Heinrich-Heine-Universität Düsseldorf
Institut für Informatik
Lehrstuhl Softwaretechnik und Programmiersprachen
Professor Dr. M. Leuschel

3. März 2017

Klausur

Funktionale Programmierung
Wintersemester 2016/17

Nachname: _____ Vorname: _____

Matrikelnummer: _____

Studienfach: _____ Semester: _____

Unterschrift: _____

Zugelassene Hilfsmittel: Eine beidseitig beschriebene oder bedruckte A4 Seite

Diese Klaujur enthält 13 nummerierte Seiten. Prüfen Sie bitte zuerst, ob alle Seiten vorhanden sind.

Schalten Sie bitte Ihr Mobiltelefon aus

Diesen Teil bitte nicht ausfüllen:

Aufgabe	1	2	3	4	5	6	7	Σ
Punktzahl	8	10	15	10	12	6	14	75
Erreicht								

Aufgabe 1 [8 Punkte]

Kreuzen Sie die richtige Antwort an. Bitte machen Sie klar deutlich, was Ihre Antwort ist. Nicht bearbeitete Fragen werden als falsch bewertet.
Bewertung:

```
(defn punkte [richtig nr_aufgaben]
   (let [p (* 2 (- richtig (/ nr_aufgaben 2)))]
     (max 0 p)))
```

Tipp: Sie können sich durch Raten **nicht** verschlechtern.

(a) Paredit ist
   A. wichtig für Code Golf.
   B. ein hilfreiches Tool in der Entwicklung mit einem Lisp.

(b) Leere Sequenzen sind in Clojure
   A. truthy.
   B. falsey.

(c) Laziness ist in Clojure
   A. im Ausführungsmodell implementiert.
   B. in den Datenstrukturen implementiert.

(d) Um gegenseitige Rekursion, die keine zusätzlichen Stackframes erzeugt, zwischen Funktionen zu ermöglichen, nutzt man
   A. `clojure.core/trampoline`
   B. `clojure.core/elevator`

(e) `concat` ist eine monadische Funktion der Sequenzmonade.
   A. Ja
   B. Nein

(f) `[`x# `x#]` kann ergeben:
   A. `[x__4976__auto__ x__4976__auto__]`
   B. `[x__4976__auto__ x__4977__auto__]`

(g) Im Namespace `user` ergibt `(let [x '(:a b)] `~x)`
   A. `(:a b)`
   B. `(:user/a b)`
   C. `(:a user/b)`
   D. `(:user/a user/b)`

(h) Die Funktionalität von Atomen kann man mit Refs abbilden, umgekehrt ist es aber nicht möglich.
   A. Ja
   B. Nein

Aufgabe 2 [10 Punkte]

Erklären Sie, was structural sharing ist. Nennen Sie dabei die wichtigsten Aspekte und geben Sie ein Beispiel.

Aufgabe 3 [15 Punkte]

(a) [6 Punkte] Schreiben Sie eine Funktion `take-until`, die ein Prädikat und eine Sequenz von Werten bekommt. Die Funktion soll den Präfix der Sequenz liefern, der als letztes Element einen Wert hat, der das Prädikat erfüllt, sofern dieser vorhanden ist. Ansonsten soll die Funktion die gesamte Sequenz zurückgeben. Die Funktion muss nicht lazy sein.

Beispielaufrufe:

```
user=> (take-until nil? [1 2 3])
(1 2 3)
user=> (take-until nil? [1 2 3 nil 5 6])
(1 2 3 nil)
user=> (take-until nil? [])
()
```

Beschreiben Sie *kurz* wie ihr Code funktioniert.

(b) [6 Punkte] Zu einer beliebigen natürlichen Zahl n lässt sich die sogenannte Collatz-Folge berechnen.

$$c(n) = \begin{cases} 3n+1 & \text{für } n \text{ ungerade} \\ n/2 & \text{für } n \text{ gerade} \end{cases} \quad (1)$$

Die Berechnung der Folge bricht ab, sobald ein Folgeglied 1 ist.

Schreiben Sie eine Funktion `collatz`, die zu einer gegebenen Zahl n die Sequenz `[n, c(n), c(c(n)) ...]` berechnet. Beispielsweise soll ein Aufruf so aussehen:

```
(collatz 11)
=> (11 34 17 52 26 13 40 20 10 5 16 8 4 2 1)
```

Hinweis: Sie dürfen `take-until` verwenden. Weiterhin können Sie annehmen, dass `take-until` lazy implementiert ist.

(c) [3 Punkte] Schreiben Sie eine `test.check` Property, die fehlschlägt, falls die Rückgabe von `collatz` nicht in $4, 2, 1$ endet. Sie dürfen annehmen, dass folgender Codeschnipsel bereits ausgeführt wurde:

```
(require '[clojure.test.check.generators :as gen])
(require '[clojure.test.check.properties :as prop])
```

Aufgabe 4 [10 Punkte]

Implementieren Sie ein Macro `debug-println`, welches eine Clojure Form als Argument nimmt. Diese Form und der Wert, zu dem sie evaluiert, sollen ausgegeben werden. Der Rückgabewert soll dieser Wert sein.

Achten Sie darauf, dass Seiteneffekte genau einmal ausgeführt werden.

```
user=> (debug-println (+ (* 3 4) 2))
(+ (* 3 4) 2) ==> 14  ;; print
14                    ;; Rueckgabewert

user=> (debug-println (println 2))
2                     ;; Seiteneffekt von (println 2)
(println 2) ==> nil   ;; print vom Macro
nil                   ;; Rueckgabewert
```

Beschreiben Sie *kurz* wie ihr Code funktioniert.

Aufgabe 5 [12 Punkte]

(a) [9 Punkte] Wir betrachten den Fall von `reduce`, der neben einer Funktion einen Startwert und eine nicht-leere Collection als Argument nimmt.

Beispielsweise `(reduce f init [0 1 2])` rechnet `(f (f (f init 0) 1) 2)` aus.

Implementieren Sie eine **Funktion** `foldr`, die auch eine zweistellige Funktion, einen Startwert und eine Collection als Argumente erhält und im Gegensatz zu `reduce` dann `(f 0 (f 1 (f 2 init)))` berechnet.

Beispielaufrufe:

```
user=> (reduce + 0 (range 10))
45
user=> (foldr + 0 (range 10))
45
user=> (reduce conj [] (range 3))
[0 1 2]
user=> (foldr (fn [e a] (conj a e)) [] (range 3))
[2 1 0]
```

Beschreiben Sie *kurz* wie ihr Code funktioniert.

(b) [3 Punkte]

- `(fn [x] (str x))` hat den Typ `a -> String`.
- `rest` hat den Typ `[a] -> [a]`.
- `(fn [x y] x)` hat den Typ `a -> b -> a`.

Welchen Typ hat `foldr`?

Aufgabe 6 [6 Punkte]

Gegeben seien folgende Spezifikationen:

```
(ns poker
  (:require [clojure.spec :as s]))

(def suit? #{:clubs :diamonds :hearts :spades})
(def rank? (into #{:jack :queen :king :ace} (range 2 11)))

(s/def ::card (s/tuple rank? suit?))
(s/def ::hand (s/+ ::card))

(s/def ::name string?)

(s/def ::player (s/keys :req [::name ::hand]))
```

Geben Sie eine Datenstruktur an, die mit `::player` konform geht.

Aufgabe 7 [14 Punkte]

Gegeben sei folgendes Atom:

```
(def state (atom {:buf []
                  :seen #{}
                  :cnt 0}))
```

(a) [4 Punkte] Schreiben Sie eine Funktion `insert!`, die `:buf` in `state` ein Element entweder konsequent vorne oder hinten hinzufügt. Der Rückgabewert der Funktion ist egal. Es wird nie `nil` eingefügt.

Beispiel:

```
user=> (deref state)
{:buf [], :seen #{}, :cnt 0}
user=> (do (insert! :item) (:buf (deref state)))
{:buf [:item], :seen #{}, :cnt 0}
```

(b) [7 Punkte] Schreiben Sie eine Funktion `process!`, die aus `:buf` in `state` das vorderste Element entfernt und in die `:seen` Menge einfügt. Wenn das Element vorher nicht in der Menge `:seen` enthalten war, so soll der Zähler `:cnt` um 1 erhöht werden. Sie dürfen annehmen, dass die Funktion nur aufgerufen wird, wenn `:buf` nicht leer ist. Weiterhin dürfen Sie davon ausgehen, dass `:buf` nicht `nil` enthält.

Beispiel:

```
user=> (deref state)
{:buf [:a :b], :seen #{:a}, :cnt 1}
user=> (do (process!) (deref state))
{:buf [:b], :seen #{:a}, :cnt 1}
user=> (do (process!) (deref state))
{:buf [], :seen #{:a :b}, :cnt 2}
```

(c) [3 Punkte] Was passiert, wenn die beiden Funktionen `insert!` und `process!` nun gleichzeitig ausgeführt werden? Was ist, wenn man es mit je einer Ref für jeden Wert implementiert hätte?