Hochschule für angewandte Wissenschaften
Würzburg-Schweinfurt
Algorithmen und Datenstrukturen 1
Wintersemester 2021/2022
Prof. Dr. Frank Deinzer, M.Sc. Markus Ebner
09.02.2022
08:30 - 10:00

Matrikelnummer

Name, Vorname:

Diese Prüfung besteht aus 16 Seiten (inkl. dieses Deckblattes) und 8 Fragen. Sie können maximal 69 Punkte erreichen.

Bitte beantworten Sie alle Fragen soweit möglich auf diesem Prüfungsbogen. Entfernen Sie **nicht** die Klammern, welche diesen Prüfungsbogen zusammen halten. Geben Sie alle zusätzlich verwendeten Blätter (versehen mit ihrer Matrikelnummer) am Ende ab. Bitte markieren Sie darauf eindeutig, auf welche Frage sich Ihre Antworten beziehen.

Frage	1	2	3	4	5	6	7	8	Gesamt
Punkte	13	9	10	12	2	7	6	10	69
Erreicht									

7

1. Paare und Listen

13 Punkte

[4 Punkte] (a) Geben Sie für die folgenden Scheme-Ausdrücke die entsprechenden Ausgaben des Scheme-Interpreters an.

Beispiel:
Eingabe: `(list (list 1 2) (list 3 4 5))`
Ausgabe: `((1 2) (3 4 5))`

1

Eingabe 1: `(list 1 (cons 2 3) (list 4 5))`

Ausgabe 1: ~~((2.3~~ (1 (2.3)(4 5)) ✓

((1 2).(3 4)).(5 6)

Eingabe 2: `(cons (cons (list 1 2) (list 3 4)) (cons 5 6))`

Ausgabe 2: (((1 2).(3 4)).(5.6)) ✗

(1 (2.(3 (4))))

Eingabe 3: `(list 1 (cons 2 (list 3 (list 4 '()))))`

Ausgabe 3: (1(2 3 (4))) ✗

(((1.'()) 2).3) '() 4)

Eingabe 4: `(list (cons (list (cons 1 '()) 2) 3) '() 4)`

Ausgabe 4: (((1 '()) 2).3) '() 4) ✗

[4 Punkte] (b) Geben Sie jeweils einen Scheme-Ausdruck an, der die angegebene Datenstruktur erzeugt.

Beispiel:
Datenstruktur: `((1 2) (3 4 5))`
Scheme-Ausdruck: `(list (list 1 2) (list 3 4 5))` 1

Datenstruktur 1: `(1 2 (3 . 4))` (list 1 2 (cons 3 4))

Scheme-Ausdruck 1: (list 1 2 (cons 3,4)) ✓
↖ durchgestrichener Punkt

Datenstruktur 2: `(((1 2 . 3) (4)))`

Scheme-Ausdruck 2: (list (list (list 1 2 '. 3) (list 4))) f

Datenstruktur 3: `((1 2 . 3) 4)`

Scheme-Ausdruck 3: (list (list 1 2 '. 3) 4) f

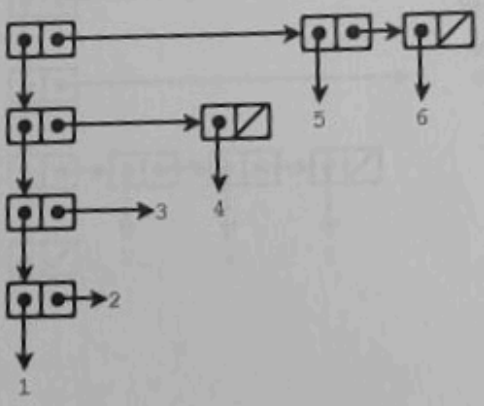
Datenstruktur 4: `(define (f x) (+ x 1))`

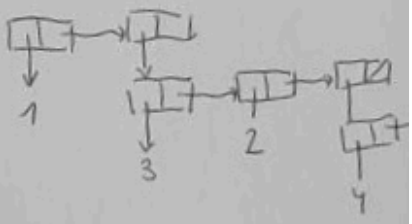
Scheme-Ausdruck 4: (list 'define (list 'f 'x) (list '+ 'x 1)) f

[5 Punkte] (c) Die Namen `a1, a2, ..., a8` sind als folgende Scheme-Ausdrücke definiert. Tragen Sie in die leeren ovalen Kästchen jeweils den Namen des Ausdrucks ein, der dargestellt wird.

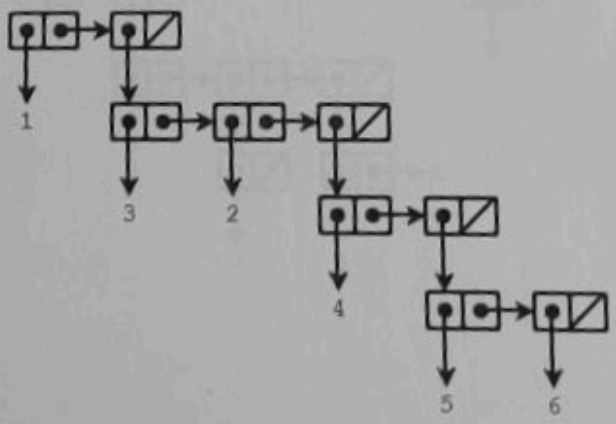
```
(define a1 (list 1 (cons 3 (list 2 (list 4 '(5 6))))))
(define a2 (cons (list (list 1 2) (cons 3 4)) (list 5 6)))
(define a3 (list (list (cons (cons 1 2) 3) 4) 5 6))
(define a4 (list 1 (list 2 (cons 3 '()) (cons 4 5)) 6))
(define a5 (list 1 (cons 2 3) (list 4 5) 6))
(define a6 (cons (cons (list 1 2) (cons 3 4)) (list 5 6)))
(define a7 (list 1 (cons 2 (list 3 (cons 4 '()) 5 '() 6))))
(define a8 (list (cons (list (cons 1 '()) 2 3 4) 5) 6))
```

a3

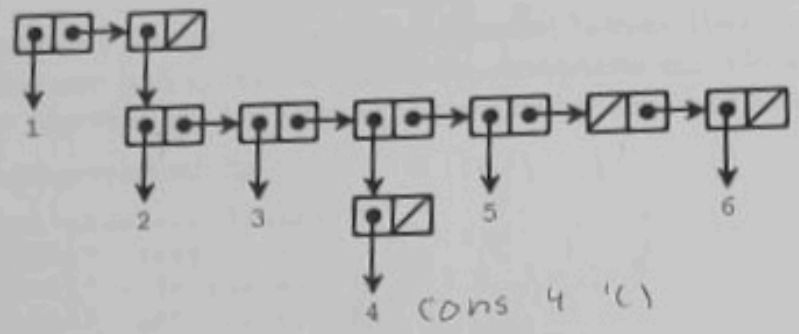




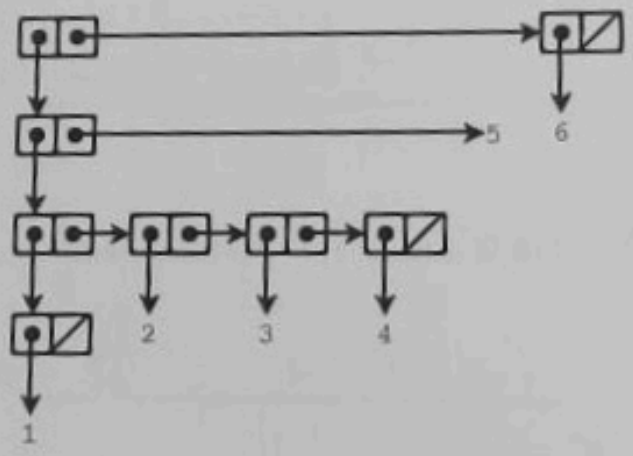
a1



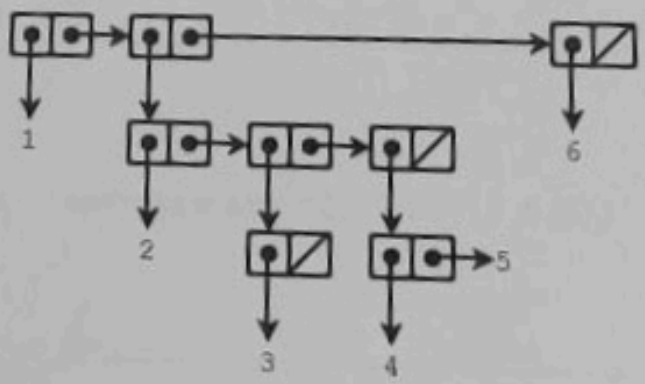
✓ a7



✓ a8



✓ a4



3

9 Punkte

2. Ablaufanalyse

Stellen Sie sich vor, Sie starten für jede der folgenden beiden Teilaufgaben den Scheme-Interpreter neu und geben die folgenden Ausdrücke ein. Geben Sie für jede Eingabe das Ergebnis der Auswertung des angegebenen Ausdrucks an.

[5 Punkte] (a) Rekursive Listenverarbeitung

```
1 (define (was-passiert liste)
2   (cond ((null? liste) -1)
3         ((null? (cdr liste)) 1)
4         ((null? (car liste)) 2)
5         ((list? (car liste)) 0)
6         (else (+ (car liste) (was-passiert (cdr liste))))))
```

3

Eingabe 1: `(was-passiert (list))`

Ausgabe 1: −1 ✓

(E (+)

Eingabe 2: `(was-passiert '(1 2 3 4 5 6))` 1+2+3+4+5+6 1 (6 10 15) null cdr

Ausgabe 2: 16 ✓

Eingabe 3: `(was-passiert '(1 2 3 () 5 6))` 1+2+3−1

Ausgabe 3: 5 ∫

Eingabe 4: `(was-passiert '(1 2 3 '() 5 6))` 1+2+3+0

Ausgabe 4: 6 ✓

Eingabe 5: `(was-passiert '(1 (2 (3 (() 5 6)))))` 1+2+3+2

Ausgabe 5: 8 ∫

[4 Punkte] (b) Variablenverschattung

```
1  (define p 3)
2  (define q -4)
3  (define (funktion-A u v)
4    (let* ((p (* 2 u v))
5           (v u)
6           (q (+ u v p)))
7      (+ p q v)))
8
9  (define (funktion-B u v)
10   (let ((p (* 2 u v))
11         (v u)
12         (q (+ u v p)))
13     (+ p q v)))
```

Eingabe 1: `(funktion-A 4 3)`

Ausgabe 1: _____

Eingabe 2: `(funktion-B 4 3)`

Ausgabe 2: _____

Eingabe 3: `(funktion-A 1 -1)`

Ausgabe 3: _____

Eingabe 4: `(funktion-B 1 -1)`

Ausgabe 4: _____

3. Prozeduranalyse **10 Punkte** /5

Analysieren Sie die folgenden zwei Prozeduren, und füllen Sie die zugehörigen Informationsfelder aus.

[5 Punkte] 2

(a) **Prozedur 1: Element in Vektor einfügen**

```
1  (define (vektor-einfuegen vektor index element)
2    (define (einfuegen n)
3      (cond ((= n index) (vector-set! vektor index element) vektor)
4            (else (vector-set! vektor n (vector-ref vektor (- n 1)))
5                  (einfuegen (- n 1)))
6      )
7    )
8    (let ((len (vector-length vektor)))
9      (cond ((or (< index 0) (>= index len)) #f)
10           (else (einfuegen (- (vector-length vektor) 1)))
11     )
12   )
13 )
```

verzögernder Operator

iterativ

Handelt es sich um einen iterativen oder rekursiven Prozess? rekursiv

Zeitkomplexität: ~~T(n)~~ $T(n) = O(n)$ Speicherkomplexität: $P(n) = O(n)$?

Art der Rekursion (z.B.: Linear, Baum, ...) linear

Für den Fall, dass es sich um einen rekursiven Prozess handelt, markieren Sie alle verzögernden Operationen in der jeweiligen Prozedur.

X

[5 Punkte] 3

(b) **Prozedur 2: Listenelement Wiederholer**

```
1  (define (element-wiederholer liste)
2      (define (list-length liste n)
3      (cond ((null? liste) n)
4            (else (list-length (cdr liste) (+ n 1)))))
5
6    (define (element-wiederholen el n)
7      (cond ((= n 0) '())
8            (else (cons el (element-wiederholen el (- n 1))))))
9
10   (define (internal liste n)
11     (cond ((empty? liste) '())
12           (else (cons
13                  (element-wiederholen (car liste) n)
14                  (internal (cdr liste) n)))))
15
16   (internal liste (list-length liste 0))
17 )
```

Handelt es sich um einen iterativen oder rekursiven Prozess? rekursiv ✓

Zeitkomplexität: $T(n) = O(n^2)$ Speicherkomplexität: $P(n) = O(n)$ ✓

Art der Rekursion (z.B.: Linear, Baum, ...) linear ✓

Für den Fall, dass es sich um einen rekursiven Prozess handelt, markieren Sie alle verzögernden Operationen in der jeweiligen Prozedur.

X

12 Punkte

4. Komplexitäts-Einschätzung
Geben Sie für die folgenden algorithmischen Lösungen jeweils die Speicher- und Zeitkomplexität an.

[2 Punkte] (a) Berechnung der Fakultät in einem rekursiven Prozess.

Zeitkomplexität: $T(n) = O(n)$ ✓ Speicherkomplexität: $P(n) = O(n)$ ✓

[2 Punkte] (b) Berechnung der Fibonacci-Zahlen in einem iterativen Prozess.

Zeitkomplexität: $T = k^n$ Speicherkomplexität: $P(n) = O(1)$ ✓

[2 Punkte] (c) Berechnung der Fibonacci-Zahlen in einem rekursiven Prozess.

Zeitkomplexität: $T(n) = O(n)$ ✓ Speicherkomplexität: $P(n) = O(n)$ ✓

[2 Punkte] (d) Iterative Suche nach einem Element in einer Menge, wenn die zugrunde liegende Datenstruktur eine sortierte Liste ist.

Zeitkomplexität: $T(n) = O(\log n)$ ✓ Speicherkomplexität: $P(n) = O(1)$ ✓

[2 Punkte] (e) Berechnung der Fakultät in einem iterativen Prozess.

Zeitkomplexität: $T(n) = O(n)$ ✓ Speicherkomplexität: $P(n) = O(1)$ ✓

[2 Punkte] (f) Iterative Suche nach einem Element in einer Menge, wenn die zugrunde liegende Datenstruktur ein perfekt ausbalancierter Baum ist.

Zeitkomplexität: $T(n) = O(\log n)$ ✓ Speicherkomplexität: $P(n) = O(1)$ ✓

77

5. Prozesseigenschaften **2 Punkte**

[1 Punkt] (a) Was sind die Vor- und Nachteile eines rekursiven bzw. eines iterativen Prozesses?

7

- rekursiv benötigt mehr Speicher, ist aber übersichtlicher

- iterativ wird für komplexere/aufwendigere Vorgänge genutzt und benötigt etwas weniger Speicher ✓

[1 Punkt] (b) Woran erkennt man, welche Art von Prozess eine Prozedur erzeugt?

0

- am verzögernden Operater (zB Methode ruft sich selbst auf)
- ~~an der verbrau~~ an der benötigten Speicherkapazität im direkten Vergleich

7

6. Substitutionsmodell

7 Punkte 17

Wenden Sie das Substitutionsmodell auf die Prozeduren `fn1` und `fn2` an, d.h. beschreiben Sie die Prozeduraufrufe die der Scheme-Interpreter bei der Auswertung macht. Ignorieren Sie hierbei die Aufrufe von Bedingungen.

Hier, **beispielhaft ein Lösungsvorschlag** für die `fakultaet`-Prozedur aus dem Skript. Nutzen Sie diese als Vorlage für ihre Lösung.

```
1 (define (fakultaet n)
2   (if (= n 1)
3       1
4       (* n (fakultaet (- n 1))))))  n)))
```

Substitutionsmodell für den Aufruf `(fakultaet 6)`:

```
(fakultaet 6)
(* 6 (fakultaet 5))
(* 6 (* 5 (fakultaet 4)))
(* 6 (* 5 (* 4 (fakultaet 3))))
(* 6 (* 5 (* 4 (* 3 (fakultaet 2)))))
(* 6 (* 5 (* 4 (* 3 (* 2 (fakultaet 1))))))
(* 6 (* 5 (* 4 (* 3 (* 2 1)))))
(* 6 (* 5 (* 4 (* 3 2))))
(* 6 (* 5 (* 4 6)))
(* 6 (* 5 24))
(* 6 120)
720
```

[3 Punkte] 3

(a) **Prozedur 1:**

```
1 (define (fn1 n)
2   (define (inner a b zaehler)
3   (if (= zaehler 0)
4       b
5       (inner (+ a b) a (- zaehler 1))))
6   (inner 1 0 n))
```

(fn1 7)

((fn1 7))

(inner 1 0 7) ✓

(inner (+ 1 0) 1 (- 7 1))

(inner 1 1 6)

(inner (+ 1 1) 1 (-6 1))

(inner 2 1 5)

(inner (+ 2 1) 2 (-5 1))

(inner 3 2 4)

(inner 5 3 3) ✓

(inner 8 5 2)

(inner 13 8 1)

(inner 21 13 0)

13 ✓

[4 Punkte] (b) **Prozedur 2:**

```
1 (define (fn2 b n)
2   (if (= n 0)
3       1
4       (* b (fn2 b (- n 1)))))
```

4

(fn2 2 4)

(fn2 2 4)

(* 2 (fn2 2 (- 4 1))) → (fn2 2 3) (b n)

(* 2 (fn2 2 2))

(* 2 (fn2 2 1))

(* 2 (fn2 2 0)) → 1 wird an alle zu

(* 2 1)

(* 2 2) → 4

(* 2 4)

(* 2 8)

~~16~~

← auf der Rückseite von (13 von 16)

~~applikativ dargestellt, Ergebnis von fn2 wird jeweils zurückgegeben und multipliziert~~

6 Punkte

7. Fehlersuche

Die Methode `vektor-filter` implementiert das Remove-Erase-Idiom. Der übergebene `vektor` wird mit dem ebenfalls übergebenen `praedikat` gefiltert, indem die Elemente, für die das `praedikat` true zurück liefert, an den Anfang des Vektors verschoben, und der `vektor` anschließend auf die benötigte Länge gekürzt wird. Eine beispielhafte Ausgabe der Prozedur sieht also so aus:

```
(vektor-filter odd? (vector 1 2 3 4 5 6))   →  #(1 3 5)
(vektor-filter even? (vector 1 2 3 4 5 6))  →  #(2 4 6)
```

In der untenstehenden Implemlentierung sind 3 Fehler enthalten.
Finden (1 Punkt) und korrigieren (1 Punkt) Sie die eingebauten Fehler.
Hinweis: Falsche Korrekturen führen zu Punktabzug; Sie können in dieser Aufgabe aber nicht weniger als 0 Punkte erreichen.

```
1  (define (vektor-filter praedikat vektor)
2    (let ((laenge (vector-length vektor)))
3      (define (filter lese-pos schreib-pos)
4        (cond ((<= lese-pos laenge) (- lese-pos schreib-pos))
5              ((praedikat (vector-ref vektor lese-pos))
6               (vector-set! vektor schreib-pos
7                            (vector-ref vektor lese-pos))
8               (filter (+ 1 schreib-pos) (+ 1 lese-pos)))
9              (else (filter (+ 1 lese-pos) schreib-pos))))
10     (vector-drop-right vektor (filter 0 -1))))
```

3

8. Theoretische Grundlagen

10 Punkte

Bestimmen Sie, ob die folgenden Aussagen wahr oder falsch sind. Bitte raten Sie nicht, da falsche Antworten zu Punktabzug führen. Für jede richtige Antwort wird ein Punkt vergeben; für eine falsche Antwort wird ein Punkt abgezogen, wobei die Nebenbedingung gilt, dass die Gesamtpunktzahl der Aufgabe nicht negativ ist. Unbeantwortete Fragen haben keinen Einfluss auf die Punktevergabe.

Sollte eine Frage aus Ihrer Sicht nicht eindeutig gestellt sein oder Sie zusätzliche Anmerkungen haben, geben Sie Kommentare zu Ihren Antworten unter den Fragen an.

		wahr	falsch	
1.	Quotierung verhindert die Auswertung von Ausdrücken	☒	☐	✓
2.	Jede baumrekursive Funktion lässt sich auch iterativ implementieren	☒	☐	✓
3.	Prozeduren höherer Ordnung können Prozeduren höherer Ordnung als Ergebnis liefern	☐	☐	–
4.	`expt` ist eine Spezialform, weicht also von der normalen Auswertungsreihenfolge ab	☐	☐	–
5.	`or` ist eine Spezialform, weicht also von der normalen Auswertungsreihenfolge ab	☒	☐	✓
6.	Lokale Namen können andere lokale Namen verdecken	☒	☐	✓
7.	`not` ist eine Spezialform, weicht also von der normalen Auswertungsreihenfolge ab	☐	☒	✓
8.	Lokale Namen sind ein Abstraktionsmechanismus	☒	☐	✓
9.	Mengen können effizient durch Bäume dargestellt werden	☒	☐	✓
10.	Closures haben Zugriff auf ihre Erstellungsumgebung	☒	☐	–

not