Heinrich-Heine-Universität Düsseldorf
Institut für Informatik
Lehrstuhl Softwaretechnik und Programmiersprachen
Dr. Philipp Körner

09. Februar 2023

Klausur

Einführung in die Funktionale Programmierung
Wintersemester 2022/23

Nachname: _____ Vorname: _____

Matrikelnummer: _____

Zugelassene Hilfsmittel: verkürztes Clojure Cheat Sheet
Dauer: 90 Minuten

Diese Klaujur enthält 10 nummerierte Seiten. Prüfen Sie bitte zuerst, ob alle Seiten vorhanden sind. Mit Ihrer Unterschrift versichern Sie, eine vollständige Klausur erhalten zu haben.

Unterschrift: _____

Schalten Sie bitte jegliche elektronischen Geräte aus

Diesen Teil bitte nicht ausfüllen:

Aufgabe	1	2	3	4	5	6	Σ
Punktzahl	19	12	28	18	14	9	100
Erreicht							

Aufgabe 1 [19 Punkte]

Gegeben sei ein Präfixbaum, der als verschachtelte Maps dargestellt ist, etwa:

```
{\s {\i {\m {\p {\l {\e {:end true}}}}}},
 \e {\a {\s {\y {:end true}},
         \g {\e {\r {:end true}},
             \l {\e {:end true,
                     \s {:end true}}}}}}}
```

Um zu entscheiden, ob ein Wort enthalten ist, schlägt man Buchstabe für Buchstabe nach. Am Ende des Wortes wird das Keyword `:end` auf `true` abgebildet um zu signalisieren, dass dieses Wort enthalten und nicht nur ein Präfix eines anderen Wortes ist. Der Baum oben enthält also die Wörter “simple”, “easy”, “eager”, “eagle” und “eagles”.

(a) [14 Punkte] Schreiben Sie eine Funktion `(lookup prefix-tree word)`, die einen solchen Baum (eine Map) und einen String `word` nimmt und entscheidet, ob das Wort im Baum enthalten ist oder nicht (d.h. entsprechend einen truthy oder falsey Wert zurückgibt).

Die Verwendung von `get-in` ist nicht erlaubt (`get` aber natürlich schon!).

(b) [5 Punkte] Welches Problem könnte auftreten, wenn eine Rekursion sehr tief geht bzw. sehr viele rekursive Aufrufe macht? Wie kann man dieses Problem lösen?

Aufgabe 2 [12 Punkte]

Wie sind Maps (im allgemeinen Fall) in Clojure implementiert? Gehen Sie insbesondere auf

- die zugrundeliegende Datenstruktur,
- die Verwendung des Hashcodes der Schlüssel,
- und die effiziente Umsetzung von Immutability

ein. Skizzieren Sie zudem ein Beispiel, in dem ein Element eingefügt wird, und beschreiben Sie dieses kurz. Das Beispiel soll das Konzept vollständig beschreiben (es soll also nicht trivial sein). Implementierungsdetails hingegen dürfen vereinfacht werden.

Aufgabe 3 [28 Punkte]

(a) [10 Punkte] Beschreiben Sie das epochale Zeit-Modell. Definieren Sie dazu insbesondere die Begriffe

- Wert,
- Identität,
- Beobachter,
- Zustand und
- Zustandsübergang.

Gehen Sie zudem auf das Zusammenspiel von Beobachtern und Zustand ein, sowie auf das Verhalten von Berechnungen, wenn sich der Zustand währenddessen verändert.

(b) [7 Punkte] Gegeben sei das Atom `(def a (atom 42))`. Welches Problem hat der folgende Code, der als Einheit ausgeführt werden soll? Geben Sie eine problemlose Version an.

```
(if (zero? @a)
  (reset! a 42)
  (swap! a dec))
```

(c) [5 Punkte] John behauptet, `(alter ref f ... args ...)` ist eigentlich nicht nötig. Stattdessen könne man immer `(ref-set ref (f @ref ... args ...))` (innerhalb einer Transaktion) aufrufen. Liegt John richtig? Begründen Sie Ihre Antwort.

(d) [6 Punkte] Gegeben seien die Refs

```
(def a (ref 0))  (def b (ref 0))  (def c (ref 0))
```

Betrachten Sie folgende Transaktionen:

Transaktion A:

```
(dosync (alter a + 1))
```

Transaktion B:

```
(dosync (alter b + @a))
```

Transaktion C:

```
(dosync (alter a + 1)
        (alter b + 1))
```

Transaktion D:

```
(dosync (alter c + 1)
        (commute a + 1))
```

Was sind jeweils die Konfliktmengen der Transaktionen?

Transaktion	**Konfliktmenge**
A	
B	
C	
D	

Aufgabe 4 [18 Punkte]

Das Gefangenendilemma ist ein Problem aus der Spieltheorie. Zwei Parteien müssen über mehrere Runden unabhängig voneinander entscheiden, ob sie miteinander kooperieren oder versuchen, die jeweilig andere Partei zu betrügen. In dieser Aufgabe sollen verschiedene Strategien aufgrund der bisher gespielten Runden implementiert werden.

Das Verhalten einer Partei (`:coop` oder `:betray`) in der nächsten Runde ist wie folgt festgelegt:

- `:copycat` führt dieselbe Aktion wie die andere Partei in der vorherigen Runde aus (oder kooperiert, wenn es keine vorherige Runde gab).
- `:copykitten` betrügt nur, wenn sie zuletzt zweimal betrogen wurde.
- `:snitch` betrügt immer.
- `:grudger` kooperiert, außer er wurde (irgendwann) einmal betrogen.

Schreiben Sie eine um Strategien erweiterbare Funktion `(next-move m)`, wobei `m` eine Map der Form `{:strategy ..., :history [...]}` ist. Unter `:history` werden die Aktionen (älteste zuerst) der anderen Partei angegeben.

Beispielaufrufe:

```
user=> (next-move {:strategy :copycat, :history []}
:coop
user=> (next-move {:strategy :copycat, :history [:coop :coop]}
:coop
user=> (next-move {:strategy :copycat, :history [:coop :betray]}
:betray
user=> (next-move {:strategy :snitch, :history [:coop :coop]}
:betray
```

Aufgabe 5 [14 Punkte]

(a) [5 Punkte] Definieren Sie die Begriffe simple und easy, und benennen Sie jeweils die Gegenteile.

(b) [9 Punkte] Was sind nach Mosely und Marks die zwei wichtigsten Ansätze, um Software zu verstehen? Wie funktionieren diese? Was ist die größte Komplexitätsquelle? Welche Auswirkungen hat diese Komplexitätsquelle auf die o.g. Ansätze?

Aufgabe 6 [9 Punkte]

Ziel der Aufgabe ist die Implementierung eines Macros der Form:

```
(nth-expr n-th expr0 expr1 ... exprk)
```

Der Aufruf des Macros soll die n-te der Expressions (`expr0` ... `exprk`) evaluieren, wobei n das Ergebnis der Evaluation vom Argument `n-th` ist. Alle anderen Argumente bleiben unevaluiert. Sie dürfen davon ausgehen, dass für alle Aufrufe $0 \leq n \leq k$ gilt. In der gesamten Aufgabe darf `eval` nicht verwendet werden.

Dazu soll der Aufruf

```
(nth-expr (inc 1) 0 "one" (println 2))
```

gegen folgenden Code expandiert werden:

```
(case (inc 1)
  0 0
  1 "one"
  2 (println 2))
```

Implementieren Sie das Macro.