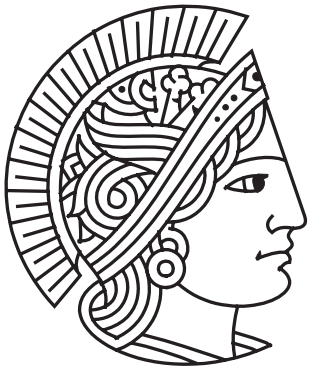
TU Darmstadt — Fachbereich Informatik — Telekooperation / Rechnerbetriebsgruppe



Klausur

Grundlagen der Informatik 1 – WS 2008/09

01.12.2008 – 18:10 - 20:10 Uhr

-A-

Hinweise:

- Als Schreibmittel ist nur ein schwarzer oder blauer Schreibstift erlaubt.
- 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	$\sum$
Punkte						
Maximum	15	25	20	15	25	100

1 Scheme Syntax (15P)

1.1 Arithmetische Ausdrücke (3 × 1P)

Schreiben Sie folgende arithmetische Ausdrücke in Scheme-Syntax. Verwenden Sie dabei nur Scheme Funktionen, die in der Vorlesung oder Übung vorkamen. Die Ausdrücke sollen *nicht* vereinfacht werden.

Beispiel: `3 * 4` ➔ `(* 3 4)`

- $(3 + 4)/10$ ➔

- $(1 + \sqrt{5})/2$ ➔

- $6 > 2 * 3$ ➔

1.2 Scheme Ausdrücke (4 × 1.5P)

Berechnen Sie das Ergebnis folgender Scheme-Ausdrücke. Gehen Sie dabei nach der **applikativen Auswertungsreihenfolge** vor und ersetzen Sie Schritt für Schritt **genau einen Ausdruck**. Wenn ein Ausdruck wegen eines Fehlers nicht weiter ausgewertet werden kann, geben Sie kurz den Grund dafür an.

Beispiel 1:

```
(+ (first '(1 2 3)) 1)
➔  (+ 1 1)
➔  2
```

Beispiel 2:

```
(+ (first '(a b c)) 1)
➔  (+ 'a 1)
➔  FEHLER: + erwartet Zahlen als Argument, 'a ist ein Symbol
```

- (+ (+ 3 4) 6 7)

- (rest '(1 3 7))

- (first (cons 3 empty))

- (rest (cons 2 (list 4)))

1.3 Scheme Ausdrücke II (3 × 2P)

Berechnen Sie folgende Ausdrücke wie in der vorherigen Aufgabe.

- (rest (first (cons 'a (list 'b 'c))))

- (map list '(1 2))

- (rest (cons a '(empty)))

2 Scheme Funktionen (25P)

2.1 Listenelemente Multiplizieren (4P)

Auf Ihrer Wunschliste steht die Funktion `mult`, die das Produkt einer Liste von Zahlen berechnen soll. Ergänzen Sie den Vertrag von `mult` (1P). Ergänzen Sie die Implementierung von `mult`, **ohne map oder fold zu verwenden** (3P). Gehen Sie davon aus, dass die Eingabewerte dem Vertrag entsprechen!

```
;; contract:




;;purpose: multipliziert die Zahlen in einer Liste
;;example: (mult '(1 2 3)) -> 6

(define (mult




;; TEST
(mult '(2 3 4))
;; should be
24
```

2.2 Strukturelle Rekursion I (5P)

Ergänzen Sie für folgende Funktion `f` Vertrag (1P), Beschreibung (2P), Beispiel (1P) und einen Testfall (1P):

```
;; contract:




;; purpose:




;; example:




(define (f alox)
  (if (empty? alox)
      empty
      (append
       (f (rest alox))
       (list (first alox)))))


;; TEST




;; should be
```

2.3 Strukturelle Rekursion II (8P)

Ergänzen Sie für folgende Funktiong Vertrag (1P), Beschreibung (4P), Beispiel (1P) und die Sollwerte für beide Testfälle (je 1P). Wenn ein Testfall zu einem Fehler führt, geben Sie als Sollwert ERROR an.

```
;; contract:




;; purpose:




;; example:




(define (g alox)
  (local (
    (define (f x) (list x x)))
    (if (empty? alox)
        empty
        (append
         (f (first alox))
         (g (rest alox))))))
;; TEST
(g '(1 2 3 4))
;; should be




;; TEST
(g '(1 2 4))
;; should be
```

2.4 Lambda Ausdrücke (8P)

Ergänzen Sie für folgende Funktion curry Vertrag (3P), Beschreibung (3P) und die Sollwerte für beide Testfälle (je 1P). Wenn ein Testfall zu einem Fehler führt, geben Sie als Sollwert ERROR an.

```
;; contract:




;; purpose:




;; example: (((curry *) 3) 4) -> 12

(define (curry fun)
  (lambda (m) (lambda (n) (fun m n))))

;; TEST
(((curry +) 1) 2)
;; should be




;; TEST
(((curry +) 2) 3 4)
;; should be
```

3 Map und Fold (20P)

3.1 Vektoraddition (5P)

Implementieren Sie die Funktion `vec-add`, die zwei Vektoren (gleich lange Listen von Zahlen) konsumiert und deren Summe (die komponentenweise Addition) zurückliefert. **Verwenden Sie map** (2P). Geben Sie Vertrag (1P), Beispiel (1P) und einen Testfall (1P) an.

```
;; contract:




;; purpose: addiert zwei Vektoren
;; example:




(define (vec-add




;; TEST




;; should be
```

3.2 Foldl (5P)

Ergänzen Sie für folgende Funktion `f` Vertrag (1P), Beschreibung (2P), Beispiel (1P) und einen Testfall (1P):

```
;; contract:




;; purpose:




;; example:




(define (f alox)
  (foldl (lambda (a akk) (+ 1 akk)) 0 alox))

;; TEST




;; should be
```

3.3 Map-Fun (10P)

Die Funktion `map-fun` erhält als Parameter eine Liste von Funktionen $f_1 ... f_n$ und ein Argument e. `map-fun` wendet jede Funktion auf e an und gibt eine Liste von Funktionswerten $f_1(e) ... f_n(e)$ zurück.

Beispiel: `(map-fun (list odd? even? succ) 5)` ergibt `'(true false 6)`. Die Funktionen `odd?`, `even?` und `succ` sind in Scheme eingebaut bzw. im Skript definiert und überprüfen, ob eine Zahl ungerade (`odd?`) oder gerade (`even?`) ist, bzw. addieren 1 zu einer Zahl hinzu (`succ`).

3.3.1 Verwendung von Map-Fun (3P)

Betrachten Sie die Funktion `make-pair`:

```
;; contract: make-pair: X -> (Y -> (list X Y))
;; purpose: gibt eine Funktion zurück die ein Element konsumiert (second-element)
;;          und eine Liste zurückgibt, die aus first-element und second-element
;;          besteht.
;; example: (make-pair 1) 2) -> '(1 2)
(define (make-pair first-element)
  (lambda (second-element) (list first-element second-element)))

;; TEST
(make-pair 'a) 'b)
;; should be
'(a b)
```

Berechnen Sie das Ergebnis des folgenden Scheme Ausdrucks:

```
(map-fun
 (map make-pair '(1 2 3))
 'a)
```

3.3.2 Implementierung von Map-Fun (7P)

Geben Sie den Vertrag von `map-fun` an (3P), ergänzen Sie ggf. weitere Definitionen, die Sie zur Angabe des Vertrags benötigen. Beschreibung und Beispiel für die Funktion sind bereits gegeben. Einen Test brauchen Sie für diese Aufgabe **nicht** anzugeben.

```
;;  contract:
```

```
;; purpose:  wendet jede Funktion aus list-of-f auf X an
;;           und gibt die Liste der Funktionswerte zurück.
;; example:  (map-fun (list - list succ) 3) -> (list -3 (list 3) 4)
```

Implementieren Sie die Funktion `map-fun`, **ohne Rekursion zu verwenden; verwenden Sie stattdessen map** (4P).

```
(define (map-fun
```

4 O-Notation (15P)

1. **(2 P)** Was müssen Sie beweisen, um zu zeigen, dass $f(n) \in O(g(n))$?

2. **(3 P)** Beweisen Sie, dass $f(n) = 4n + 2$ zur Komplexitätsklasse $O(n)$ gehört, indem Sie geeignete Werte für n_0 und c angeben.

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 $2^n \in O(n^2)$ und ein F, wenn $2^n \notin O(n^2)$. Als Beispiel haben wir die Spalte für die Funktion $f(n) = sin(n)$ bereits ausgefüllt.

   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** $\downarrow$	2^n	$14 \cdot n + 7$	17	$3 \cdot n^2$	$\sin(n)$
$O(n^2)$					T
$\Omega(n \cdot log(n))$					F
$O(n)$					T
$O(n^n)$					T
$\Theta(n)$					F

5 Logarithmus Naturalis (25P)

Betrachten Sie die Zahlenfolge

$$a_n = n(\sqrt[n]{x} - 1)$$

Die Folge liefert mit größerem n immer bessere Näherungswerte für den Logarithmus Naturalis (ln) einer Zahl x. Je größer n ist, desto näher liegt a_n also am tatsächlichen Wert $ln(x)$. Gesucht ist nun das kleinste $n_0 \in \mathbb{N}$, so dass die Differenz zwischen a_{n_0} und a_{n_0+1} unter einer vorgegebenen Schranke `diff` liegt (d.h. $|a_{n_0} - a_{n_0+1}| < diff$).

5.1 Close-enough (5P)

Implementieren Sie die Prozedur `close-enough?`, die überprüft, ob sich zwei Werte im Betrag um weniger als `diff` unterscheiden (4P). Geben Sie den Vertrag (1P) an. Beschreibung und Beispiel sind bereits vorhanden. **Auf die Angabe von Testfällen können Sie verzichten**.

Hinweise:

- Der Aufruf (abs x) berechnet den Wert $|x|$.
- Die Konstante `diff` ist bereits definiert.

```
;;global definierte Schranke
(define diff 0.1)

;; contract:




;; purpose: überprüft ob sich zwei Werte weniger als diff unterscheiden
;; example: Angenommen, diff = 0.1: (close-enough? 3 3.01) -> true

(define (close-enough? a b)
```

5.2 Approx-ln (20P)

Implementieren Sie die Prozedur `approx-ln`, die eine Zahl x konsumiert und a_{n_0+1} zurück gibt (a_n und n_0 sind oben definiert). Verwenden Sie die Funktion `close-enough?`. Sie können auch weitere eigene Hilfsfunktionen definieren.

- Die Funktion `root` ist bereits wie unten angegeben definiert. `(root n x)` berechnet den Wert $\sqrt[n]{x}$.
- Verwenden Sie **kein local**!
- Geben Sie für **alle** von Ihnen definierten Funktionen Vertrag und Beschreibung an!
- Fehlende Verträge oder Beschreibungen führen zu einem Abzug von jeweils einem Punkt pro Funktion.
- Sie brauchen keine Testfälle oder Beispiele anzugeben.
- Sie können von eiener korrekten Eingabe $x > 0$ ausgehen.

```
;;contract: num num -> num
;;purpose: berechnet die n-te Wurzel von x
;;example: (root 3 27) -> 3
(define (root n x)
 (expt x (/ 1 n)))
```