Lösung Probeklausur Informatik I

Lösung Aufgabe 1 (5 Punkte) ALGORITHMEN UND PROGRAMME

Was ist der Unterschied zwischen einem Algorithmus und einem Programm?

Lösung:

Ein Algorithmus ist eine Vorschrift zur Durchführung einer Berechnung, wobei es nicht darauf ankommt, wie diese Vorschrift notiert ist, sondern vielmehr auf den Inhalt der Vorschrift. Ein Programm ist ein Algorithmus notiert in einer Programmiersprache.

Lösung Aufgabe 2 (5+5 Punkte) LEXIKALISCHE BINDUNG

(a) Was ist die Ausgabe des folgenden Scheme-Programms?

```
(define x 1)
(define y 5)

((lambda (x y)
   (+ (* 2 x) y))
 y x)

((lambda (a b)
   (+ (* 2 x) y))
 y x)
```

Lösung:
Die Ausgabe ist

```
11
7
```

(b) Erklären Sie kurz, wodurch sich die beiden Ausdrücke unterscheiden.

Lösung:
Gemäß der lexikalischen Bindung ist der Wert von `x` im Rumpf des ersten Ausdrucks gerade der Wert von `y` außerhalb des Ausdrucks, also 5, und umgekehrt, der Wert von `y` im Rumpf des ersten Ausdrucks ist gerade der Wert von `y` außerhalb des Ausdrucks, also 1. Demnach wird `(+ (* 2 x) y)` zu 11 ausgewertet.

Beim zweiten Ausdruck werden die Argumente des lambda-Ausdrucks weggeworfen, und der Wert von `x` und `y` im Rumpf des Ausdrucks ist gerade der entsprechende Wert außerhalb des Ausdrucks, also 1 bzw. 5. Demnach wird `(+ (* 2 x) y)` zu 7 ausgewertet.

Lösung Aufgabe 3 (5+5 Punkte) FALLUNTERSCHEIDUNGEN

(a) Schreiben Sie einen Scheme-Ausdruck, der den Absolutbetrag einer Zahl n berechnet.

Lösung:

```
(if (>= n 0) n (- n)) oder

(cond
  ((>= n 0) n)
  ((< n 0) (- n)))
```

(b) Erweitern Sie den Ausdruck zu einer Scheme-Prozedur, die den Absolutbetrag einer Zahl berechnet.

Lösung:

```
(define absolute
  (lambda (n)
    (cond
      ((>= n 0) n)
      ((< n 0) (- n)))))
```

Lösung Aufgabe 4 (5+3 Punkte) RECORD-DATENTYPEN

Ein Schokokeks in Scheme ist durch folgende Record-Definition definiert:

```
(define-record-procedures chocolate-cookie
  make-chocolate-cookie chocolate-cookie?
  (chocolate-cookie-chocolate
   chocolate-cookie-cookie))
```

(a) Ergänzen Sie das folgende Fragment einer Prozedur, die das Gewicht eines Schokokekses berechnet (wenn Sie davon ausgehen, dass ein Schokokeks exakt so viel wiegt wie seine beiden Anteile):

```
(define chocolate-cookie-weight
  (lambda (c)
    (
                                    )))
```

Lösung:

```
(define chocolate-cookie-weight
  (lambda (c)
    (+ (chocolate-cookie-chocolate c)
       (chocolate-cookie-cookie c))))
```

(b) Erklären Sie kurz, welcher Konstruktionsanleitung Sie gefolgt sind.

Lösung:
Dies entspricht der Konstruktionsanleitung für Prozeduren, die zusammengesetzte Daten konsumieren. Die Anleitung besagt, dass im Rumpf die Selektoren für den Record `chocolate-cookie` verwendet werden sollen.

Lösung Aufgabe 5 (8 Punkte) LISTEN

Die Sorte der Zahlenlisten soll `zahlenliste` heißen (d. h., die Sorte der Elemente ist `number`). Um sie zu definieren, haben wir folgende Record-Definition für die Sorte `nll` der *nichtleeren* Zahlenlisten erstellt:

```
(define-record-procedures nll
  kons nichtleer?
  (kopf rumpf))
(: kons                                          )
(: nichtleer?                                    )
(: kopf                                          )
(: rumpf                                         )
```

Ergänzen Sie die unvollständigen Signaturen.

Lösung:

```
(define-record-procedures nll
  kons nichtleer?
  (kopf rumpf))
(: kons (number zahlenliste -> nll))
(: nichtleer? (%value -> boolean))
(: kopf (nll -> number))
(: rumpf (nll -> zahlenliste))
```

Lösung Aufgabe 6 (12 Punkte) SIGNATUREN

Betrachten Sie folgendes Scheme-Programmstück, in dem für einige Bezeichner Signaturen und Definitionen angegeben werden:

```
(: list-1 (list number))
(define list-1 (cons "Banana" (cons 1 empty)))

(: list-2 (list number))
(define list-2 (cons 1 (cons 4 empty)))

(: list-3 (list string))
(define list-3 (cons "Apple" (cons "Banana" empty)))

(: list-4 (list (list string)))
(define list-4 (cons list-3 empty))

(: list-5 (list string))
(define list-5 list-4)

(: list-6 (list %value))
(define list-6 (cons 1 (cons "Banana" empty)))
```

Kennzeichnen Sie, an welchen Stellen Signaturverletzungen vorliegen, jeweils mit einer kurzen Erklärung. Kennzeichnen Sie Stellen, an denen *keine* Signaturverletzungen vorliegen, durch Abhaken.

Lösung:

```
(: list-1 (list number))
(define list-1 (cons "Banana" (cons 1 empty)))
```

Signaturverletzung wegen `"Banana"`, das nicht von der Sorte `number` ist.

```
(: list-5 (list string))
(define list-5 list-4)
```

Signaturverletzung, da `list-4` die Sorte `(list (list string))`, hingegen `list-5` die Sorte `(list string)` hat.

Lösung Aufgabe 7 (4 Punkte) REKURSION

Was berechnet die Prozedur `spam`?

```
(: spam (natural -> natural))
(define spam
  (lambda (n)
    (if (zero? n)
        1
        (* n (spam (- n 1))))))
```

Lösung:
Die Prozedur `spam` berechnet die Fakultätsfunktion: $f(n) = \Pi_{i=1}^{n} i$.

Lösung Aufgabe 8 (4+4 Punkte) LOKALE VARIABLEN

(a) Was ist die Ausgabe des folgenden Scheme-Programms?

```
(define a 5)

(let ((a 1)
      (b (+ a 1)))
  b)

(let* ((a 1)
       (b (+ a 1)))
  b)
```

Lösung:
Die Ausgabe ist

```
6
2
```

(b) Erklären Sie kurz, wodurch sich die beiden Ausdrücke unterscheiden.

Lösung:
Ein `let` mit mehreren Variablen bindet alle Variablen gleichzeitig, daher ist die Bindung von `a` an 5 maßgeblich. Bei `let*` erfolgt die Bindung sequentiell, daher ist die Bindung von `a` an 1 maßgeblich.

Lösung Aufgabe 9 (2+1+1+1 Punkte) EIGENSCHAFTEN VON RELATIONEN

Sei $A = \{1, 2, 3\}$ und $R = \{(1, 2), (2, 3), (1, 3)\} \subseteq A \times A$ eine binäre Relation über A.

(a) Geben Sie zu R die Umkehrrelation R^{-1} an.

Lösung:
$R^{-1} = \{(2, 1), (3, 2), (3, 1)\}$.

(b) Ist R reflexiv?

Lösung:
Nein.

(c) Ist R transitiv?

Lösung:
Ja.

(d) Ist R symmetrisch?

Lösung:
Nein.

Lösung Aufgabe 10 (5 Punkte) TRANSITIVE HÜLLE

Sei $A = \{1, 2, 3, 4\}$. Geben Sie zur binären Relation $R = \{(1, 2), (2, 3), (3, 4)\} \subseteq A \times A$ die transitive Hülle an, d. h., die kleinste Relation, die R enthält und transitiv ist.

Lösung:

Die transitiive Hülle ist $\{(1, 2), (2, 3), (3, 4), (1, 3), (1, 4), (2, 4)\}$.

Lösung Aufgabe 11 (10 Punkte) INDUKTION

Beweisen Sie mittels vollständiger Induktion, dass für alle $n \in \mathbb{N}$ gilt:

$$\sum_{i=1}^{n} \frac{i}{(i+1)!} = 1 - \frac{1}{(n+1)!}$$

Lösung:

Zu beweisen:

$$\sum_{i=1}^{n} \frac{i}{(i+1)!} = 1 - \frac{1}{(n+1)!}$$

Induktionsbasis ($n = 0$):

$$\sum_{i=1}^{0} \frac{i}{(i+1)!} = 0 = 1 - \frac{1}{(0+1)!}$$

Induktionsschritt ($P(n) \Rightarrow P(n+1)$ für alle $n \in \mathbb{N}$):

$$\sum_{i=1}^{n+1} \frac{i}{(i+1)!} = \frac{n+1}{(n+2)!} + \sum_{i=1}^{n} \frac{i}{(i+1)!} \overset{\text{IH}}{=} \frac{n+1}{(n+2)!} + 1 - \frac{1}{(n+1)!} =$$
$$1 - \frac{(n+2)-(n+1)}{(n+2)!} = 1 - \frac{1}{(n+2)!}$$

Lösung Aufgabe 12 (5+2 Punkte) TERMINIERUNG

(a) Wird folgende Prozedur auf der Eingabe 10 terminieren? (Verschiedene Antworten sind möglich, es kommt auf eine plausible Erklärung an. Achten Sie auf die Signatur!)

```
(: l (number -> number))
(define l
  (lambda (n)
    (if (zero? n)
        0
        (+ 1 (l (/ n 2))))))
```

Lösung:
Zumindest theoretisch wird diese Prozedur auf der Eingabe 10 nicht terminieren, da eine (reelle) Zahl unendlich oft durch 2 dividiert werden kann, d.h. $\{(\frac{x}{2}, x)\}$ ist keine wohlfundierte Ordnung. In der Praxis dürfte die Zahlendarstellung eine Rolle spielen; evtl. wird eine sehr kleine Zahl irgendwann mit der 0 identifiziert und die Prozedur bricht ab.

(b) Korrigieren Sie das Programm derart, dass es $l : \mathbb{N} \to \mathbb{N}$ mit $l(n) = \lceil \log_2(n+1) \rceil$ berechnet.

Lösung:
Die korrigierte Prozedur benutzt `quotient` und ist nur auf den natürlichen Zahlen definiert:

```
(: l (natural -> natural))
(define l
  (lambda (n)
    (if (zero? n)
        0
        (+ 1 (l (quotient n 2))))))
```

Lösung Aufgabe 13 (4+4 Punkte) PROGRAMMIERUNG HÖHERER ORDNUNG

(a) Die Prozedur zum Falten einer Liste sieht folgendermaßen aus:

```
(: list-fold (_ (_ _ -> _) (list %b) -> %a))
(define list-fold
  (lambda (e f l)
    (cond
     ((empty? l)
      e)
     ((cons? l)
      (f (first l) (list-fold e f (rest l)))))))
```

Ergänzen Sie die fehlenden Sorten (angedeutet durch _) in der Signatur.

Lösung:

```
(: list-fold (%a (%b %a -> %a) (list %b) -> %a))
```

(b) Man kann Listenfaltung verwenden, um die Konkatenation von Listen zu definieren:

```
(: append ((list %a) (list %a) -> (list %a)))
(define append
  (lambda (xs ys)
    (
                                          )))
```

Ergänzen Sie den fehlenden Rumpf der Definition.

Lösung:

```
(define append
  (lambda (xs ys)
    (list-fold ys cons xs)))
```