Philipps-Universität Marburg
Fachbereich Mathematik und Informatik
Arbeitsgruppe Programmiersprachen und -werkzeuge

Sommersemester 2019
17.07.2019

Prof. Dr. Christoph Bockisch

Klausur zur Vorlesung

Deklarative Programmierung

Wichtige Hinweise:

- Schalten Sie, soweit noch nicht geschehen, sofort Ihr Mobiltelefon aus!
- Schalten Sie auch alle anderen medizinisch nicht notwendigen potentiellen Lärmquellen aus.
- Entfernen Sie jetzt alle Gegenstände vom Tisch, außer einen Stift (kein Rot-, Grün-, oder Bleistift), Ihren Studentenausweis und Ihren Personalausweis/Reisepass, Getränke.
- Schreiben Sie jetzt auf jedes Blatt in Druckbuchstaben Ihren Namen und Ihre Matrikelnummer. Blätter ohne Namen ergeben 0 Punkte und werden nicht korrigiert! Füllen Sie insbesondere auch folgende Tabelle in Druckbuchstaben aus:

Vorname	
Nachname	
Matrikelnummer	
Fachbereich	
Studienfach	
Angestrebter Abschluss	

- Im Zweifel sollten Sie Diagramme in der Klausur möglichst detailliert und auf einem möglichst konkreten Niveau erstellen.
- Die Bearbeitungszeit beträgt **2** Stunden.
- Verwenden Sie kein eigenes Papier, für Notizen falls der Platz für die Lösung nicht ausreicht, sondern die Rückseiten. In dringenden Fällen erhalten Sie auf Anfrage Zusatzblätter. Machen Sie gut kenntlich, wenn Sie Rückseiten oder Zusatzblätter für Lösungen verwenden und tragen Sie dort Name und Matrikelnummer ein.
- Es sind keine Hilfsmittel erlaubt. Zuwiderhandlungen führen zum Ausschluss.
- Mehrere widersprüchliche Lösungen zu einer Aufgabe werden mit 0 Punkten bewertet.
- Falls Sie eine Frage haben, so wenden Sie sich bitte leise an einen der Tutoren.
- Die Klausur ist definitiv bestanden mit mindestens 45 Punkten. Mit mindestens 95 Punkten ist definitiv die Note 15 erreicht.

Wird bei der Korrektur ausgefüllt:

Aufgabe	1	2	3	4	5	6	7	8	Total
Punkte	10	13	17	6	17	11	11	15	100
Punkte erreicht									

Aufgabe 1: Wissensfragen 10 Punkte

Beantworten Sie die folgenden Fragen in 1–2 kurzen Sätzen!

(a) Beschreiben Sie kurz in eigenen Worten, was es bedeutet, wenn in Prolog ein Term T' die Instanz eines anderen Terms T ist. **2**

(b) Racket verwendet lexikalisches Scoping. Beim Anwenden der LOCAL-Regel, werden die lokalen Definitionen in die globale Umgebung übertragen. Wie wird hierbei lexikalisches Scoping sichergestellt? **2**

(c) Wofür steht das Loch ([ ]) im Auswertungskontext der BSL? **2**

(d) Was sind Magic-Numbers und warum sollten sie vermieden werden? **2**

(e) Welche Bestandteile hat eine Methodensignatur in TypedRacket? **2**

Vorname:	Nachname:	Matrikelnummer:

Aufgabe 2: Ausdrücke, Werte & Reduktion

13 Punkte

Für die Teilaufgaben a) bis c): Welches Ergebnis liefern die folgenden Ausdrücke? Es genügt, wenn Sie das Ergebnis notieren. Die angewendeten Regeln müssen nicht aufgeschrieben werden. Tritt beim Auswerten ein Fehler auf, geben Sie die Ursache des Fehlers an.

(a) **2**

```
(define a 1)
(define b 1)
((if (symbol=? 'a 'b)
     (lambda (x y) (+ x y))
     (lambda (x y) (* x y)))
 a b)
```

(b) `(foldr (lambda (x y) y) '() '(1 2 3 4))` **2**

(c) **2**

```
(define (signum n)
  (cond [(> n 0) 1]
        [(<= n 0) -1]
        [(= n 0) 0]
        [else (error "Ungültiges Argument")]))
(signum 0)
```

(d) In der Umgebung befindet sich die folgende Definition der Funktion `abs` (mit dem Sprachniveau Fortgeschritten + Lambdas, kurz ISL), die den Absolutbetrag für eine Ganze Zahl bestimmt. **5**

```
(define f
  (lambda (x y)
    (cond [(< x y) y]
          [true x])))
```

Reduzieren Sie das folgende Programm schrittweise und **unter Angabe der jeweils verwendeten Regel**!

Hinweis: Es reicht, in jedem Schritt den relevanten Teil des Programms zu notieren. Unveränderte Teile können durch "..." abgekürzt werden. Sie können für diese Aufgabe den ausgeteilten Zettel mit Auswertungs- und Äquivalenzregeln verwenden.

```
(f 3 4)
```

(e) Geben Sie ein Beispiel für ein Racket-Programm an, bei dem ein Laufzeitfehler auftritt. Um was für einen Fehler handelt es sich hierbei? **2**

Aufgabe 3: Funktionsdefinitionen und Signaturen 17 Punkte

(a) Geben Sie die Signatur der folgenden Funktion an. Verwenden Sie soweit nötig Typparameter, welche die Typbeziehungen zwischen den Parametern und dem Rückgabetyp korrekt repräsentieren. Gehen Sie davon aus, dass **alle verwendeten Listen homogen** sind, also nur Werte von einem Typ enthalten. Dabei kommen als Element-Typen einer List **keine** Summentypen in Frage. Begründen Sie alle Teile der Signatur, und nehmen Sie dabei auf die Funktionsdefinition Bezug! **6**

```
(define (f x y z)
  (cond [x (cons y z)]
        [(y x) '()]))
```

(b) Geben Sie für die unten angegebene Funktion mit Signatur folgendes an: eine **Aufgabenbeschreibung** und Testfälle gemäß des zutreffenden Entwurfsrezepts. **3**

```
;(listof Number) Number -> (listof Number)
(define (f a b)
  (match a
    ['() '()]
    [(cons c d) (cons (+ c b) (f d b))]))
```

Wenn Sie Bücher von Terry Pratchett lesen, wissen Sie, Trolle werden oftmals fälschlich für dumm gehalten, da sie so zählen: 1, 2, 3, viele. Tatsächlich sind sie aber intelligent und können zählen, wobei “viele” einfach für die Ziffer 4 steht. Trolle zählen daher nach folgendem Schema: 1, 2, 3, viele, viele1, viele2, viele3, vieleviele, usw.

(c) Definieren Sie einen Datentypen “TrollZiffer” der alle Ziffern des Troll-Zahlensystems umfasst. Geben Sie an, was für eine Art Datentyp Sie definiert haben. **3**

Vorname:	Nachname:	Matrikelnummer:

(d) Eine Troll-Zahl soll nun als Liste von TrollZiffern dargestellt werden, also `(listof TrollZiffer)`. Geben Sie für eine Funktion `(trollzahl->string l)` Testfälle in ausreichender Anzahl (gemäß der zutreffenden Entwurfsrezepte) und die Implementierung an. Die Funktion soll eine Zeichenkette zurückgeben, welche der Zahl in Troll-Schreibweise entspricht (also z.B. `"vieleviele"`). Für die leere List soll die leere Zeichenkette zurückgegeben werden. **5**

Aufgabe 4: Algebraische Datentypen 6 Punkte

Entwerfen Sie einen algebraischen Datentypen `Laufwerk` mit den Alternativen `festplatte` und `sdd` (Solid-State Disk). Eine Festplatte hat als Eigenschaften: die Kapazität (`kapazitaet`) in Gigabyte und Rotationsgeschwindigkeit (`rotation`) in Umdrehungen **pro Minute**. Eine SDD hat als Eigenschaften: die Kapazität (`kapazitaet`) in Gigabyte, die Zugriffszeit (`zugriff`) **in Millisekunden** und die maximale Anzahl von Schreibzugriffen (`max`).

(a) Definieren Sie den algebraischen Datentyp *Laufwerk* mithilfe von `define-type`. **3**

(b) Implementieren Sie eine Funktion, `zugriffszeit`, die für ein Laufwerk die mittlere Zugriffszeit **in Sekunden** berechnet. Für eine Festplatte entspricht die mittlere Zugriffszeit der Zeit für eine halbe Umdrehung. Achten Sie darauf, jeweils die Zeit-Einheiten korrekt umzurechnen. Verwenden Sie zur Fallunterscheidung das `type-case` Konstrukt. **3**

```
(define (zugriffszeit lw)
```

Vorname:	Nachname:	Matrikelnummer:

Aufgabe 5: Äquivalenz 17 Punkte

Betrachten Sie die folgende Funktion (Sprachniveau Anfänger, kurz BSL). Gehen Sie davon aus, dass als Argumente für `n` nur natürliche Zahlen (0, 1, 2, ...) vorkommen:

```
; Number -> Number
(define (ind n)
  (cond [(zero? n) 2]
        [true (- 2 (/ 1 (ind (sub1 n))))]))
```

Beweisen Sie die Äquivalenz `(ind n)` $\equiv$ `(/ (+ n 2) (+ n 1))` durch strukturelle Induktion über `n` **unter Angabe der jeweils verwendeten Regel**! Sie können die folgende zusätzliche Äquivalenz-Regel verwenden:

(EVorgabe) `(- 2 (/ 1 (/ (+ n 2) (+ n 1))))` $\equiv$ `(/ (+ n 3) (+ n 2))`

Hinweis: Es reicht, in jedem Schritt den relevanten Teil des Programms zu notieren. Unveränderte Teile können durch "..." abgekürzt werden. Sie können für diese Aufgabe den ausgeteilten Zettel mit Auswertungs- und Äquivalenzregeln verwenden.

(a) Stellen Sie **die im Induktionsanfang zu beweisende Äquivalenz** auf und **führen Sie den Induktionsanfang** durch. **6**

(b) Stellen Sie die im Induktionsschritt **zu beweisende Äquivalenz** und die **Induktionsannahme** auf. Führen Sie anschließend den **Induktionsschritt** durch. **11**

Vorname:	Nachname:	Matrikelnummer:

Aufgabe 6: Listen und Funktionen höherer Ordnung 11 Punkte

Hinweis: Die folgenden aus der Vorlesung bekannten Funktionen höherer Ordnung dürfen Sie in Aufgabenteil a) verwenden:

```
; [X] (X -> Boolean) (listof X) -> (listof X)
; Gibt eine Liste zurück mit allen Elementen aus l, die das Prädikat p erfüllen
(filter p l)

; [X Y] (X -> Y) (listof X) -> (listof Y)
; Bildet alle Elemente aus l mit f ab und liefert die Liste der Ergebnisse.
(map f l)

; [X Y] (X Y -> Y) Y (listof X) -> Y
; Kombiniert alle Elemente der Liste l durch f, wobei das letzte Element mit
; base kombinitert wird. Die leere Liste wird auf base abgebildet, die Elemente
; werden von rechts nach links durchlaufen.
(foldr f base l)
```

(a) Definieren Sie die nachfolgende Racket-Funktion **nur durch Aufrufe der oben genannten Funktionen höherer Ordnung**. Das heißt, der Body-Ausdruck Ihrer Funktion **darf keine Aufrufe an andere primitive oder selbst-definierte Funktionen enthalten**. Primitive oder selbst-definierte Funktionen dürfen aber als Argument an die Funktionen höherer Ordnung übergeben werden. **Hinweis: Die Racket Funktion** `(string->number s)` **gibt den entsprechenden Zahlenwert zurück, wenn** `s` **eine Zahl darstelle und** `false` **sonst.** **4**

```
; (listof String) -> Number
; Interpretiert alle Elemente aus l als Zahlen und gibt deren Summe zurück.
(check-expect (mysum '("1" "2")) 3)
(check-expect (mysum '()) 0)

(define (mysum l)
```

(b) Geben Sie eine mögliche Implementierung der Funktion `insert` (siehe unten) an. Greifen Sie hierbei **nicht** auf vordefinierte Funktionen höherer Ordnung zurück. **Verwenden Sie** `match` **für die Implementierung.** **7**

```
; [X] (X-> Boolean) X (listof X)  -> (listof X)
; Gibt eine Liste zurück, die alle Elemente aus l und a enthält, wobei
; der Wert a hinter dem ersten Element in l eingefügt ist, das das Prädikat
; p erfüllt. Wenn kein solches Element in der Liste enthalten ist, wird a
; als letztes Element eingefügt.
(check-expect (insert (lambda (x) (= x 1)) 5 '(0 1 2)) '(0 1 5 2))

(define (insert p a l)
```

Vorname:	Nachname:	Matrikelnummer:

Aufgabe 7: Akkumulatoren & Terminierung 11 Punkte

(a) Implementieren Sie eine Funktion `b->n` (das steht für "binary to number"), welche eine Liste aus binären Ziffern (gehen Sie davon aus, dass nur die Werte `0` und `1` als Listenelemente vorkommen) in eine Number umrechnet. Dabei steht das erste Element in der Liste für die höchstwertige Stelle (also $2^{(i-1)}$ bei einer Liste mit i Elementen) und das letzte Element für die niederwertigste Stelle (also 2^0). **Benutzen Sie in der Implementierung einen Akkumulator**. Geben Sie die **Akkumulatorinvariante** an und verwenden Sie sinnvoll **lokale Definitionen**. **8**

Hinweis: Die Racket-Funktion `(expt b e)` berechnet die Funktion b^e. Die Racket-Funktion `(length l)` gibt die Länge der Liste `l` zurück.

```
(define (b->n l)
```

(b) Betrachten Sie die folgende Funktion zur Berechnung des größten gemeinsamen Teilers zweier natürlicher Zahlen. **3**

```
(define (gcd a b)
  (cond [(= b 0) a]
        [else (gcd b (modulo a b))]))
```

Begründen Sie, warum die Funktion `gcd` stets terminiert (gehen Sie davon aus, dass a und b immer vom Typ Number sind und nur natürliche Zahlen enthalten).

Aufgabe 8: Prolog 15 Punkte

(a) Definieren Sie die folgende Relation in Prolog mit Hilfe von Rekursion und Pattern-Matching. Die Verwendung von Bibliotheks-Relationen aus Prolog ist dabei **nicht** gestattet. Definieren Sie benötigte Hilfsrelationen selbst. **5**

Hinweis: Die arithmetischen Vergleichsprädikate <, >, =<, >=, == (Gleichheit), \= (Ungleichheit) dürfen verwendet werden und werden in Prolog infix geschrieben, zum Beispiel: `?- 2 =< 3.`

`sieve(L1, T, L2)`: L2 ist die Liste, die alle Elemente aus der Liste L1 enthält, welche größer sind als T.

Vorname:	Nachname:	Matrikelnummer:

Gegeben seien die folgenden Definitionen der Relationen b – e und die Abfragen in Teilaufgaben b) – e), die diese verwenden. Geben Sie für jede Abfrage an, was das Resultat ist. Wenn eine Abfrage erfüllt ist, geben Sie eine gültige Substitution aller Variablen an. In allen anderen Fällen geben Sie eine kurze Begründung an.

```
b(X, Y, Z, [X, Y, Z]).

c(id(X), X, X, X).
c(sw(Op),X, Y, Z) :- Y < X, c(Op, Y, X, Z).
c(dec(Op), X, Y, Z) :- X < Y, U is Y - 1, c(Op, X, U, V), Z is V + 1.

d([Erde, Wasser, Feuer, Luft]).

e([apfel, birne, pflaume]).
```

(b) `?- b(u, v, w, L).` **2**

(c) `?- c(Desc, 4, 3, R).` **4**

(d) `?- d([erde, wasser | [feuer, luft]]).` **2**

(e) `?- e([Apfel, Birne, Pflaume, Obst]).` **2**