Wednesday May 29, 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 `FPExam2024` 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. See the `README` for details. 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 that does not meet this minimal requirement.

1: Transactions (25%)

Consider the following type for a sequence of transactions

```
type transactions =
     | Empty
     | Pay     of string * int * transactions
     | Receive of string * int * transactions
```

where

- `Empty` means that there are no more transactions,
- `Pay(name, amount, trs)` means that we pay `amount` of money to `name` and `trs` contains the remaining transactions
- `Receive(name, amount, trs)` means that we receive `amount` of money from `name` and `trs` contains the remaining transactions

Note: We call a value of type `transactions` a *sequence of transactions* even though it is not a sequence (`Seq`) from the standard library.

Question 1.1

Create a recursive, but not tail-recursive, function `balance` of type `transactions -> int` that given a sequence of transactions `trs` returns the amount of money left after all transactions have been done, i.e.

- Remove `amount` from the result if `trs` is equal to `Pay(name, amount, trs')`
- Add `amount` to the result if `trs` is equal to `Receive(name, amount, trs')`

Examples:

```
> balance Empty
val it: int = 0

> balance (Pay("Alice", 500,
               Receive("Bob", 200, Empty)))
val it: int = -300
```

Question 1.2

Create a tail-recursive function, using an accumulator, `balanceAcc` of type `transactions -> int` that behaves exactly as `balance` from Question 1.1.

Question 1.3

Create a function `participants` of type `transactions -> Set<string> * Set<string>` that given a sequences of transactions `trs` returns two sets of strings where the first contains the names of all people being paid, and the second contains the names of all the people you have received money from in `trs`.

Examples:

```
> participants (Pay("Alice", 500,
                 Receive("Bob", 200, Empty)))
val it: Set<string> * Set<string> =
    (set ["Alice"], set ["Bob"])
```

Question 1.4

Create a function `balanceFold` of type

```
('a -> string -> int -> 'a) ->
('a -> string -> int -> 'a) ->
'a -> transactions -> 'a
```

that given functions `payFolder` and `receiveFolder`, an initial accumulator `acc`, and a sequence of transactions `trs` folds over `trs` from left to right and updates `acc` by

- `payFolder name amount acc` if `trs` is equal to `Pay(name, amount, _)`
- `receiveFolder name amount acc` if `trs` is equal to `Receieve(name, amount, _)`

The functions `balance` and `balanceAcc` from Questions 1.1 and 1.2 can, for instance, be written as

```
let balance' trs =
   balanceFold (fun acc _ x -> acc - x)
               (fun acc _ x -> acc + x)
               0
               trs
```

Examples:

```
> balance' Empty
val it: int = 0

> balance' (Pay("Alice", 500,
                 Receive("Bob", 200, Empty)))
val it: int = -300
```

Question 1.5

Create a function `collect` of type `transactions -> Map<string, int>` that given a sequence of transactions `trs` returns a map where the keys are the names of those we either pay money to or receive money from in `trs`, and the value is the total sum that we have paid and received from the corresponding individual. This sum will be negative if we have paid more than we have received.

For full marks your function must not be recursive and use the `balanceFold` function from Question 1.4.

Examples:

```
> collect Empty
    val it: Map<string,int> = map []

> collect (Pay("Alice", 500,
               Receive("Bob", 200, Empty)))
val it: Map<string,int> =
    map [("Alice", -500); ("Bob", 200)]

> collect (Pay("Bob", 100,
               Pay("Alice", 500,
                   Receive("Bob", 200, Empty))))
val it: Map<string,int> =
    map [("Alice", -500); ("Bob", 100)]
```

2: Code Comprehension (25%)

Consider the following three functions

```
let foo (x : char) =
    x |> int |> fun y -> y - (int '0')

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

let rec baz =
    function
    | [] -> 0
    | x :: xs -> x + 10 * baz xs
```

Question 2.1

- What are the types of functions `foo`, `bar`, and `baz`?
- 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`?
- The function `foo` only behaves reasonably if certain constraint(s) are met on its argument. What is/are these constraints?
- The function `baz` only behaves reasonably if certain constraint(s) are met on its argument. What is/are these constraints?

Question 2.2

Using only

- function composition
- `foo`, `bar`, and `baz` from Question 2.1
- `List.map` and `List.rev` from the standard library

create a function `stringToInt` of type `string -> int` that given a string containing a single non-negative integer that fits into a 32-bit signed integer, returns its corresponding integer.

Examples:

```
> stringToInt "0"
val it: int = 0

> stringToInt "987655443"
val it: int = 987655443
```

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) from the `List` library.

Question 2.4

The function `baz` from Question 2.1 is not tail recursive. Explain why. To make a compelling argument you must 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).

Question 2.5

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

3: Caesar cipher (25%)

Julius Caesar is alleged to have used a cipher when he wanted to communicate covertly. The cipher worked by offsetting letters in the alphabet by three steps, wrapping around when you hit the end of the alphabet. In this system

- `a` becomes `d`
- `b` becomes `e`
- ...
- `z` becomes `c`.
  As an example, the text `hello world` becomes `khoor zruog`

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 -> int -> string` that given a string `text` and an integer `offset` encodes `text` using the Caesar Cipher by offsetting each character in `text` by `offset`, wrapping around if `z` is reached.

Examples:

```
> encrypt "hello world" 0
- val it: string = "hello world"

> encrypt "hello world" 3
- val it: string = "khoor zruog"

> encrypt "hello world" 127
- val it: string = "ebiil tloia"
```

Question 3.2

Create a function `decrypt` of type `string -> int -> string` that given a string `text` and an integer `offset` decodes `text` using the Caesar Cipher by offsetting each character in `text` by the negation `offset`, wrapping around if `a` is reached.

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

Examples:

```
> decrypt "hello world" 0
val it: string = "hello world"

> decrypt "khoor zruog" 3
val it: string = "hello world"

> decrypt "ebiil tloia" 127
val it: string = "hello world"
```

Question 3.3

Create a function `decode` of type `string -> string -> int option` that given two strings `plainText` and `encryptedText` returns an integer option which is

- `Some offset` where `offset` is the minimal non-negative offset that encodes `plainText` to `encryptedText` using the Caesar Cipher if such an offset exists
- `None` otherwise

```
> decode "hello world" "hello world"
val it: int option = Some 0

> decode "hello world" "hello mom"
val it: int option = None

> decode "hello world" "khoor zruog"
val it: int option = Some 3

> decode "hello world" "ebiil tloia"
val it: int option = Some 23
```

Question 3.4

Create a function `parEncrypt` of type `string -> int -> string` that works exactly the same way as `encrypt` but which splits the string into separate words at every white space, decrypts each word in parallel, and then recombines all of the encrypted words into one string.

Note: For this assignment you may use arrays as `Async.Parallel` returns an array. There is also a function that splits a string into an array (`str.Split` , if `str` is a string) that you may use. You may not, however, mutate either 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 function `parseEncrypt` of type `int -> Parser<string>` that given an integer `offset` returns a parser that parses a string `text` and encodes `text` using the Caesar Cipher by offsetting each character in `text` by `offset`, wrapping around if `z` is reached (exactly like `encrypt` and `parEncrypt`).

The parser should run as long as valid characters are encountered which are either letters from `'a'`-`'z'` or whitespaces `' '`.

Examples:

```
> run (parseEncrypt 0) "hello world"
val it: ParserResult<string> =
    Success "hello world"

> run (parseEncrypt 3) "hello world"
val it: ParserResult<string> =
    Success "khoor zruog"

run (parseEncrypt 127) "hello world"
val it: ParserResult<string> =
    Success "ebiil tloia"

run (parseEncrypt 127) "hello World"
val it: ParserResult<string> =
    Success "ebiil "
```

4: Letterboxes (25%)

For this assignment we will work with letterboxes. A letterbox allows senders to put letters in a letterbox and the owner to read messages from individual senders in the order that they were posted. Consider the following exchange:

- Alice places a letter with the text `"Hello John!"`
- Bob places a letter with the text `"John, how are you?"`
- Alice places a letter with the text `"Good bye John."`

If the letterbox owner now first reads the letter from Bob, then two letters from Alice they will get them in the order `"John, how are you?"`, `"Hello John!"`, `"Good bye John."`.

Question 4.1

Letters consist of a message text, and a sender. Both are strings.

Create a type `letterbox` that contains letters and that allows senders to post new letters in the letterbox and receivers to read letters from a sender, one at a time, from the letterbox in the order that they were put in the letterbox.

Posting new letters in and reading letters from the letterbox must not have worse than linear time complexity with respect to the number of letters sent by the sender currently in the box.

Hint: Creating a separate type for letters is not the way to go.

Create a function `empty` of type `unit -> letterbox` that returns an empty letterbox.

Question 4.2

Create a function `post` of type `string -> string -> letterbox -> letterbox` that given a string `sender`, a string `message`, and a letterbox `mb` adds `message` to the end of the messages from `sender` in `mb` and returns the updated letterbox.

Create a function `read` of type `string -> letterbox -> option (string * letterbox)` that given a string `sender` and a letterbox `mb` returns

- `Some (message, mb')` where `message` is the first message from `sender` in `mb`, if such a message exists, and `mb'` is the same letterbox as `mb` but with `message` removed
- `None` if there are no messages from `sender`

Examples:

```
> empty () |>
  post "Alice" "Hello John!" |>
  post "Bob" "John, how are you?" |>
  post "Alice" "Good bye John." |>
  read "Alice" |>
  Option.map fst
val it: string option = Some "Hello John!"

> empty () |>
  post "Alice" "Hello John!" |>
  post "Bob" "John, how are you?" |>
  post "Alice" "Good bye John." |>
  read "Charlie" |>
  Option.map fst
val it: string option = None
```

Question 4.3

For this assignment we will be using a state monad to hide the letterbox. 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 letterbox from Question 4.1).

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

let ret x = SM (fun s -> Some (x, s))
let fail  = SM (fun _ -> None)
let bind f (SM a) : StateMonad<'b> =
    SM (fun s ->
        match a s with
        | Some (x, s') ->
            let (SM g) = f x
            g s'
        | None -> None)

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

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

Create functions `post2 : string -> string -> SM<unit>` and `read2 : string -> SM<string>` where

- `post2` takes strings `sender` and `message` and places `message` from `sender` in the letterbox.
- `read2` take a string `sender` and returns the first message in the letterbox from `sender` and removes the message from the letterbox.

Removing a message from a sender that has no sent messages should result in failure (monadic `fail`, **not** `failwith`).

Important: You cannot use monadic operators, like `bind` or `ret`, for `post2` 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.

Hint: use `post` and `read` from Question 4.2.

Examples:

```
> post2 "Alice" "Hello John!" >>>= post2 "Bob" "John, how are you?" >>>=
  post2 "Alice" "Good bye John." >>>= read2 "Alice" |> evalSM |> Option.map fst
val it: string option = Some "Hello John!"

> post2 "Alice" "Hello John!" >>>= post2 "Bob" "John, how are you?" >>>=
  post2 "Alice" "Good bye John." >>>= read2 "Charlie" |> evalSM |> Option.map fst
val it : string option = None
```

Question 4.4

For this assignment you may, if you want to, use computation expressions in which case 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()
```

You may also solve the assignment using monadic operators, but you may not break the abstraction of the state monad (you may not pattern match on anything of type `StateMonad`).

We define logs as lists of either posting or reading messages.

```
type MType =
    | Post of string * string
    | Read of string
type log = MType list
```

where

- `Post(sender, message)` represents the message `message` being sent by `sender`
- `Read(sender)` represents receiving and message from `sender`.

Create a function `trace` of type `log -> StateMonad<string list>` that given a log `l` goes through every element in `l` and

- Posts `message` from `sender` if the next element from the log is `Post(sender, message)`
- Stores `message` in the result (the `string list` in the monad return type) if the next element from the log is `Read(message)`

Examples:

```
> [Post("Alice", "Hello John!"); Post("Bob", "John, how are you?");
   Post("Alice", "Good bye John.");
   Read("Bob"); Read("Alice"); Read("Alice")] |>
   trace |> evalSM |> Option.map fst
val it: string list option =
    Some ["John, how are you?"; "Hello John!"; "Good bye John."]

> [Post("Alice", "Hello John!"); Post("Bob", "John, how are you?");
   Post("Alice", "Good bye John.");
   Read("Charlie"); Read("Bob"); Read("Alice");  Read("Alice")] |>
   trace |>  evalSM |> Option.map fst
val it: string list option = Some None
```