Klausur Einführung in die Programmierung (I167) 13.03.2017
J. Brauer, S. Jasser, J. Sauer

Aufgabe 1 (6 Punkte)

	Bewerten Sie folgende Aussagen:	wahr	falsch
a)	Die Synatx legt die Struktur von Programmen, die Semantik ihre Funktionsweise fest.		
b)	Racket ist eine recht alte Programmiersprache, die auf Konzepten aus C aufbaut.		
c)	Funktionale Programme bestehen aus der Definition und dem Aufruf von Funktionen.		
d)	Bei der funktionalen Programmierung werden Ausdrücke ausgewertet, indem Teilausdrücke durch äquivalente, einfachere Ausdrücke ersetzt werden.		
e)	Eine Funktion besteht entweder aus einem atomaren Wert oder aus einem rekursiven Aufruf.		
f)	Eine Funktion höherer Ordnung bekommt eine Funktion als Argument übergeben.		

Eine richtige Antwort wird mit je 1 Punkt, jede falsche oder nicht gegebene Antwort mit 0 Punkten bewertet.

Aufgabe 2 (14 Punkte)

Für die folgenden Ausdrücke bzw. Ausdruckssequenzen geben Sie jeweils Wert und Typ des Ergebnisses an, das sich aus der Auswertung des jeweils letzten Ausdrucks ergibt. Sie brauchen nur die Endergebnisse – nicht deren Ableitung – angeben.

Für den **Typ** des Ergebnisses genügen Angaben wie „Zahl“, „Symbol“ oder dergleichen. Falls es sich beim Ergebnis um eine primitive (eingebaute) Funktion handelt, schreiben Sie als Wert „primitive Funktion“, falls es sich um eine benutzerdefinierte Funktion handelt, schreiben Sie einfach „Funktion“. Für den **Typ** einer Funktion geben Sie den Vertrag z.B. `(list-of any → boolean)` an.

Ausdruck	Wert	Typ
(> 4 5)	#false	Boolean
(lambda [x] (* x x)	Funktion	Zahl → Zahl

Hinweis: Alle Ausdrücke lassen sich ohne Fehler auswerten.

2.1 (2 Punkte):

```
(let [(* +)
      (+ /) ]
(* (+ 2 2) 1))
```

Wert:

Typ:

2.2 (3 Punkte):

```
(define x 4)
(define y 8)
(define f
      (lambda [x]
            (let [(y 24)]
                  (/ y x))))
(f 8)
```

Wert:

Typ:

2.3 (4 Punkte):

```
((lambda [u <= v] (<= v u)) 3 > 4)
```

Wert:

Typ:

2.4 (5 Punkte):

```
((lambda [x]
      (lambda [y]
            (cons (* x x) y)))
```

Wert:

Typ:

Aufgabe 3 (6 Punkte)

Gegeben sei die folgende Funktionsdefinition:

```
(define f
      (lambda [n m]
            (cond
                  [(< n m) n]
                  [else (f (-n m) m)])))
```

3.1 (1 Punkt): Was berechnet die Funktion `f`?
3.2 (5 Punkte): Werten Sie den folgenden Ausdruck (durch Anwendung des Ersetzungsmodells) **Schritt für Schritt** aus: `(f 5 3)`

Aufgabe 4 (9 Punkte)

4.1 (6 Punkte): Entwickeln Sie unter Anwendung der Regeln für listenverarbeitende Funktionen die folgende Funktion: Die Funktion `entferne` liefere, angewendet auf eine Liste von Symbolen `l` und ein Symbol `s`, eine Liste mit allen Symbolen aus `l`, wobei jedes Auftreten von `s` entfernt wurde.
Beispiele:

```
(entferne '( a b a c ) 'a) => '( b c )
(entferne '( a b a c ) 'd) => '( a b a c )
```

4.2 (3 Punkte): Mit welcher bekannten Funktion höherer Ordnung kann die Funktion `entferne` implementiert werden? Formulieren Sie Ihre Implementierung unter Anwendung der gewählten Funktion höherer Ordnung!
Hinweis: Die Funktion `not` negiert einen booleschen Wert.

Aufgabe 5 (17 Punkte)

Hinweis: Für die Funktionen dieser Aufgabe müssen Sie keine Tests angeben!

5.1 (1 Punkt): Legen Sie eine Datenstrukturdefinition `(define-struct)` für Kalenderdaten an, die aus drei ganzen Zahlen für den Tag, den Monat und das Jahr besteht. Ein korrektes Datum enthält ein positives Jahr, einen Monat zwischen 1 und 12 und einen Tag nicht größer als 31 (oder weniger, abhängig vom Monat).

Achtung: Die im Folgenden zu entwerfenden Funktionen müssen nur korrekte Kalenderdaten für das richtige Resultat liefern, d.h. sie müssen die Korrektheit eines Datums nicht prüfen.

5.2 (4 Punkte): Schreiben Sie eine Funktion `ist-frueher?`, die zwei Kalenderdaten akzeptiert und einen booleschen Wert liefert. Sie liefert `#true`, wenn das erste Datum zeitlich vor dem zweiten liegt. Wenn beide Kalenderdaten gleich sind, ist das Ergebnis `#false`.

Beispiele:

```
(ist-frueher? (make-datum 2016 2 28)
              (make-datum 2017 2 28)) => #true
(ist-frueher? (make-datum 2017 2 28)
              (make-datum 2017 2 28)) => #false
```

5.3 (6 Punkte): Schreiben Sie eine Funktion `kalenderdaten-in-monat`, die eine Liste von Kalenderdaten und einen Monat (ganze Zahl) akzeptiert. Sie liefert eine Liste mit denjenigen Kalenderdaten aus der Argumentliste, die in dem gegebenen Monat liegen. Die Reihenfolge der Kalenderdaten in der Ergebnisliste ist unerheblich.

Beispiel:

```
(kalenderdaten-in-monat (list (make-datum 2017 2 28)
                              (make-datum 2017 3 28)
                              (make-datum 2017 2 26))
                        2)
=> (list (make-datum 2017 2 28) (make-datum 2017 2 26))
```

5.4 (6 Punkte) Schreiben Sie eine Funktion `kalenderdaten-in-monaten`, die eine Liste von Kalenderdaten und eine Liste von Monaten (Liste ganzer Zahlen) akzeptiert. Sie liefert eine Liste

bestehend aus den Kalenderdaten aus der Argumentliste, die in irgendeinem Monat aus der Monatsliste liegen. Die Reihenfolge der Kalenderdaten in der Ergebnisliste ist unerheblich.

Hinweise:
- Benutzen Sie die Lösung des vorangegangenen Aufgabenteils!
- Sie fürfen davon ausgehen, dass kein Monat in der Monatsliste mehrfach vorkommt.

Beispiel:
```
(kalenderdaten-in-monaten (list (make-datum 2017 2 28)
                                (make-datum 2017 3 28)
                                (make-datum 2017 2 26)
                                (make-datum 2017 1 26))
                          '(2 3))
=> (list (make-datum 2017 2 28)
         (make-datum 2017 2 26)
         (make-datum 2017 3 28))
```

Aufgabe 6 (10 Punkte)

Gegeben sei folgende Funktion:
```
(define f
      (lambda [n]
            (cond
                  [(= n 1) 1]
                  [else (+ (+ n n) (f (- n 1)))])))
```

Beweisen Sie mittels rekursiver Induktion, dass der Aufruf `(f n)` für jede natürliche Zahl (n >= 1) den Wert 1/6 n (n+1) (2n+1) liefert.

Aufgabe 7 (12 Punkte)

7.1 (5 Punkte): Analog zur Vorlesung ist ein Produkt wie folgt definiert:
```
;; Ein Produkt ist ein Record
(define-struct produkt [name preis])
```

Definieren Sie eine Funktion `namen-teurer-produkte`, die einen Preis (Zahl) und eine Produktliste als Eingabe erwartet und eine Liste mit den Namen derjenigen Produkte zurückliefert, die teurer sind als der angegebene Schwellwert.

Hinweise:
- Nutzen Sie die bekannten Funktionen höherer Ordnung!
- Falls sinnvoll, nutzen Sie lokale Definitionen zur Verbesserung der Lesbarkeit!
- Sie müssen für diese Funktion keine Tests angeben.

Beispiel:
```
(namen-teurer-produkte 150
                       (list (make-produkt 'bund-bananen 150)
                             (make-produkt 'kaffee 699)
                             (make-produkt 'edel-vollmilch 259)
                             (make-produkt 'apfel 50)
                             (make-produkt 'fischfilet 359)))
=> '(kaffee edel-vollmilch fischfilet)
```

7.2 (5 Punkte): Schreiben Sie eine Funktion `flip`, die eine Funktion mit drei Paramentern als Argument erwartet. `flip` soll eine neue Funktion zurückliefern, die die drei Paramenter der Eingabefunktion in umgekehrter Reihenfolge erwartet!

Beispiel:

```
((flip (lambda [x f y] (f x y)) 2 expt 3) => 9 (=3²)
```

Hinweis: Sie müssen für diese Funktion keine Tests angeben.

7.3 (2 Punkte) Begründen Sie, warum `flip` eine Funktion höherer Ordnung ist.

Aufgabe 8 (16 Punkte)

Für die in dieser Aufgabe zu entwicklenden Funktionen für natürliche Zahlen bilden die bekannten Peano-Axiome die mathematische Grundlage.

Mit den Peano-Axiomen werden natürliche Zahlen und ihre Eigenschaften beschreiben. n´ ist dabei definiert als der Nachfolger von n.

Peano-Axiome:

1. $0 \in \mathbb{N}$
2. $\forall n \in \mathbb{N}: n' \in \mathbb{N}$
3. $\forall n \in \mathbb{N}: n' \neq 0$
4. $\forall n,m \in \mathbb{N}: n' = m' \Rightarrow n = m$
5. $\forall M: (0 \in M \land (\forall n \in \mathbb{N}: n \in M \Rightarrow n' \in M)) \Rightarrow N \subseteq M$

Auf dieser Grundlage können natürliche Zahlen in Racket wie folgt rekursiv definiert werden:

```
;; Eine natürliche Zahl (Nat) ist entweder
;; 1. zero oder
;; 2. (succ n), wenn n eine natürliche Zahl ist.
(define zero 'zero)

;; Die Funktion succ gibt den Nachfolger der natürlichen Zahl n zurück
;; succ: Nat -> Nat
(check-expect (succ zero) (succ zero))
(check-expect (succ (succ zero)) (succ (succ (zero))))

(define succ
      (lambda [n]
            (list 'succ n)))
```

Ihnen stehen außerdem folgende Funktionen zur Verfügung:

- Das Prädikat `zero?` Prüft, ob eine natürliche Zahl `zero` ist oder nicht.
- Die Funktion `plus`, die zwei natürliche Zahlen addiert.
  Für diese Funktion gilt mit n, m ∈ Nat:

```
plus(zero n) = n                                    (1)
plus(succ(n),m) = succ(plus(n,m))                   (2)
```

Definieren Sie folgende Funktionen unter Anwendung aller bekannten Regeln (also mit Beschreibung, Vertrag und Tests):

8.1 (4 Punkte): Eine Funktion `pred`, die den Vorgänger einer natürlichen Zahl zurückgibt.

8.2 (6 Punkte): Eine Funktion `mul`, die basierend auf der bereits definierten Addition (die Funktion `plus`) die Multiplikation umsetzt.

8.3 (6 Punkte): Diskutieren Sie die Möglichkeit, rationale Zahlen (Brüche) auf der Grundlage der oben definierten natürlichen Zahlen darzustellen.
Beschreiben Sie wie eine Addition dieser rationalen Zahlen erfolgen kann.

Hinweis: Sie brauchen die Funktionen nicht zu implementieren.