TU Darmstadt — Fachbereich Informatik — Telekooperation / Rechnerbetriebsgruppe

Zwischenklausur

Grundlagen der Informatik 1 – Sommer 2009

03. 06. 2009 – 16:15 - 18:15 Uhr

Hinweise:

- Als Schreibmittel ist nur ein schwarzer oder blauer Schreibstift erlaubt.
- Als Hilfsmittel können Sie unsere Folien, Übungen, sonstige Notizen, Bücher, Wörterbücher und sonstige Literatur benutzen.
- Füllen Sie das Deckblatt vollständig aus!
- Schreiben Sie auf jedes Aufgabenblatt Ihren Namen und Matrikelnummer.
- Schreiben Sie Ihre Lösung in die vorgesehenen Zwischenräume oder auf die Rückseite des jeweiligen Aufgabenblattes.

Nachname	
Vorname	
Matrikelnr.	
Studiengang	
Semester	

Aufgabe	1	2	3	4	5	6	$\sum$
Erreichbar	15	20	15	20	15	15	100
Punkte							

1 Scheme-Syntax (15 Punkte)

1.1 Scheme-Ausdrücke (3 × 1 Punkt)

Berechnen Sie das Ergebnis folgender Scheme-Ausdrücke. Gehen Sie dabei nach der **applikativen Auswertungsreihenfolge** vor und ersetzen Sie in jedem Schritt **genau einen Ausdruck**. Wenn ein Ausdruck wegen eines Fehlers nicht weiter ausgewertet werden kann, geben Sie an dieser Stelle *Error* sowie kurz den Grund dafür an.
Beispiel:
`(* 3 (* 4 8))` ➔ `(* 3 32)` ➔ `96`

- `(+ (- 3 7) (+ 4 8))`

- (/ (∗ 2 8) 4)

- (= (**max** 7 3) (**min** 3 7))

1.2 Listenausdrücke (3 × 2 Punkte)

Berechnen Sie das Ergebnis folgender Scheme-Ausdrücke. Gehen Sie dabei nach der **applikativen Auswertungsreihenfolge** vor und ersetzen Sie in jedem Schritt **genau einen Ausdruck**. Wenn ein Ausdruck wegen eines Fehlers nicht weiter ausgewertet werden kann, geben Sie an dieser Stelle *Error* sowie kurz den Grund dafür an.
list - oder *Quote*-Ausdrücke müssen *nicht* weiter zu *cons* umgeformt werden. Anstelle von *cons* dürfen Sie auch mit äquivalenten **list** - oder Quote-Ausdrücken arbeiten.
Beispiel: (**rest** (**rest** (**cons** a (**cons** b **empty**))))
➔ (**rest** (**cons** b **empty**))
➔ **empty**

- (**first** (**rest** '(1 a 2)))

- (**rest** (**first** '(a b c)))

- (**rest** (**list** (**rest** '(3 7 a)) 2))

1.3 Map-Ausdrücke (3 × 2 Punkte)

Verfahren Sie wie in der vorherigen Teilaufgabe. Sie dürfen **map** *gleichzeitig* auf alle Listenelemente anwenden und auch für alle Elemente das Ergebnis synchron berechnen. Denken Sie aber daran, die Einzelschritte der Berechnung (sofern es welche gibt) auch separat anzugeben!
Beispiel: (**map** odd? '(1 3 4))
➔ (**list** (odd? 1) (odd? 3) (odd? 4))
➔ (**list true true false**)

- (**map** ∗ '(7 3) (**list** 2 4))

- (**map** <= '(1 2) '(2 1))

- (**first** (**map sqr** '(7 2 3)))

2 Strukturelle Rekursion (20 Punkte)

2.1 Prozeduren verstehen (5 Punkte)

Betrachten Sie die folgende Prozedur g.

```
(define (g a b c)
  (if (empty? a)
      empty
      (if (and (>= (first a) b)
               (<= (first a) c))
          (cons (first a) (g (rest a) b c))
          (g (rest a) b c))
      ))
```

1. (1 Punkt) Was ergibt (g '(1 3 5) 2 5)?

2. (1 Punkt) Was ergibt (g '(5 7 1 2) 6 4)?

3. (3 Punkte) Geben Sie nun den *Vertrag*, *Zweck*, *Beispiel* und *zwei Tests* für g an.

2.2 Noten berechnen (15 Punkte)

Gegeben seien die unten stehenden Strukturen zur Repräsentation von Studierenden und Klausurpunktzahlen sowie zwei Begriffsdefinitionen zur Vereinfachung.
Implementieren Sie eine Prozedur *evaluate*. Diese konsumiert einen *record* (*siehe Zeile 8-9 im Code unten*) und produziert einen *exam-record* der gleichen Länge (0, 1 oder entsprechend der Länge der übergebenen Liste). Wird also eine Liste von drei Einträgen übergeben, ist das Ergebnis wiederum eine Liste mit drei Einträgen. Wird *evaluate* mit einer Instanz der Struktur *exam* aufgerufen, soll eine Instanz von *result* produziert werden.
Vervollständigen Sie nun die untenstehende Definition. Geben Sie mindestens zwei Tests an, für die Sie auch dave und john nutzen können.

```
1  ;; an exam is a tuple of student id, student name,
2  ;; and points for tasks 1 to 3
3  (define-struct exam (id name t1 t2 t3))
4  ;; a res is a tuple of student id, student name,
5  ;; and sum of points
6  (define-struct res (id name sum))
7
8  ;; A record is either an exam or a (listof exam)
9  ;; An exam-record is either a res or a (listof res)
10
11 ;; Example data
12 (define dave  (make-exam 1 "Dave V" 3 5 12))
13 (define john  (make-exam 3 "John W' 0 5 8))
14
15 ;; evaluate: record -> exam-record
16 ;; determines the exam result(s) for the student record
17 ;; or list of student records passed in
18 ;; example: (evaluate (make-exam 1 "X" 1 2 3)) is
19 ;; (make-res 1 "X" 6)
20 (define (evaluate r)
```

3 Nachkorrektur in einer Klausur (15 Punkte)

Schreiben Sie eine Funktion `reevaluate`, die eine Liste von Studierenden erhält und daraus diejenigen Studierenden bestimmt, deren Klausur für eine Nachkorrektur in Frage kommt. Studenten bestehen aus der *Punktzahl* und einem *Fachbereich* (engl. department, dept.). In Frage für eine Nachkorrektur kommen Studierende, die eines der beiden Kriterien erfüllen:

- Der Studierende hat den Fachbereich 20 und 45-49.5 Punkte erzielt, oder
- der Studierende gehört zum Fachbereich 1 und hat 27.5-29.5 Punkte erzielt.

Implementieren Sie die Prozedur reevaluate nach einem Ansatz Ihrer Wahl. Die Deklaration der Struktur *student* ist wie folgt:

```
1 (define-struct student (score dept))
```

Denken Sie an Vertrag, Zweck, Beispiel und mindestens zwei Tests für alle Hauptprozeduren! Hilfsprozeduren, die *local* oder mit **lambda** definiert sind, brauchen Sie *nicht zu dokumentieren*.
Hinweis: Mit der richtigen „Eingebung“ können Sie das Problem sehr kompakt lösen.
Hinweis: Diese Regelung galt nur für die Zwischenklausur im Wintersemester 2008/2009.

4 Komplexität (20 Punkte)

1. (4 P.) Geben Sie für die Behauptung, dass $f(n) = 3 \cdot n^2 + 7$ zur Komplexitätsklasse $O(n^2)$ gehört, geeignete Werte für n_0 und c an.

2. (6 P.) Bestimmen Sie die *Komplexitätsklasse* der folgenden Funktion:
   $f(n) = 7 \cdot n^3 + 21 \cdot n^2 - 13$. Vergessen Sie nicht die Angabe von c, n_0!

3. (10 P.) Notieren Sie in folgender Tabelle, welche Funktion f (n) zu welcher Komplexitätsklasse gehört. Tragen Sie dazu in jedes Kästchen ein, ob die entsprechende Funktion zu der jeweiligen Komplexitätsklasse gehört (**T**) oder nicht (**F**). So soll in der ersten freien Zelle der ersten Zeile ein T stehen, wenn $5 \cdot n^3 \in O(1)$ und ein F, wenn $5 \cdot n^3 \notin O(1)$. Als Beispiel wurde die Spalte für die Funktion $f(n) = \sin(n)$ bereits ausgefüllt.

   Hinweis: Sie erhalten in dieser Teilaufgabe für jedes korrekt eingetragene T oder F 0.5 Punkte. Für jeden *falschen* Eintrag werden 0.5 Punkte **abgezogen**. Eine negative Gesamtpunktzahl wird als 0 Punkte gewertet.

   Wenn Sie sich nicht sicher sind, lassen Sie das Feld frei!

$f(n) \rightarrow$ **Komplexitätsklasse** $\downarrow$	$5 \cdot n^3$	$3 \cdot n^2 - 5 \cdot n$	$3 \cdot 10^7$	$4 \cdot n \cdot \log_{10} n$	$\sin(n)$
$O(1)$					T
$O(n^2)$					T
$\Omega(n^2)$					F
$\Theta(n^2)$					F
$O(n^3)$					T

5 Pascal'sches Dreieck (15 Punkte)

Implementieren Sie eine Prozedur pascal−line, die eine Zeile des Pascal'schen Dreiecks berechnet. Jede Zeile n im Pascal-Dreieck, beginnend ab Zeile 0, hat die Länge $n + 1$.

- Die Pascal-Zeile für n $= 0$ ist eine Liste mit dem einzigen Element 1.
- Die Pascal-Zeile für $n > 0$ hat die Länge $n + 1$ und entsteht aus der Pascal-Zeile für $n - 1$ durch die Regel $p_i(n) = p_{i-1}(n - 1) + p_i(n - 1)$. Dabei steht i für die Position innerhalb der Zeile, beginnend ab $i = 1$. Für $p_0(n - 1)$ ist der Wert 0 anzusetzen.

Die folgende Tabelle gibt die Werte des Dreiecks wieder für die Zeilen 0-4. Jeder Wert in einer gegebenen Spalte entsteht durch die Summe des Wertes direkt darüber und links schräg darüber. Falls der Wert *links darüber* nicht existiert, ist der Wert 0 dafür anzunehmen.

Zeile 0:	1				
Zeile 1:	1	1			
Zeile 2:	1	2	1		
Zeile 3:	1	3	3	1	
Zeile 4:	1	4	6	4	1

Implementieren Sie nun eine Prozedur pascal−line, die einen Wert $n \geq 0$ konsumiert und die Pascal-Zeile n produziert. Vergessen Sie nicht die Angabe von *Vertrag, Zweck* und *Beispiel* für jede von Ihnen geschriebene Prozedur. Für *nicht-lokale* Prozeduren sind zusätzlich mindestens jeweils zwei Tests anzugeben.
Hinweis: Bei der richtigen „Eingebung" können Sie das Problem mit *map* sehr elegant und kurz lösen. Alternativ sollten Sie über die Nutzung eines *Akkumulators* nachdenken!

6 Akkumulatoren (15 Punkte)

Unten finden Sie die Definition einer Prozedur *g*, die drei Parameter *a, b* und *c* konsumiert. Zur Erinnerung: (remainder x y) berechnet den Rest bei der Division von x durch y.

1. (3 P.) Welches Ergebnis produziert der Aufruf (g (**list** 7 2 3) 10 7)? Bitte geben Sie sowohl das Endergebnis als auch die Werte des Parameters *d* der lokalen Prozedur h an, in der Notation *d = 9* ➔ *d = 13* ➔ *d = 25* (diese Zahlen sind *falsch*!).

2. (3 P.) Welches Ergebnis produziert der Aufruf (g (**list** 3 1 7) 8 3)? Bitte machen Sie die gleichen Angaben wie im ersten Aufgabenteil.

3. (3 P.) Geben Sie nun den Vertrag, die Beschreibung und ein Beispiel für g und h an. Sie können die Angaben direkt in die Lücken im Code eintragen.

```
;; g :
;;
;; purpose:
;;
;;
;;
;; example:
;;
(define (g a b c)
  (local
    ;; h :
    ;;
    ;; purpose:
    ;;
    ;;
    ;;
    ;; example:
    ;;
    ((define (h a b c d)
       (if (empty? a) (remainder d c)
           (h (rest a) b c
              (+ (* d b) (first a))))))
    (h a b c 0)))
```

4. (6 P.) Bitte implementieren Sie eine Variante von *g*, die das gleiche Verhalten zeigt, aber anstelle von Rekursion oder Akkumulatoren *ausschließlich* die Funktionen **map**, **foldl**, **foldl** oder **filter** nutzt. Die an diese Funktionen übergebene Funktion soll als *Lambda-Ausdruck* implementiert werden.

   Auf die Angabe des Vertrags, Zwecks, Beispiels und Tests dürfen Sie verzichten, da diese identisch zu den obigen sein können sollten.