = \frac{n(n+1)}{2}$ where x_n is the nth element in the sequence. The first element has index $n = 0$.

  Hint: You may use `Seq.initInfinite`. The sequence is `seq [0;1;3;6;...]`.

- Declare a cached version `triNumC` of `triNum` such that already computed elements are cached. The type of `triNumC` is `seq<int>`.

Question 3.2

The function `filterOddIndex` s filters out all elements e_i of the sequence s where i is odd. The function declaration is based on the assumption that the input sequence s is infinite but unfortunately goes into an infinite loop. For instance `filterOddIndex triNum` never terminates.

```
let rec filterOddIndex s =
  Seq.append (Seq.singleton (Seq.item 0 s))
             (filterOddIndex (Seq.skip 2 s))
```

- Declare your own version `myFilterOddIndex` similar to `filterOddIndex` except that it does not enter an infinite loop but returns the intended sequence.

  Hint: You may be inspired by Section 11.3 in HR. The sequence for `myFilterOddIndex triNum` is `seq [0;3;10;21;...]`.

Question 3.3

The sequence library `Seq` contains a number of functions to manipulate sequences, see Table 11.1 in HR. One such function is `Seq.zip` s_1 s_2 of type

```
(seq<'a> -> seq<'b> -> seq<'a * 'b>)
```

For instance executing `Seq.zip triNum triNum` returns the value

```
seq [(0, 0); (1, 1); (3, 3); (6, 6); ...]
```

- Declare a function `seqZip` of type

  ```
  (seq<'a> -> seq<'b> -> seq<'a * 'b>)
  ```

  that works the same as `Seq.zip`. You are not allowed to use `Seq.zip` but should implement `seqZip` using *sequence expressions* as explained in Section 11.6 in HR. You may use the template below.

  ```
  let rec zipSeq s1 s2 =
    seq {let e1 = Seq.item 0 s1
         let e2 = Seq.item 0 s2
         ... }
  ```

Question 4 (25%)

We now consider an internal domain specific language (DSL) called *Fig* to be used for specifying figures constructed from *basic figures*, that is, circles and lines. The DSL contains constructors for forming a collection of figures (constructor `Combine`), for specifying a move of a figure by a given offset (constructor `Move`) and for naming and referencing figures (constructors `Label` and `Ref`).

```
exception FigError of string
type Point = P of double * double
type Fig =
    Circle of Point * double
  | Line of Point * Point
  | Move of double * double * Fig
  | Combine of Fig list
  | Label of string * Fig
  | Ref of string
```

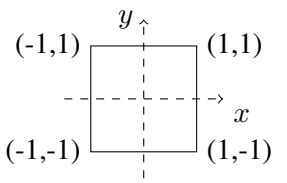
- The exception `FigError` represents an error condition from a function in the library.
- The type `Point` represents a point (x, y) in the two dimensional space.
- The type `Fig` represents the DSL for figures
  - `Circle` (p, r) is the circle with center p and radius r.
  - `Line` (p_1, p_2) is the line between the two points p_1 and p_2
  - `Move` $(d_x, d_y$,*fig*$)$ denotes the figure obtained from *fig* by moving the figures contained in *fig* as specified by d_x and d_y.
  - `Combine` *figs* is the collection of figures in *figs*.
  - `Label` (*lab*,*fig*) gives the *fig* a name *lab*. We assume *fig* does not contain references (`Ref`) such that cyclic structures are avoided.
  - `Ref` *lab* references the figure with name *lab* assuming it exists.

The example `figEx01` represents a figure consisting of a circle with center (1.0,1.0) and radius 2.0 together with a line between the points (0.0,0.0) and (1.0,1.0).

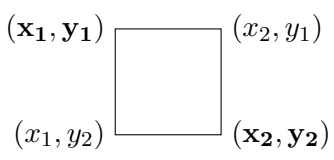
```
let figEx01 = Combine [Circle(P(1.0,1.0),2.0);Line(P(0.0,0.0),P(1.0,1.0))]
```

Question 4.1

- Declare an F# value `rectEx` of type `Fig` that represents a rectangle with the four points (-1,1) , (1,1), (1,-1) and (-1,-1) as shown in the figure to the right.



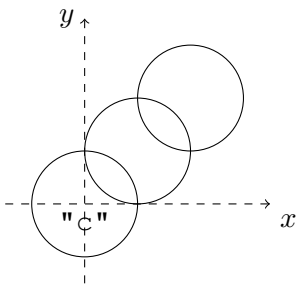
- Declare an F# function `rect` (x_1, y_1) (x_2, y_2) of type `double * double -> double * double -> Fig`, that given two orthogonal coordinates as shown in the figure to the right returns a figure representing the rectangle by its four sides. For instance `rect (-2.0,1.0) (1.0,-1.0)` returns




```
Combine [Line(P(-2.0,1.0),P(1.0,1.0));Line(P(1.0,1.0),P(1.0,-1.0));
         Line(P(1.0,-1.0),P(-2.0,-1.0));Line(P(-2.0,-1.0),P(-2.0,1.0))]
```

Question 4.2

Consider the F# value `figEx02` consisting of a labeled circle `"c"` which is referenced twice. The referenced circles are moved such that we obtain a figure like the one to the right.




```
let figEx02 =
  Combine [Label("c",Circle(P(0.0,0.0),1.0));
           Move(1.0,1.0,Ref "c");
           Move(2.0,2.0,Ref "c")]
```

- Declare an F# function `buildEnv` *fig* of type `Fig -> Map<string,Fig>` that traverses the figure *fig* and builds an environment mapping labels to figures. For instance

  ```
  let envEx02 = buildEnv figEx02
  ```

  binds `envEx02` to the value

  ```
  map [("c",Circle(P(0.0,0.0),1.0))]
  ```

Question 4.3

Given a figure *fig* and an environment *env* mapping labels to figures, we can substitute referenced figures with the actual figures.

- Declare an F# function `substFigRefs` *env fig* of type `Map<string,Fig> -> Fig -> Fig` that substitues all references with actual figures. As we substitute all references there is no need to keep the labels either. The result figure should therefore not contain any references `Ref` or labels `Label`. For instance

  ```
  let substEx02 = substFigRefs envEx02 figEx02
  ```

  binds `substEx02` to the value

  ```
  Combine
    [Circle(P(0.0,0.0),1.0);
     Move(1.0,1.0,Circle(P(0.0,0.0),1.0));
     Move(2.0,2.0,Circle(P(0.0,0.0),1.0))]
  ```

Question 4.4

We now assume that figures do not contain labels and references. For such figures, we can remove the `Move` constructors by updating the positions of the circles and lines. We thus obtain a figure consisting of `Combine`, `Circle` and `Line` constructors only.

- Declare an F# function `reduceMove` *fig* of type `Fig -> Fig` that updates the line and circle positions and removes the `Move` constructors. For instance

  ```
  let reduceEx02 = reduceMove substEx02
  ```

  binds `reduceEx02` to the value

  ```
  Combine [Circle(P(0.0,0.0),1.0);
           Circle(P(1.0,1.0),1.0);
           Circle(P(2.0,2.0),1.0)]
  ```