Thursday August 15, 2024

- The exam duration is five hours
- There are four 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 etc. during the examination. This includes any form of device that can execute programs written in F#.
- You may **NOT** use the internet in any way other than LearnIT for the duration of the exam.
- You may **NOT** use any form of code generators like Visual Studio CoPilot. All code you hand in must either be in the template we provide, or written by yourself. The use of any such tool is grounds for disciplinary action. Standard auto-completion like Intellisense is ok.
- You are (unless otherwise instructed) 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 (unless otherwise instructed) use that function in subsequent subquestions, even if you have not managed to define it. Providing the signature of the missing function will help in such cases.
- If a subquestion requires you to define a particular function, then you may (unless otherwise instructed) 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 asked for.
- Unless explicitly stated you are required to provide functional solutions, and solutions with side effects will not be considered. The one exception to this rule concerns parallelism as `Async.Parallel` returns the results of the individual processes in an array and these results may be used.
- You are required to use the provided code project `FPReExam2024` as a basis for your submission and you should **only hand in** the `Exam.fs` file (no other file). The project includes everything you need to run as an independent project, but you may also use the F# top loop. Any helper functions that we provide in `Exam.fs` file may also be part of your submission.
- Most functions that you need to write are present in the code skeleton. If an assignment asks that you write a function `isEven : int -> bool`, for instance, then there is nearly always a corresponding `let isEven _ = failwith "not implemented"` in the source file. You may change these functions (changing a `let` to a `let rec` for instance) as long as their signatures correspond to those given in the assignment. In this case that could be `let isEven x = x % 2 = 0`. Be wary of polymorphic variables as notation sometimes differs and some IDEs, for instance, will write `MyType<'a when 'a : equality>` while others may write `MyType<'a> when 'a : equality`. These are identical.

You MUST include explanations and comments to support your solutions for the questions that require them. 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. This includes using solutions or code found online, or tools that write code for you (such as CoPilot).

Your solution MUST compile. We reserve the right to fail any submission outright that does not meet this minimal requirement.

1: Shapes (25%)

Consider the following type for three basic geometric shapes

```
type shape =
    | Rectangle of float * float
    | Circle of float
    | Triangle of float * float
```

where

- `Rectale(width, height)` denotes a rectangle with width `width` and height `height`.
- `Circle(radius)` denotes a circle with radius `radius`
- `Triangle(base, height)` denotes a right triangle (one of its angles is 90 degrees) with base `base` and height `height`

Question 1.1

Create a function `area` of type `shape -> float` that given a shape `s` computes the area of `s` where the area of a

- rectangle with width w and height h is $w \times h$.
- circle with radius r is πr^2.
- triangle with base b and height h is $\frac{b \times h}{2}$

Create a function `circumference` of type `shape -> float` that given a shape `s` computes the circumference of `s` where the circumference of a

- rectangle with width w and height h is $2w + 2h$.
- circle with radius r is $2\pi r$.
- triangle with base b and height h is $b + h + \sqrt{b^2 + h^2}$, which only works because we are dealing with right triangles.

Note: You can find functions for both π and square roots in the `System.Math` library.

Examples:

```
let rect = Rectangle(2., 3.)
let circ = Circle 4.
let trig = Triangle(2., 3.)
```

```
> area rect
val it: float = 6.0

> area circ
val it: float = 50.26548246

> area trig
val it: float = 3.0
```

```
> circumference rect
val it: float = 10.0

> circumference circ
val it: float = 25.13274123

 > circumference trig
val it: float = 8.605551275
```

Question 1.2

Consider the type

```
type shapeList =
    | Empty
    | AddShape of shape * shape * shapeList
```

that denotes even length lists of shapes. We will call members of this type *shape lists*.

Create a recursive, but not a tail-recursive, function `totalArea` of type `shapeList -> float` that given a shape list `sl` returns the sum of the areas of all shapes in `sl`.

Examples::

```
totalArea Empty
val it: float = 0.0

totalArea (AddShape(rect, circ,
                    AddShape(circ, trig, Empty)))
val it: float = 109.5309649
```

Question 1.3

Create a tail recursive function, using an accumulator, `totalCircumference` of type `shapeList -> float` that given a shape list `sl` returns the sum of the circumferences of all shapes in `sl`.

Examples::

```
totalCircumference Empty
val it: float = 0.0

totalCircumference (AddShape(rect, circ,
                             AddShape(circ, trig, Empty)))
val it: float = 68.87103373
```

Question 1.4

Create a function `shapeListFold` of type

```
('a -> shape -> 'a) -> 'a -> shapeList -> 'a
```

that given the folding function `f`, an initial accumulator `acc`, and a shape list `sl` folds over `sl` from left to right in the following manner (similar to lists):

```
shapeListFold f (AddShape(s1, s2, AddShape(..., AddShape(sn, sm, Empty) ... )))
                =
f (f (... (f (f acc s1) s2) ...) sn) sm
```

As an example, a function `containsCircle` of type `shapeList -> bool`, that given a shape list `sl` returns `true` if `sl` contains a circle and false otherwise can be written in the following manner

```
let isCircle =
  function
  | Circle _ -> true
  | _        -> false

let containsCircle trs =
   shapeListFold (fun acc c -> acc || isCircle c) false trs
```

Examples:

```
> containsCircle (AddShape(rect, circ, AddShape(circ, trig, Empty)))
val it: bool = true

> containsCircle (AddShape(rect, rect, AddShape(trig, trig, Empty)))
val it: bool = false
```

Question 1.5

Create the non-recursive functions `totalArea2` and `totalCircumference2` that behave exactly the same as `totalArea` and `totalCircumference` respectively, but which are coded using the `shapeListFold` function from Q1.4. You are encouraged to use the `area` and `circumference` functions from Q1.1.

2: Code Comprehension (25%)

Consider the following three functions

```
let foo =
    function
    | c when Char.IsWhiteSpace c -> c
    | c when c > 'w'             -> char (int c - 23)
    | c when c < 'x'             -> char (int c + 3)

let bar (str : string) = [for c in str -> c]

let baz str =
    let rec aux =
        function
        | [] -> ""
        | c :: cs -> string (foo c) + (aux cs)

    aux (bar str)
```

Question 2.1

- What are the types of functions `foo`, `bar`, and `baz`? Use standard F# type notation.
- What do the functions `foo`, `bar`, and `baz` do? Focus on what they do rather than how they do it.
- What would be appropriate names for functions functions `foo`, `bar`, and `baz`?

Question 2.2

Compiling the `foo` function generates a warning `Incomplete pattern matches on this expression.`. Why do we get this warning?

Create a function `foo2` that behaves exactly the same as `foo` from Question 2.1 but which does not generate this warning on compilation.

Question 2.3

Create a non-recursive function `baz2` that behaves the same as `baz` from Question 2.1 for all possible inputs but which is constructed using higher-order function(s).

You are encouraged to use `bar` and `foo` but the only recursive functions that you are allowed to use are higher-order functions from the standard library.

Question 2.4

The function `baz` from Question 2.1 is not tail recursive. Demonstrate why.

To make a compelling argument you should evaluate a function call of the function, similarly to what is done in Chapter 1.4 of HR, and reason about that evaluation. You need to make clear what aspects of the evaluation tell you that the function
is not tail recursive. Keep in mind that all steps in an evaluation chain must evaluate to the same value `((5 + 4) * 3 --> 9 * 3 --> 27` , for instance).

You do not have to step through the foo- or the bar functions. You are allowed to evaluate these functions immediately.

Question 2.5

Create a function `bazTail` that behaves the same way as `baz` from Question 2.1 but which is tail recursive and coded using continuations.

3: Atbash Cipher (25%)

The Atbash Cipher is a very simple cipher where reversing the alphabet helps create the encryption. In this system

- `a` becomes `z`
- `b` becomes `y`
- `c` becomes `x`
- ...
- `n` becomes `m`
- `m` becomes `n`
- ...
- `z` becomes `a`
- whitespace characters (`' '`) remain the same
  As an example, the text `"hello world"` becomes `"svool dliow"`

Important: For all strings used in each sub-question of Question 3, except for Question 3.5, you may assume that they contain only lower-case letters between `a` and `z` and spaces `' '`.

Question 3.1

Create a function `encrypt` of type `string -> string` that given a string `text` encodes `text` using the Atbash Cipher as explained above.

Examples:

```
> encrypt "hello world"
- val it: string = "svool dliow"

> encrypt "get to da choppaaah"
- val it: string = "tvg gl wz xslkkzzzs"
```

Question 3.2

Create a function `decrypt` of type `string -> string` that given a string `text` decodes `text` using the Atbash Cipher.

Make sure that `decrypt (encrypt text) = text` for all possible instances of `text`.

Examples:

```
> decrypt "svool dliow"
val it: string = "hello world"

> decrypt "tvg gl wz xslkkzzzs"
val it: string = "get to da choppaaah"
```

Question 3.3

Create a function `splitAt` of type `int -> string -> string list` that given an integer `i` and a string `str` splits `str` into chunks of size `i` and returns the result as a list. It is possible that the last element in the list is shorter than `i`, but it must not be the empty string `""`. You may assume that `i` is greater than `0`.

Hint: Remember that

- `str[i..j]` returns the substring from index `i` to index `j` in the string `str`
- `str[i..]` returns the substring from index `i` to the end in the string `str`
- `str[..j]` returns the substring from the start to index `j` in the string `str`

Examples:

```
> splitAt 1 "hello world"
val it: string list = ["h"; "e"; "l"; "l"; "o"; " "; "w"; "o"; "r"; "l"; "d"]

> splitAt 2 "hello world"
val it: string list = ["he"; "ll"; "o "; "wo"; "rl"; "d"]

> splitAt 7 "hello world"
val it: string list = ["hello w"; "orld"]
```

Question 3.4

Create a function `parEncrypt` of type `string -> int -> string` that given a string `str` and a chunk size `i` works exactly like `encrypt` for `str` but which splits `str` into chunks of size `i`, using `splitAt` from Question 3.3, encrypts each chunk in parallel, and then recombines them all into one string.

Note: For this assignment you may use arrays as `Async.Parallel` returns an array, but you may not modify this array in any way.

Question 3.5

For this assignment you may **not** assume that the string only contains lower case characters and spaces.

Create a term `parseEncrypt` of type `Parser<string>` that is a parser that parses a string `text` and encodes `text` using the Atbash Cipher (exactly like `encrypt` and `parEncrypt`).

The parser should run as long as valid characters are encountered which are either letters from `'a'`-`'z'`, `'A'`-`'Z'` or whitespaces `' '`, where upper case characters are encrypted as their lower case counterparts.

Important: All work must be done inside the parser. You may not, for instance, generate a string with only lower-case characters and then pass that to the parser.

You will find functions in the `System.Char` library that are useful for handling upper- and lower case characters.

Examples:

```
> run parseEncrypt "hello world"
val it: ParserResult<string> = Success "svool dliow"

> run parseEncrypt "HELLO world"
val it: ParserResult<string> = Success "svool dliow"

> run parseEncrypt "HELLO world!!!!!"
val it: ParserResult<string> = Success "svool dliow"
```

4: Tally clickers (25%)

For this assignment we will work with tally clickers. In the real world, a tally clicker is a device that displays a number on a series of spinning wheels, and when a button is pushed on the device the wheels spin such that the number is increased by one. A clicker with three wheels can give any number between 0 and 999.

Instead of just numbers, we can have wheels that contain any character, but other than that the clicker works in the same way as normal. When the button is clicked

- the wheel furthest to the right spins one step
- if this wheel spins to the beginning, the second wheel from the right spins one step, and if it spins to the beginning the third wheel from the right spins one step, and so on.

For instance, if we have a clicker with two wheels of the form `{a, b, c}`, then `aa` is the initial state, and clicking continuously will result in the sequence `ab`, `ac`, `ba`, `bb`, `bc`, `ca`, `cb`, `cc` after which it loops back to `aa` and the cycle starts again.

Question 4.1

For this assignment (4.1) you may, but do not have to, use mutable types like arrays. You may not, however, now or in a future assignment (4.2-4.5) work with the arrays you create here in a non-functional way which means that you must never change their contents after they have been created.

Create a type `clicker` that represents a tally clicker where all wheels are identical and the symbols on the wheels have type `char`. You must keep track of

- what wheel is in the clicker (since the wheels are identical this should only be stored once)
- the position that every wheel is at (the positions of the wheels change over time so every wheel should have its own position)

Create a function `newClicker` of type `char list -> int -> clicker` that given a list of characters `wheel`, and an integer `numWheels`, returns a clicker that uses `numWheel` occurrences of the wheel `wheel` each of which is in their first position, where the first position is the first element of `wheel`.

For example the clicker

- from the introduction with two `{a, b, c}` wheels is created by `newClicker ['a'; 'b'; 'c'] 2`.
- The clicker in the image at the top of the question with three wheels from 0-9 is created by `newClicker ['0'..'9'] 3`.

Important: For full credit pressing the clicker once must be possible to do in linear time with respect to the number of wheels in the clicker, not the number of characters on the wheels. Be wary of the lookup times for the data structures you choose.

Question 4.2

Create a function `click` of type `clicker -> clicker` which takes a clicker `cl` and returns a clicker that is identical to `cl` but where the state has been increased one step as described above.

Create a function `read` of type `clicker -> string` that given a clicker `cl` returns the current state of `cl` as a string, by reading the current value of each wheel in order.

Examples:

```
let cl = newClicker ['a'; 'b'; 'c'] 2

> cl |> read
val it: string = "aa"

> cl |> click |> read
val it: string = "ab"

> cl |> click |> click |> read
val it: string = "ac"

> cl |> click |> click |> click |> read
val it: string = "ba"

> cl |> click |> click |> click |> click |> read
val it: string = "bb"

> cl |> click |> click |> click |> click |> click |> read
val it: string = "bc"

> cl |> click |> click |> click |> click |> click |> click |> read
val it: string = "ca"
```

Question 4.3

For this assignment we will be using a state monad to hide the clicker. The state monad you will be working on is very similar to the one that you used for Assignment 6, but the state is much simpler (the clicker from Question 4.1).

```
type StateMonad<'a> = SM of (clicker -> 'a * clicker)

let ret x = SM (fun cl -> (x, cl))

let bind f (SM a) : StateMonad<'b> =
    SM (fun cl ->
            let x, cl'  = a cl
            let (SM g) = f x
            g cl')

let (>>=) x f = bind f x  let (>>>=) x y = x >>= (fun _ -> y)

let evalSM cl (SM f) = f cl
```

Create functions `click2` of type `StateMonad<unit>` and `read2` of type `StateMonad<string>` where

- `click2` increases the state of the clicker one step.
- `read2` reads the curren state of the clicker.

Important: You cannot use monadic operators, like `bind` or `ret`, for `click` or `read2` as you must break the abstraction of the state monad to implement these functions, but we include them here for debugging purposes, and you will need them for Question 4.4 and 4.5

Hint: Use `click` and `read` from Question 4.2.

Examples:

```
> read2 |> evalSM cl |> fst
val it: string = "aa"

> (click2 >>>= read2) |> evalSM cl |> fst
val it: string = "ab"

> (click2 >>>= click2 >>>= click2 >>>= click2 >>>= read2) |> evalSM cl |> fst
val it: string = "bb"
```

Question 4.4

For this assignment you must use monadic operators (like `bind` or `ret`), but not computation expressions. Moreover you may not break the abstraction of the state monad (you may not pattern match on anything of type `StateMonad`).

Create a function `multipleClicks` of type `int -> StateMonad<string list>` that given an integer `x` returns a list containing every state, starting with the current one, of clicking the clicker `x - 1` times, so the total number of elements in the list will be `x` since the initial state before any clicks are done is also included.

Examples:

```
> 10 |> multipleClicks |> evalSM cl |> fst
val it: string list =
  ["aa"; "ab"; "ac"; "ba"; "bb"; "bc"; "ca"; "cb"; "cc"; "aa"]
```

Question 4.5

For this assignment you must use computation expressions and you will need the following definitions.

```
type StateBuilder() =

    member this.Bind(f, x)    = bind x f
    member this.Return(x)     = ret x
    member this.ReturnFrom(x) = x
    member this.Combine(a, b) = a >>= (fun _ -> b)

let state = new StateBuilder()
```

Create a function `multipleClicks2` that behaves exactly as `multipleClicks` from Question 4.4, but which uses computation expressions, and not the standard monadic operators (like `bind` or `ret`), and which does not break the abstraction of the state monad (you may not pattern match on anything of type `StateMonad`).