
PROBE-. KLAUSUR ZU 'SOFTWAREENTWICKLUNG III:
FUNKTIONALE PROGRAMMIERUNG'
WS 2009/2010
LEONIE DRESCHLER-FISCHER

Note: _____ und Begründung:

Hamburg, _____ Unterschrift (Prüferin) _____

Rechtsmittelbelehrung: Gegen die Bewertung dieser Prüfungsleistung kann innerhalb eines Monats nach ihrer Bekanntgabe Widerspruch erhoben werden. In diesem Zeitraum kann die Bewertung der Klausur eingesehen werden. Der Widerspruch ist schriftlich oder zur Niederschrift bei der bzw. dem Vorsitzenden des für das Hauptfach zuständigen Prüfungsausschusses einzulegen. Es wird darauf hingewiesen, dass ein erfolgloses Widerspruchsverfahren kostenpflichtig ist.

Zugelassene Hilfsmittel: Die ausgeteilten Prüfungsunterlagen, ein Handbuch zu Scheme (Revised Report on Scheme), ein Wörterbuch, Schreibstifte.

Nicht verwendet werden dürfen: Mobiltelefone oder sonstige elektronische Kommunikationsmittel, sonstige schriftliche Aufzeichnungen zur Vorlesung und zu den Übungen.

**Datum & Unterschrift
Klausurteilnehmer/in:** _____

Wichtige Hinweise:

- Die Klausur bitte auf jeden Fall geheftet lassen! Kein zusätzliches Papier verwenden!
- **Unterschreiben Sie die Klausur auf dieser Seite.**
- Punkte für Teilaufgaben werden durch $\{\circ\}$ angegeben, z.B. $\{\circ\circ\} = 2$ Punkte. Die Gesamtanzahl von Punkten pro Aufgabe steht jeweils im Kasten am Rand der entsprechenden Aufgabe unter "von". Beachten Sie, dass Sie für jeden zu erzielenden Punkt etwa zwei Minuten Zeit zur Verfügung haben.
- Sollte der Platz für eine Aufgabe nicht ausreichen, so verwenden Sie die Leerseiten am Ende und machen Sie einen Vermerk am Rand, etwa siehe " \implies Seite ??".
- Für alle Aufgaben können Sie die Funktionen aus dem tools-module der se3-bib als vordefiniert voraussetzen.

```
#lang scheme
(require "se3-bib/tools-module.ss")
```

- Im Anhang der Klausur, auf Seite ??, finden Sie ein kleines Handbuch zu Funktionen höherer Ordnung, CLOS und Prolog-in-Scheme.

1 Aufgaben

1. Zu welchen Werten evaluieren die folgenden Scheme-Ausdrücke?

(a) `(+ 3 (- 6 7))`

{ \emptyset }

von
10

(b) `'(+ 3 (- 6 7))`

{ \emptyset }

(c) `(car '(auto bus))`

{ \emptyset }

(d) `(cdr '(auto bus))`

{ \emptyset }

(e) `(map sqrt '(1 9 4 25))`

{ $\emptyset \emptyset$ }

(f) `((curry < 6) 2)`

{ $\emptyset \emptyset$ }

(g) `(reduce * '(1 2 6 0 1) 1)`

{ $\emptyset \emptyset$ }

VON
10

2. Reduktionsstrategien

- (a) Geben Sie einen Scheme-Ausdruck an, für den die Auswertung durch innere Reduktion vorteilhafter ist als die äußere Reduktion. Begründung? $\{\circ\circ\}$

- (b) Ausdrücke mit `if` und `cond` werden in Scheme anders ausgewertet als funktionale Ausdrücke. $\{\circ\circ\circ\}$
Worin besteht der Unterschied, und warum ist das notwendig?
Nennen Sie zwei weitere *special form operators*.

- (c) Zur Syntax von Scheme: $\{\circ\circ\circ\circ\circ\}$
Gegeben seien die folgenden Definitionen:

```
(define *a* 10)
(define *b* '*a*)
(define (merke x)(lambda () x))
(define (test x)
  (let ((x (+ x *a*)))
    (+ x 2)))
```

Haben die folgenden Ausdrücke einen wohldefinierten Wert und zu welchen Werten evaluieren sie?

1. `*a*`
2. `(+ *a* *b*)`
3. `(+ (eval *a*) (eval *b*))`
4. `(and (> *a* 10) (> *b* 3))`
5. `(or (> *a* 10) (/ *a* 0))`
6. `(+ 2 (merke 3))`
7. `(+ 2 ((merke 3)))`
8. `(test 4)`

VON
10

3. **Formen der Rekursion:** Gegeben seien die folgenden Funktionsdefinitionen:

```
(define (alle-neune xs)
  (cond ((null? xs) 0)
        ((= 9 (car xs)) (+ 1 (alle-neune (cdr xs))))
        (else (alle-neune (cdr xs)))))
```

```
(define (alle-zehne xs wieviele)
  (cond ((null? xs) wieviele)
        ((= 10 (car xs))
         (alle-zehne (cdr xs) (+ 1 wieviele)))
        (else (alle-zehne (cdr xs) wieviele))))
```

```
(define (anzahl-atome xs)
  (cond ((null? xs) 0)
        ((atom? xs) 1)
        (else (+ (anzahl-atome (car xs))
                  (anzahl-atome (cdr xs))))))
```

Betrachten Sie die angegebenen Funktionen `alle-neune`, `alle-zehne` und `anzahl-atome`.
Geben Sie für jede dieser Funktionen an, welche Art von Rekursion vorliegt. {○○○○○}

Begründen Sie Ihre Antwort.

{○○○○○}

4. Rekursion:

(a) Nennen Sie eine rekursive Scheme-Datenstruktur. Begründung? $\{\emptyset\emptyset\}$

VON
10

(b) Definieren Sie eine *rekursive* Funktion (`laengen xss`) in Scheme, die folgendes leistet:

Gegeben sei eine Liste von Listen. Bilden Sie diese ab auf die Liste der Längen der Unterlisten.

Beispiel:

`(laengen (1 3 4) (Auto Bus) () (3 4 5 6))` \longrightarrow `(3 2 0 4)`.

Geben Sie die Funktion in drei Varianten an:

i. linear rekursiv, $\{\emptyset\emptyset\}$

ii. endrekursiv, $\{\emptyset\emptyset\}$

iii. und mittel einer Funktion höherer Ordnung $\{\emptyset\emptyset\}$

(c) Woran erkennen Sie eine endrekursive Funktion? $\{\emptyset\emptyset\}$

VON
14

5. Funktionen höherer Ordnung: Regressionsgerade

Gegeben sei eine Liste von Wertpaaren $((x_1.y_1), (x_2.y_2), \dots, (x_n.y_n))$ als gepunktete Paare. Definieren Sie Scheme-Funktionen zur Berechnung der folgenden Werte. Verwenden Sie dabei nach Möglichkeit Funktionen höherer Ordnung, wie `map`, `reduce`, `curry` usw.

- (a) Eine Funktion `xliste`, die aus einer Liste von Wertpaaren die Liste aller x-Komponenten extrahiert, z.B. $\{\emptyset \emptyset\}$

`(xliste '((1 . 3) (2 . 4) (3 . 5)))` \longrightarrow `(1 2 3)`

- (b) Eine Funktion `yliste`, die aus einer Liste von Wertpaaren die Liste aller y-Komponenten extrahiert, z.B. $\{\emptyset \emptyset\}$

`(yliste '((1 . 3) (2 . 4) (3 . 5)))` \longrightarrow `(3 4 5)`

- (c) Eine Funktion `xsumme`, die die x-Werte einer Liste von Wertpaaren addiert $\sum_{i=1}^n x_i$: $\{\emptyset \emptyset\}$

`(xsumme '((1 . 3) (2 . 4) (3 . 5)))` \longrightarrow `6`

- (d) Eine Funktion **x*y-Summe**, die die xy-Produkte einer Liste von Wertpaaren addiert $\sum_{i=1}^n x_i \cdot y_i$: $\{\emptyset \emptyset\}$

$$(\text{x*y-Summe } '((1 \ . \ 3) (2 \ . \ 4) (3 \ . \ 5))) \longrightarrow 26$$

- (e) Eine Funktion **x**2-Summe**, die die Quadrate der x-Werte einer Liste von Wertpaaren addiert $\sum_{i=1}^n x_i^2$: $\{\emptyset \emptyset\}$

$$(\text{x**2-Summe } '((1 \ . \ 3) (2 \ . \ 4) (3 \ . \ 5))) \longrightarrow 14$$

- (f) Die Gerade $y = m \cdot x + c$, deren quadratischer Anpassungsfehler für eine gegebene Menge von Wertpaaren $\{(x_1, y_1), \dots, (x_n, y_n)\}$ am kleinsten ist, hat die Steigung m und den Achsenabschnitt c mit den folgenden Werten:

$$m = \frac{n \cdot \sum_{i=1}^n (x_i \cdot y_i) - \sum_{i=1}^n x_i \cdot \sum_{i=1}^n y_i}{n \cdot \sum_{i=1}^n x_i^2 - \sum_{i=1}^n x_i \cdot \sum_{i=1}^n x_i}$$

$$c = \frac{\sum_{i=1}^n y_i - m \cdot \sum_{i=1}^n x_i}{n}$$

Zusatzaufgabe: Definieren Sie eine Scheme-Funktion `anpassungsgerade` als Funktion höherer Ordnung, die für eine Menge von Wertpaaren die Anpassungsgerade berechnet und eine *Scheme-Funktion* als Wert zurückgibt, die die y -Werte der Geraden als Funktion von x berechnet. Beispiel: $\{\circ \circ \circ \circ\}$

```
> (define g-anp
      (anpassungsgerade '((1 . 0.5) (2 . 1) (3 . 1.5))))
> (g-anp 2) -> 1
```

denn die optimale Anpassungsgerade ist die Gerade $y = 0.5 \cdot x$, und der y -Wert für $x = 2$ ist 1.

6. Vererbung in CLOS:

- (a) Definieren Sie als **CLOS-Klasse** eine Klasse von Fahrzeugen und spezialisieren Sie diese Klassen für unterschiedliche Medien, in denen sich die Fahrzeuge bewegen: Bodengebundene Landfahrzeuge, schwimmfähige Wasserfahrzeuge, flugfähige Luftfahrzeuge.

{○○○}

VON
11

(b) Definieren Sie Klassen von Mehrzweckfahrzeugen: $\{\emptyset\emptyset\}$

- i. Ein Amphibienfahrzeug, das sich zu Wasser und zu Lande bewegen kann.
- ii. Ein Batman-Superfahrzeug, das sich zu Wasser, zu Lande und in der Luft bewegen kann.

(c) Die Klasse Fahrzeug soll folgende Operationen bieten: $\{\emptyset\emptyset\emptyset\emptyset\emptyset\emptyset\}$

- i. Abfrage des Mediums, in dem sich das Fahrzeug bewegt,
- ii. Abfrage der Maximalgeschwindigkeit,
- iii. Abfrage der Zuladung (Tragfähigkeit)
- iv. Abfrage des Verbrauchs pro 100 km
- v. Abfrage der der Passagierzahl

Spezifizieren Sie generische Funktionen als Signatur für die Operationen der Klasse Fahrzeug und diskutieren Sie, welche Methodenkombination für die jeweilige Operation sinnvoll erscheint.

VON
10

7. Prolog in Scheme:

Gegeben sei eine Prolog-in-Scheme-Datenbasis, die Angaben zu Medikamenten, deren Wirkstoffen und Indikationen enthält.

- Die Relation „wirkstoff“ beschreibt, welchen Wirkstoff ein Medikament enthält, z.B. Acetylsalicylsäure.
- Die Relation „Indikation“ beschreibt, für welche Krankheiten das Medikament eingesetzt werden kann.

Ein Auszug aus der Datenbasis:

```
;(wirkstoff Medikament Wirkstoff)
(<- (wirkstoff "Aspirin_Plus_C" Acetylsalicylsäure))
(<- (wirkstoff "Aspirin_Plus_C" Vitamin_C))
(<- (wirkstoff "Aspro" Acetylsalicylsäure))
(<- (wirkstoff "Spalt" Acetylsalicylsäure))
(<- (wirkstoff "Paracetamol-Ratiopharm" Paracetamol))
(<- (wirkstoff "Talvosilen" Paracetamol))
(<- (wirkstoff "Talvosilen" Codein))
(<- (wirkstoff "Proviron" Mesterolol))
(<- (wirkstoff "Andriol" Testetoronundecanoat))

;(indikation Wirkstoff Anwendung)
(<- (indikation Acetylsalicylsäure
          "Fiebersenkend"))
(<- (indikation Acetylsalicylsäure
          "Schmerzmittel"))
(<- (indikation Codein "Hustendaempfer"))
(<- (indikation Mesterolol "Hormonmangel"))
```

Formulieren Sie die folgenden Fragen in Prolog-in-Scheme, und stellen Sie entsprechende Anfragen an die Datenbasis:

- (a) Welche Wirkstoffe enthält das Medikament „Aspirin Plus C“? $\{\emptyset\}$
- (b) Welche Wirkstoffe haben dieselben Anwendungsgebiete wie der Wirkstoff „Acetylsalicylsäure“? $\{\emptyset\emptyset\}$

(c) Welche Medikamente sind Hustendämpfer? {⊗ ⊗}

(d) Welche Medikamente sind als Kombinationspräparate bedenklich, da sie mehr als einen Wirkstoff enthalten? {⊗ ⊗}

(e) In den Sprachen Prolog und PrologInScheme sind funktionale Sprachelemente eingebettet. Nennen Sie ein Beispiel: {⊗}

(f) Was ist zu beachten, wenn in relationalen Programmiersprachen funktionale Sprachelemente verwendet werden? Welche wichtige Eigenschaft geht verloren?
{⊗ ⊗}

2 Funktionslexikon

2.1 Funktionen höherer Ordnung

2.1.1 Verknüpfung von Funktionen

(**curry** *f* *arg1* ... *argn*) : Partielle Anwendung von *f* auf *arg1* ... *argn* von links nach rechts

(**rcurry** *f* *argm* ... *argn*) : Partielle Anwendung von *f*, Bindung der Argumente von rechts nach links

(**compose** *f1* ... *fn*) : Funktionskomposition, Hintereinanderausführung
(*f1* (... (*fn* *x*)))

(**conjoin** *p1?* ... *pn?*) : Konjunktion von Prädikaten

(**disjoin** *p1?* ... *pn?*) : Disjunktion von Prädikaten

(**always** *x*) : Die konstante Funktion, deren Wert unabhängig von den Argumenten *x* ist.

2.1.2 Idiome der funktionalen Programmierung

(**apply** *f* *xs*) : Anwendung einer Funktion *f* auf eine Liste von Argumenten *xs*

(**map** *f* *xs1* ... *xs_n*) : Abbilden einer oder mehrerer Listen auf die Liste der Bilder der Elemente. Die Stelligkeit der Funktion *f* muß mit der Anzahl der angegebenen Listen *xs1*...*xs_n* usw. übereinstimmen.

(**filter** *p?* *xs*) : Die Liste der Elemente von *xs*, die *p?* erfüllen

(**reduce** *f* *xs* *seed*) : Paarweise Verknüpfung der Elemente von *xs* mit *f*, Startwert *seed*,
(*f* *x1* (*f* *x2* (... (*f* *x_n* *seed*))))

(**iterate** *f* *end?* *start*) : Die Liste der Funktionsanwendungen, bis *end?* erfüllt ist,
(*start* (*f* *start*) (*f* (*f* *start*)) ...)

(**untilM** *f* *end?* *start*) : Der erste Wert der Folge *start*, (*f* *start*), (*f* (*f* *start*)), ...
der *end?* erfüllt

(**some** *p?* *xs*) : Finde das erste Element von *xs*, das das Prädikat *p?* erfüllt, ansonsten
gebe #f zurück.

(**every** *p?* *xs*) : wahr, wenn alle Elemente von *xs* das Prädikat *p?* erfüllen.

(**assoc** *key* *alist*) : Suche in der Assoziationsliste *alist* das erste Paar, daß das als *Kopf*
den Schlüssel *key* enthält.

(**rassoc** *key* *alist*) : Suche in der Assoziationsliste *alist* das erste Paar, daß das als *Rest*
den Schlüssel *key* enthält.

2.2 CLOS

2.2.1 Klassen

```
(defclass <Name der Klasse> ({<Oberklassen>})  
  {( <Slot> {<Slot-keys>})}  
  {<Class-keys>}  
  )
```

Optionen für die Attribute (slots):

- `:initarg keyword: <key>` *Schlüsselwort für den Konstruktor*
- `:initializer <func>` *Initialisierungsfunktion*
- `:initvalue <value>` *Defaultwert*
- `:reader <name>` *Akzessorfunktion zum Lesen des Wertes*
- `:writer <name>` *Akzessorfunktion zum Schreiben des Wertes*
- `:accessor <name>` *Akzessorfunktion zum Lesen und Schreiben des Wertes*
- `:accessor <name>` *Akzessorfunktion zum Lesen und Schreiben des Wertes*
- `:type <type>` *Typdefinition für die Klasse des slots*

```
(defgeneric <name>({(<arg> <class>)} {<arg>})  
  {:combination <combination>}  
  {:documentation <string>} )
```

```
(defmethod <name> [<qualifier>]({(<arg> <class>)} {<arg>})  
  {:documentation <string>} )
```

2.2.2 Ergänzungsmethoden

`<qualifier>` ::= `:after` | `:before` | `:around`

2.2.3 Methodenkombinationen

```
generic-+-combination, generic-list-combination  
generic-min-combination, generic-max-combination  
generic-append-combination, generic-append!-combination  
generic-begin-combination, generic-and-combination  
generic-or-combination
```

2.3 Prolog-in-Scheme-Lexikon

Variable: ?X : Die Namen von Variablen beginnen mit einem Fragezeichen.

Anonyme Variable: ? : Ein Fragezeichen bezeichnet eine anonyme Variable, die in Ausgaben unterdrückt wird.

Regeln: Klauseln mit Prämissen (Zielen) :

`<Klausel-Kopf> :- <Ziel 1>...<Ziel n>`

Negation: not : Der not-Operator negiert eine Klausel : `(not <Klausel>)`

Ungleichheit: != : Das !=-Prädikat ist wahr, wenn zwei Strukturen oder Variablen nicht unifizieren :

`(!= <struktur1> <struktur2>)`

Das assert-Macro: ← : Trage eine Klausel in die Datenbasis ein.

`(<- <Klausel>)`

Das query-Macro: ?- : Anfrage, durch welche Variablenbindungen die Konjunktion der Ziele in der Anfrage erfüllt werden kann.

`(?- <Ziel 1> ... <Ziel n>)`

findall: findall sammelt alle Resultate des Prädikatsaufrufs `<Ziel>` in einer Liste `<Liste>` als instanziierte Varianten des Ausdrucks `<Term>` .

`(findall <Term> <Ziel> <Liste>)`

count: count zählt die Resultate des Prädikatsaufrufs `<Ziel>` und bindet die Anzahl an die Variable `<Var>`.

`(count <Var> <Ziel>)`

Funktionale Auswertung: is: Das is-Prädikat bindet den Wert eines funktionalen Ausdrucks an eine Variable. Alle Variablen des Ausdrucks müssen vorher an Werte gebunden sein.

`(is <Var> <Ausdruck>)`

Funktionale Auswertung: test: Das test-Prädikat evaluiert einen funktionalen Ausdruck. Das Prädikat ist erfüllt, wenn der Ausdruck wahr ist. Alle Variablen des Ausdrucks müssen vorher an Werte gebunden sein.

`(test <Ausdruck>)`

Zusätzliche Leerseiten

Aufgabe	Punkte
1	
2	
3	
4	
5	
6	
7	
Summe:	
von	75
Bestanden?	Ja <input type="radio"/> Nein <input type="radio"/>

Datum & Unterschrift Vorkorrektur _____

Datum & Unterschrift Prüferin _____