Semesterprüfung

Name		
Vorname		
WIN/ZH		
Resultat	Punkte:	Note:

Bemerkungen.

- Wählen Sie **3** Aufgaben aus die Sie bearbeiten (Die am besten gelösten drei Aufgaben werden bewertet).
- Erlaubte Hilfsmittel: Alles ausser Kommunikationsmittel (Internet, Handy, etc.).
- Abgabe: Für jede Aufgabe eine .fsx Datei, Theorieaufgaben entweder als Kommentar in der .fsx Datei oder als separate .txt Datei. Alle Dateien zusammen in einem Ordner, der Ihren Namen trägt, bei Marko auf den memory-stick speichern.

Aufgabe 1 (Listen und Pattern-Matching)**.** (30 Pt.)

a) Gegeben ist folgende Implementierung der Funktion `revMap`.

```
let rec revMap f =
    function
    | [] -> []
    | x::xs -> (revMap f xs) @ [f x]
```

i) Was berechnet die Funktion `revMap`? (4 Pt.)

ii) Was können Sie über die Qualität der Implementierung sagen? Identifizieren und diskutieren Sie Schwachstellen. (6 Pt.)

b) Die Funktion `zip` ist wie folgt gegeben:

```
let rec zip lst1 lst2 =
    match lst1,lst2 with
    | x::xs,y::ys -> (x,y)::(zip xs ys)
    | _ -> []
```

Implementieren Sie mithilfe von `zip` eine Funktion `ordered`, die Listen darauf testet, ob diese (bezüglich `<=`) geordnet sind. (10 Pt.)
Test: Das Skript

```
let rec zip lst1 lst2 =
    match lst1,lst2 with
    | x::xs,y::ys -> (x,y)::(zip xs ys)
    | _ -> []

let ordered lst = <Ihr Code>

ordered [1;10;13;21], ordered [2;10;1], ordered []
```

produziert folgende FSI Meldung:

```
val zip : lst1:'a list -> lst2:'b list -> ('a * 'b) list
val ordered : lst:'a list -> bool when 'a : comparison
val it : bool * bool * bool = (true, false, true)
```

Fortsetzung:

c) Implementieren Sie eine Funktion `doubleFirst`, die bei Eingabe eines String das erste Zeichen verdoppelt, bei Eingabe des leeren String soll der leere String zurückgegeben werden. Benützen Sie ein "Active Pattern". (10 Pt.)
Test: Das Skript

```
let (|First|_|) x = <Ihr Code>

let doubleFirst w = <Ihr Code>

doubleFirst "sda", doubleFirst ""
```

produziert folgende FSI Meldung:

```
val ( |Start|_| ) : x:string -> char option
val doubleFirst : w:string -> string
val it : string * string = ("ssda", "")
```

Aufgabe 2 (Funktionen und Typen)**.** (30 Pt.)

a) Erklären Sie Currying und Uncurrying. Erläutern Sie, wieso Currying für die funktionale Programmierung wichtig ist. (6 Pt.)

b) Implementieren Sie Currying **oder** Uncurrying für zweistellige Funktionen in F#. (6 Pt.)
Test: Das Skript

```
let curry f x y = <Ihr Code>

let uncurry f x = <Ihr Code>

uncurry (+) (3,7) ,curry (fun (x,y) -> x*y) 3 5
```

produziert folgende FSI Meldung:

```
val curry : f:('a * 'b -> 'c) -> x:'a -> y:'b -> 'c
val uncurry : f:('a -> 'b -> 'c) -> 'a * 'b -> 'c
val it : int * int = (10, 15)
```

c) In F# müssen alle Einträge einer Liste vom selben Typ sein. Sie können zum Beispiel keine Liste als `[[1];1]` initialisieren. Deklarieren Sie eine Datentyp `'a deepList`, der Listen modelliert, die wiederum "Listen" beliebiger Tiefe enthalten können. Sie sollten in der Lage sein zum Beispiel "Listen" von der Form `[2;[[3;4];5];[[[]]];[[[6]]];7;8;[]]` zu deklarieren. (10 pt.)

d) Implementieren Sie eine Funktion `depth` vom Typ `depth :'a deepList -> int`, die die Tiefe einer `deepList` berechnet. (8 Pt.)
Test: Das Skript

```
let rec depth = <Ihr Code>

depth <ls> // wobei ls der Liste
           // [2;[[3;4];5];[[[]]];[[[6]]];7;8;[]]
           // enspricht
```

produziert folgende FSI Meldung:

```
val depth : _arg1:'a deepList -> int
val it : int = 4
```

Aufgabe 3 (Rekursion). (30 Pt.)

a) Erklären Sie den Begriff der Endrekursion und diskutieren Sie seine Bedeutung in der funktionalen Programmierung. (6 Pt.)

b) Schreiben Sie endrekursive Varianten der Funktion `zip` aus Aufgabe 1.

- Mit dem Akkumulator-Pattern (7 Pt.)
- Mit "continuation passing style" (7 Pt.)

c) Gegeben ist der Fixpunktkombinator

```
let rec fix f g = f (fix f) g
```

Deklarieren Sie eine Funktion `f`, so dass die Funktion `fix f` bei Eingabe eines Strings auf Palindromität testet (10 pt.)
Test: Das Skript

```
let rec fix f g = f (fix f) g

let f h = <Ihr Code>

fix f "yxaxay", fix f "a", fix f "aba", fix f "abba",fix f ""
```

produziert folgende FSI Meldung:

```
val fix : f:(('a -> 'b) -> 'a -> 'b) -> g:'a -> 'b
val f : h:(string -> bool) -> _arg1:string -> bool
val it : bool * bool * bool * bool * bool = (false, true, true, true, true)
```

Aufgabe 4 (Funktionaler Entwurf)**.** (30 Pt.)

Für die folgende Aufgabe wird Ihre Modellbildungskompetenz bewertet. Versuchen Sie möglichst einen Entwurf zu realisieren, der sich am funktionalen Paradigma orientiert:

- Algebraische Datentypen (z. B. Records)
- Möglichst keine veränderlichen Daten
- (End-)Rekursion anstelle von Iteration

Modellieren Sie endliche Zustandsautomaten (FSA) nach folgenden Vorgaben:

- Ein endlicher Zustandsautomat besteht aus folgenden Komponenten:
  - Eine endliche Menge von Zuständen
  - Eine Zustandsübergangsfunkion (kann zum Beispiel als `System.Collections.Generic.Map` oder `System.Collections.Generic.Dictionary` modelliert werden)
  - Eine Menge von akzeptierenden Zuständen (Teilmenge der Zustände)
- Sie sollen in der Lage sein, einen FSA auf einem endlichen Wort (String) operieren zu lassen, dabei soll bei jedem Zustandsübergang des Automaten der neue Zustand und das Restwort auf der Konsole ausgegeben werden. Zudem sollte, wenn das Wort vom Automaten vollständig gelesen wurde noch ausgegeben werden ob das Wort akzeptiert wurde oder nicht (ob der FSA am Schluss in einem akzeptierenden Zustand ist).

  Eine mögliche Ausgabe könnte etwa wie folgt aussehen:

```
Processing [1]11010111 ...new state = q1
Processing [1]1010111 ...new state = q0
Processing [1]010111 ...new state = q1
Processing [0]10111 ...new state = q2
Processing [1]0111 ...new state = q2
Processing [0]111 ...new state = q1
Processing [1]11 ...new state = q0
Processing [1]1 ...new state = q1
Processing [1] ...new state = q0
Accepted with final state q0
```