Name: Matrikelnummer:

Wintersemester 2011/2012

Probeklausur Informatik I

Es sind keine Hilfsmittel zugelassen.
Diese Klausur hat 8 Aufgaben auf insgesamt 11 Seiten inklusive Deckblatt.
Insgesamt können 60 Punkte erreicht werden.
Sprachebene: Die Macht der Abstraktion - fortgeschritten.
Im Rahmen dieser Klausur sind keine Testfälle zu spezifizieren.
Tragen Sie Namen und Matrikelnummer auf jedem Aufgabenblatt ein.

Der Rest dieser Seite ist ausschließlich vom Korrekturperonal zu beschreiben.

Aufgabe	Punktzahl
1	
2	
3	
4	
5	
6	
7	
8	

1. [9 Punkte] Schreiben sie eine Prozedur `cycle`, die eine Zahl $n \in \mathbb{N}$ und eine Liste mit beliebigen Elementen als Parameter konsumiert. Die Prozedur soll n Mal jeweils das letzte Element der Liste an den Anfang verschieben und die resultierende Liste zurückgeben.
   Wichtig: Verwenden sie **nicht** `reverse` für diese Aufgabe!

   ```
   Beispiele:
   (cycle 0 (list 1 2 3 4 5 6 7))
     => #<list 1 2 3 4 5 6 7>
   (cycle 3 (list 1 2 3 4 5 6 7))
     => #<list 5 6 7 1 2 3 4>
   (cycle 9 (list "a" "u" "s" "u" "r" "K" "l"))
     => #<list "K" "l" "a" "u" "s" "u" "r">
   ```

   Gehen sie dazu wie folgt vor:

   a) Schreiben sie eine Prozedur `last`, die eine nicht-leere Liste konsumiert und das letzte Element der Liste zurückgibt.

   b) Schreiben sie eine Prozedur `without-last`, die eine nicht-leere Liste konsumiert und die Liste ohne das letzte Element zurückgibt.

   c) Schreiben sie eine Prozedur `cycle`, die die Prozeduren `last` und `without-last` verwendet.

   Beachten sie: Die Prozedur `cycle` muss mit allen Zahlen $n \geq 0$ und beliebigen Listen (inklusive der leeren Liste) funktionieren!

Lösung

```
1  ;Das Letzte Element zurückgeben
2  (: last ((list-of %a) -> %a))
3
4  (check-expect (last (list 1 2 3))
5                3)
6  (check-expect (last (list 5))
7                5)
8  (define last
9    (lambda (lis)
10     (cond
11       ((empty? (rest lis)) (first lis))
12       ((pair? (rest lis))
13        (last (rest lis))))))
14
15 ;Alle Elemente bis auf das Letzte zurückgeben
16 (: without-last ((list-of %a) -> (list %a)))
17
18 (check-expect (without-last (list 1 2 3))
19               (list 1 2))
20 (check-expect (without-last (list 1))
21               empty)
22
23 (define without-last
24   (lambda (lis)
25     (cond
26       ((empty? (rest lis)) empty)
27       ((pair? (rest lis))
28        (make-pair (first lis)
29                   (without-last (rest lis)))))))
30
31 ;Elemente durchwechseln
```

```
32 (: cycle (natural (list-of %a) -> (list-of %a)))
33
34 (check-expect (cycle 0 (list 1 2 3 4 5 6 7))
35                (list 1 2 3 4 5 6 7))
36 (check-expect (cycle 3 (list 1 2 3 4 5 6 7))
37                (list 5 6 7 1 2 3 4))
38 (check-expect (cycle 9 (list "a" "u" "s" "u" "r" "K" "l"))
39                (list  "K" "l" "a" "u" "s" "u" "r"))
40
41 (define cycle
42   (lambda (n lis)
43     (cond
44          ((empty? lis) empty)
45      ((zero? n) lis)
46      (else (cycle (- n 1)
47                   (make-pair (last lis)
48                              (without-last lis)))))))
```

2. [8 Punkte] Zeigen Sie per Induktion, mit Hilfe der angegebenen Definitionen für `drop` und `take`, dass folgende Gleichung gilt: (Sie können davon ausgehen, dass `append` assoziativ ist):

```
1  ;(n ist eine natürliche Zahl grösser als 1)
2  (append (take n lis) (drop n lis)) = lis
3
4  (define take
5    (lambda (n lis)
6      (if (= n 0)
7          empty
8          (cond
9            ((empty? lis) empty)
10           ((pair? lis) (append (list (first lis))
11                                (take (- n 1)
12                                      (rest lis)))))))))
13
14 (define drop
15   (lambda (n lis)
16     (if (= n 0)
17         lis
18         (cond
19           ((empty? lis) empty)
20           ((pair? lis) (drop (- n 1)
21                              (rest lis)))))))
```

Lösung

```
I.A.: lis = empty
        (append (take n lis) (drop n lis))
=>      (append (take n empty) (drop n empty))
=>      (append empty empty)
=>                                  empty
I.V.: (append (take n lis) (drop n lis)) = lis
I.S.: lis -> lis* mit lis* = (make-pair x lis)
        (append (take n lis) (drop n lis))
=>      (append (append (list (first lis*)) (take (- n 1) (rest lis*)))
                  (drop (- n 1) (rest lis*)))           sei m = n - 1
=>      (append (append (list x) (take m lis))
=>                (drop m lis))
=> (append ist assoziativ)
        (append (list x) (append (take m lis) (drop m lis))
I.V.=>  (append (list x) lis)
=> (make-pair x lis)
=> lis*

q.e.d.
```

3. [10 Punkte] Im Folgenden programmieren sie einige Prozeduren auf Streams. Dafür können sie folgenden Code verwenden:

```
1  Ein Stream besteht aus
2  ; - einem ersten Element (head)
3  ; - einem Promise, den Rest des Stream generieren zu können (tail)
4  (: make-stream (%a (() -> (stream %a)) -> (stream %a)))
5  (: stream-head ((stream %a) -> %a))
6  (: stream-tail ((stream %a) -> (() -> (stream %a))))
7
8  (define-record-procedures-parametric stream_ stream-of
9   make-stream stream?
10  (stream-head stream-tail))
11
12 ;Vertrag für potentiell unendliche Streams
13 (define stream
14  (lambda (v)
15   (contract (stream-of v (promise (stream v))))))
```

(a) Schreiben sie eine Prozedur `constant-stream`, die einen Stream aus einer konstanten Zahl erzeugt.
Bsp.:

```
(take-stream (constant-stream 3) 5) ~~> (list 3 3 3 3 3)
```

(b) Schreiben sie eine Prozedur `line-stream`, die die Funktion $f(x) = x + c$ als Stream darstellt ($x \geq 0$). c wird `line-stream` übergeben.
Bsp.:

```
(take-stream (line-stream 3) 5) ~~> (list 3 4 5 6 7)
```

(c) Schreiben sie eine Prozedur `intersection`, die zwei Streams akzeptiert und den X-Wert des Schnittpunktes berechnet.

(d) Geben sie einen Ausdruck an, der für die Gerade $f(x) = x - 3$ den Schnittpunkt mit der X-Achse berechnet.

Lösung

```
1
2  ;Erzeugt einen konstanten Stream aus Zahlen
3  (: constant-stream (real -> (stream real)))
4
5  (check-expect (take-stream (constant-stream 5) 3) (list 5 5 5))
6  (check-expect (take-stream (constant-stream 1.1) 2) (list 1.1 1.1))
7
8  (define constant-stream
9    (lambda (n)
10     (make-stream n
11                  (lambda ()
12                    (constant-stream n)))))
13
14 ;Eine Gerade als Stream
15 (: line-stream (real -> (stream real)))
16
17 (check-expect (take-stream (line-stream 3) 5) (list 3 4 5 6 7))
18
19 (define line-stream
20   (lambda (c)
21     (make-stream c
22                  (lambda ()
23                    (line-stream (+ c 1))))))
24
25 ;Schnittpunkt zweier Streams berechnen
26 (: intersection ((stream real) (stream real) -> natural))
27
28 (check-expect (intersection (constant-stream 5)
29                             (line-stream 0))
30               5)
31
32 (define intersection
33   (lambda (str1 str2)
34     (if (= (stream-head str1)
35            (stream-head str2))
36         0
37         (+ 1 (intersection (force (stream-tail str1))
38                            (force (stream-tail str2)))))))
39
40 (intersection (constant-stream 0)
41               (line-stream -3))
```

4. [10 Punkte] Betrachten Sie folgende unvollständige Scheme-Prozedur:

```
(define square
 (lambda (n)
  (if (= n 0)
    0
    (if (even? n)
      (________ (square (/ n 2))
               __________________)
      (+ (square (- n 1))
         (- (+ n n) 1))))))
```

Vervollständigen Sie die Definition, so dass `square` korrekt das Quadrat jeder nicht-negativen ganzen Zahl berechnet! Beweisen Sie die Korrektheit des Ergebnisses!

Lösung

```
(define square
 (lambda (n)
  (if (= n 0)
    0
    (if (even? n)
      (* (square (/ n 2))
              4)
      (+ (square (- n 1))
         (- (+ n n) 1))))))


Korrektheitsbeweis:

z.Z.: (square n) = n^2

I.A.: n = 0
(square n) = 0 = 0^2

I.V. (square n) = n^2
I.S. n -> n+1

(square (+ n 1))

1. Fall n+1 gerade
(* (square (/ (n+1) 2))
    4)
= I.V.  (* ((n+1)/2)^2 * 4)
= (* ((n+1)^2/4) * 4)
= (n+1)^2

2. Fall n+1 ungerade
(+ (square (- (+ n 1) 1))
        (- (+ (n+1) (n+1)) 1))
= (+ (square n)
      (- (* 2(n+1)) 1))
= (+ n^2 + 2n + 2 - 1)
= (+ n^2 + 2n +1)
= (n+1)^2

q.e.d
```

5. [6 Punkte] Schreiben sie folgende Prozeduren über Bäume. Benutzen sie den angegebenen Code.

   a) Eine Prozedur, die die Tiefe eines Baumes berechnet

   b) Eine Prozedur, die die Anzahl der leeren Teilbäume berechnet

```
1  ; Ein Knoten (node) besitzt
2  ; - einen linken Zweig (left-branch),
3  ; - eine Markierung (label) und
4  ; - einen rechten Zweig (right-branch)
5  (: make-node (%a %b %c -> (node-of %a %b %c)))
6  (: node-left-branch ((node-of %a %b %c) -> %a))
7  (: node-label ((node-of %a %b %c) -> %b))
8  (: node-right-branch ((node-of %a %b %c) -> %c))
9
10 (define-record-procedures-parametric node node-of
11   make-node node?
12   (node-left-branch node-label node-right-branch))
13
14 ; Ein leerer Baum (empty-tree) besitzt
15 ; keine weiteren Eigenschaften
16 (: make-empty-tree (-> empty-tree))
17 (define-record-procedures empty-tree
18   make-empty-tree empty-tree?
19   ())
20
21 ; Der leere Baum
22 (: the-empty-tree empty-tree)
23 (define the-empty-tree (make-empty-tree))
24
25 ; Vertrag für Binärbäume (btree v) mit Markierungen des Vertrags v
26 ; (im linken/rechten Zweig jedes Knoten findet sich jeweils wieder
27 ; ein Binärbaum)
28 (define btree
29   (lambda (v)
30     (contract (mixed empty-tree
31                      (node-of (btree v) v (btree v))))))
32
33 ; Konstruiere ein Blatt mit Markierung x (: make-leaf (%a -> (btree %a)))
34 (define make-leaf
35   (lambda (x)
36     (make-node the-empty-tree x the-empty-tree)))
```

(Platz zum bearbeiten der Aufgabe)

Lösung

```
1  ;Tiefe eines Baumes berechnen
2  (: tree-depth ((btree %a) -> natural))
3
4  (check-expect (tree-depth the-empty-tree) 0)
5  (check-expect (tree-depth (make-node (make-leaf "a")
6                                       12
7                                       (make-node (make-leaf "c")
8                                                  #f
9                                                  the-empty-tree))) 3)
10
11 (define tree-depth
12   (lambda (bt)
13     (cond
14       ((empty-tree? bt) 0)
15       ((node? bt) (+ 1 (max (tree-depth (node-left-branch bt))
16                             (tree-depth (node-right-branch bt)))))))))
17
18 ;Anzahl der leeren Teilbäume berechnen
19 (: empty-trees ((btree %a) -> natural))
20
21 (check-expect (empty-trees the-empty-tree) 1)
22 (check-expect (empty-trees (make-node (make-leaf "a")
23                                       12
24                                       (make-node (make-leaf "c")
25                                                  #f
26                                                  the-empty-tree))) 5)
27
28 (define empty-trees
29   (lambda (bt)
30     (cond
31       ((empty-tree? bt) 1)
32       ((node? bt) (+ (empty-trees (node-left-branch bt))
33                      (empty-trees (node-right-branch bt)))))))
```

6. [7 Punkte] Betrachten Sie die folgenden λ -Terme:

$$(((\lambda x.(\lambda y.((yx)x)))(\lambda z.z))(\lambda y.(\lambda z.((yy)(yz)))))$$

$$((\lambda z.(\lambda x.((\lambda y.z)(xy)))(\lambda z.((zx)x)))$$

$$(((\lambda z.z)((((\lambda x.(xx))(\lambda x.y))(\lambda y.x))))(\lambda x.x))$$

a) Unterstreichen Sie in den λ-Termen alle β-Redexe.

b) Welches sind jeweils die freien und gebundenen Variablen der Terme?

c) Wandeln sie alle Terme in ihre Normalform um.

Lösung

a.) Gebundene Variablen: {x,y,z}
Freie Variablen: {}
Beta Redex: $((\lambda x.(\lambda y.((yx)x)))(\lambda z.z))$
Ergebnis: x
Schritte:
$\rightarrow ((\lambda y.((y(\lambda z.z))x))((\lambda y.((\lambda z.(yy)))(yz))))$
$\rightarrow (((\lambda y.((\lambda z.(yy))(yz))(\lambda z.z))x)$
$\rightarrow ((\lambda z.(((\lambda z.z)(\lambda z.z))((\lambda z.z)z))x)$
$\rightarrow ((\lambda z.((\lambda z.z)((\lambda z.z)z))x)$
$\rightarrow ((\lambda z.((\lambda z.z)z))x)$
$\rightarrow ((\lambda z.(z))x)$
$\rightarrow x$

b.) Gebundene Variablen: {x,y,z}
Freie Variablen: {x}
Beta Redex: $((\lambda z.(\lambda x.((\lambda y.z)(xy)))(\lambda z.((zx)x)))$
und $((\lambda y.z)(xy))$
Ergebnis: $(\lambda x.(\lambda z.((zx)x)))$
Schritte:
$\rightarrow ((((\lambda z.(\lambda x.z))(\lambda z.((zx)x)))$
$\rightarrow (\lambda x.(\lambda z.((zx)x)))$

c.) Gebundene Variablen: {x,z}
Freie Variablen: {x,y}
Beta Redex: $((\lambda z.z)((((\lambda x.(xx))(\lambda x.y))(\lambda y.x))))$
und $((\lambda x.(xx))(\lambda x.y))$
Ergebnis: $((y(\lambda y.x))(\lambda x.x))$
Schritte:
$\rightarrow ((((\lambda x.(xx))(\lambda x.y))(\lambda y.x)))(\lambda x.x))$
$\rightarrow ((((\lambda x.y)(\lambda x.y))(\lambda y.x))(\lambda x.x))$
$\rightarrow (((y(\lambda y.x))(\lambda x.x))$

7. [4 Punkte] Bestimmen Sie den Vertrag für die folgenden Funktionen.

```
1
2  (define c
3    (lambda (f g h)
4      (letrec ((a (g h)) ( b (f a)))
5        b)))
6
7
8  (define ???
9    (lambda (a b c)
10     (lambda (f)
11       (a (b (lambda (f) (f c)))))))
```

Lösung

```
1  (: c ((%b -> %c) (%a -> %b) %a -> %c))
2
3  (: ??? ((%c -> %d) ((%a -> %b) -> %c) %a -> %d))
```

8. [6 Punkte] Betrachten sie die angegebene Prozedur. Ist die endrekursiv? Warum? Wenn Ja, schreibe sie eine nicht-endrekursive Variante, sonst eine endrekursive.

```
1 (define power
2   (lambda (base exponent)
3     (if (= exponent 0) 1
4         (* base (power base (- exponent 1))))))
```

Lösung

Die Prozedur ist nicht-endrekursiv, da sie beim multiplizieren einen Kontext aufbaut (* base (* base ... (* base 1))). Endrekursive Prozeduren bauen keinen Kontext auf -> Die Prozedur ist nicht-endrekursiv.

```
1   (lambda (base exponent)
2     (letrec ((helper (lambda (b e acc)
3                        (if (= e 0)
4                            acc
5                            (helper b
6                                    (- e 1)
7                                    (* acc base))))))
8       (helper base exponent 1))))
```