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

19. Februar 2018

Klausur

Funktionale Programmierung
Wintersemester 2017/18

Nachname: _____ Vorname: _____

Matrikelnummer: _____

Studienfach: _____ Semester: _____

Unterschrift: _____

Zugelassene Hilfsmittel: Eine beidseitig beschriebene oder bedruckte A4 Seite

Diese Klaujur enthält 14 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	8	Σ
Punktzahl	8	17	10	10	21	15	11	7	99
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.
Wählen Sie zunächst ein Bewertungsschema aus. Ist nicht erkennbar, welches Sie auswählen, wird das erste verwendet.

☐ (Raten verschlechtert nicht die Punkte):

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

oder

☐ (falsche Antworten führen innerhalb der Aufgabe zu Punktabzug):

```
(defn punkte [richtig falsch _]
  (max 0 (- richtig falsch)))
```

(a) Lisps sind Implementierungen
   A. des Lambda-Kalküls.
   B. einer deterministischen Turing-Maschine.

(b) `frequencies` ist
   A. lazy.
   B. nicht lazy.

(c) Vektoren sind
   A. lazy.
   B. nicht lazy.

(d) Die Funktionalität von `let` muss als Special Form implementiert sein.
   A. Wahr
   B. Gelogen

(e) Das Expression Problem kann man mit
   A. `do` lösen.
   B. Multimethoden lösen.

(f) Auf der **REPL** eingegeben ergibt `` `(+ 1 ~(* 2 3)) ``
   A.
```
(clojure.core/seq (clojure.core/concat
                   (clojure.core/list (quote clojure.core/+))
                   (clojure.core/list 1)
                   (clojure.core/list (* 2 3))))
```
   B. `(clojure.core/+ 1 6)`

(g) Wenn mehrere Threads denselben Agenten verwenden, kann es zu Kollisionen und Retries kommen.
   A. Wahr
   B. Gelogen

(h) Simplicity ist
   A. subjektiv.
   B. objektiv.

Aufgabe 2 [17 Punkte]

(a) [12 Punkte] Was ist das epochale Zeit Modell? Fertigen Sie eine Skizze an, die alle Komponenten beinhaltet, benennen Sie diese und beschreiben Sie kurz ihre jeweilige Funktion. Erklären Sie anhand dessen auch, was Zustand ist.

(b) [5 Punkte] Erläutern Sie, wie Atome (atoms) dieses Modell implementieren.

Aufgabe 3 [10 Punkte]

In einer Variante des Spieles LightBot geht es darum einen Roboter mit einem Programm zu steuern, um auf bestimmten Feldern das Licht einzuschalten. Dazu stehen unter anderem die Instruktionen `:forward`, `:switch-on` und `:jump` zur Verfügung.

Im Folgenden sollen Sie eine Funktion `(valid-action? action start ziel)` implementieren, die für Befehle entscheiden kann, ob sie erlaubt sind. Dazu erhält sie das der Instruktion entsprechende Keyword sowie Start- und Zielfeld als Argumente.

Felder sind wie folgt repräsentiert:

1. `nil`, wenn das Feld außerhalb des erlaubten Spielfeldes liegt.
2. Ansonsten als Map mit den Schlüssels `:level` und `:switch`, wobei unter `:level` die Ebene des Spielfeldes als natürliche Zahl und unter `:switch` boolesche Werte assoziiert werden können.

Es gelten folgende Regeln:

1. `:forward` ist genau dann erlaubt, wenn das Zielfeld innerhalb des erlaubten Spielfeldes liegt und die Ebenen vom Start- und Zielfeld gleich sind.
2. `:switch-on` ist genau dann erlaubt, wenn `:switch` im Startfeld `true` ist.
3. `:jump` ist genau dann erlaubt, wenn das Zielfeld innerhalb des erlaubten Spielfeldes liegt und sich die Ebenen vom Start- und Zielfeld um exakt Eins unterscheiden.

Implementieren Sie `valid-action?` für diese drei Befehle. **Achten Sie darauf, dass die Funktion um weitere Befehle erweitert werden kann.**

Beispielaufrufe:

```
user=> (valid-action? :forward {:level 1, :switch :false}
                               {:level 1, :switch :false})
true
user=> (valid-action? :forward {:level 1, :switch :false}
                               {:level 2, :switch :false})
false
user=> (valid-action? :forward {:level 1, :switch :false} nil)
false
user=> (valid-action? :jump {:level 1, :switch :false}
                            {:level 2, :switch :false})
true
```

Aufgabe 4 [10 Punkte]

Die Funktion `every-pred` nimmt eine oder mehrere Funktionen `fs` als Argumente und gibt eine Funktion zurück, die beliebig viele Argumente `args` nimmt und `true` liefert, wenn alle Funktionen aus `fs` auf alle Werte in `args` angewandt einen truthy Wert ergibt. Ansonsten wird `false` zurückgegeben.

Implementieren Sie `every-pred`.

Beispielaufrufe:

```
user=> (every-pred even?)
#object[user$every_pred$fn__1282 ...]
user=> ((every-pred even?) 2)
true
user=> ((every-pred even? #(< % 10))) ;; keine args
true
user=> ((every-pred even? #(< % 10)) 2 4 6 8)
true
user=> ((every-pred even? #(< % 10)) 2 4 6 8 10)
false ;; (#(< % 10) 10) ergibt false
user=> ((every-pred even? #(< % 10)) 2 4 5 6 8)
false ;; (even? 5) ergibt false
```

Aufgabe 5 [21 Punkte]

(a) [12 Punkte] Implementieren Sie eine Funktion `powerset`, die zu einer gegebenen Menge die Potenzmenge berechnet. Es ist egal, ob eine Sequenz von Sequenzen oder eine Menge von Mengen zurückgegeben wird, solange jedes Element nur einmal vorkommt. Ihre Lösung muss nicht auf Performance optimiert werden und muss nicht lazy sein.

Beispielaufrufe:

```
user=> (powerset #{})
#{#{}} ;; auch okay: [[]] oder (())
user=> (powerset #{:a :b})
#{#{}, #{:a}, #{:b}, #{:a :b}}
user=> (powerset #{:a :b :c})
#{#{}, #{:a}, #{:b}, #{:c} #{:a :b}, #{:a :c}, #{:b :c}, #{:a :b :c}}
```

Beschreiben Sie *kurz* wie ihr Code funktioniert.

Besteht eine Menge aus n Elementen, so besteht die Potenzmenge aus 2^n Elementen. Schreiben Sie eine `test.check` Property, die diese Eigenschaft für `powerset` überprüft.
Folgender Code wurde bereits ausgeführt:

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

Gehen Sie dazu wie folgt vor:

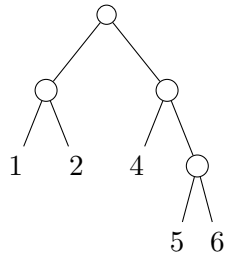
(b) [3 Punkte] Implementieren Sie eine Funktion `(pow base exponent)`, die $\text{base}^{\text{exponent}}$ berechnet. Sowohl `base` und `exponent` sind natürliche Zahlen.
Hinweis: Achten Sie hier auf Überläufe.

(c) [3 Punkte] Geben Sie einen Generator an, der Mengen erzeugt. Als Generator für einzelne Elemente eignet sich `gen/simple-type`.

(d) [3 Punkte] Geben Sie die oben beschriebene Property an.

Aufgabe 6 [15 Punkte]

Ein balancierter Baum ist ein Spezialfall eines Baumes, bei dem sich die Tiefe von je zwei Blättern um höchstens Eins unterscheiden. Der folgende Baum ist balanciert (und nur zufälligerweise sortiert):



(a) [3 Punkte] Geben Sie eine Repräsentation des obigen Baumes in Clojure an. Sie dürfen davon ausgehen, dass in Blättern nur Zahlen gespeichert werden.

Hinweis: Sie sollen mit dieser Repräsentation in den nächsten Teilaufgaben weiterarbeiten.

(b) [4 Punkte] Schreiben Sie eine Spec für das von Ihnen gewählten Format für einen Baum. Sie müssen *nicht* sicherstellen, dass der Baum balanciert ist. Sie dürfen wieder annehmen, dass in Blättern nur Zahlen gespeichert werden und dass bereits folgender Code ausgeführt wurde:

```
(require '[clojure.spec.alpha :as s])
```

(c) [8 Punkte] Schreiben Sie eine Funktion `balanced?`, die für einen Baum in der von Ihnen in (a) und (b) gewählten Repräsentation entscheidet, ob er balanciert ist. Wenn ja, soll von `balanced?` `true` zurückgegeben, ansonsten `false`. Ungültige Eingaben müssen nicht behandelt werden.

Beschreiben Sie *kurz* wie ihr Code funktioniert.

Aufgabe 7 [11 Punkte]

Das Macro `if-let` kriegt als Argument einen Vektor, dessen erster Eintrag ein Symbol und der zweite Eintrag ein Wert ist, sowie zwei Formen. Ist der Wert falsey, wird die zweite Form ausgeführt. Ist der Wert truthy, wird er an das Symbol gebunden und innerhalb dieses Kontexts wird die erste Form ausgeführt.

(a) [8 Punkte] Implementieren Sie `if-let`.

Hinweis: Achten Sie darauf, dass Seiteneffekte nur einmal ausgeführt werden. Den Fall ohne `else`-Zweig müssen Sie nicht implementieren.

Beispielaufrufe:

```
user=> (if-let [x 3] :true :false)
:true
user=> (if-let [x 3] x 0)
3
user=> (if-let [x nil] x 0)
0
user=> (if-let [x (do (println :hallo) 42)] x :false)
:hallo ;; print
42     ;; Rueckgabe
user=> (if-let [x nil] nil x)
CompilerException java.lang.RuntimeException:
Unable to resolve symbol: x in this context, compiling:(null:1:1)
```

Beschreiben Sie *kurz* wie ihr Code funktioniert.

(b) [3 Punkte] Kann man `if-let` auch (ohne Änderungen am Aufruf) als Funktion implementieren? Warum (nicht)?

Aufgabe 8 [7 Punkte]

Implementieren Sie eine Funktion `(transplace m)`, die eine Map `m` als Argument bekommt und einen Transducer zurückgibt. Wenn ein Element als Schlüssel in `m` vorhanden ist, soll es durch den assoziierten Wert ersetzt werden, ansonsten soll das originale Element verwendet werden.

Die Funktion `replace` darf dabei nicht benutzt werden.

Beispielaufrufe:

```
user=> (transduce (transplace {:y :a}) conj [:x :y :z])
[:x :a :z]
user=> (transduce (comp (transplace {nil 0}) (map inc))
                  conj
                  [42 nil 3])
[43 1 4]
user=> (transduce (comp (transplace {nil -1}) (partition-by pos?))
                  conj
                  [1 2 3 0 5 6])
[[1 2 3] [0] [5 6]]
```

Beschreiben Sie *kurz* wie ihr Code funktioniert.