Grundlagen der Informatik 1
WS 2008/09

Prof. Mühlhäuser, Dr. Rößling, Melanie Hartmann, Daniel Schreiber
http://proffs.tk.informatik.tu-darmstadt.de/gdi1

Übung 5 - Lösungsvorschlag Version: 1.0 17.11.2008

---

1 Mini Quiz

Kreuzen Sie die wahren Aussagen an!

1. ☒ lambda-Ausdrücke sollten verwendet werden, wenn eine Prozedur nicht rekursiv ist und nur einmal als Argument einer anderen Prozedur gebraucht wird.
2. ☒ Strukturell rekursive Prozeduren terminieren naturgemäß.
3. ☐ Prozeduren mit Gedächtnis können nur mit `lamda` Ausdrücken erzeugt werden.
4. ☐ Generative Rekursion ist effizienter als strukturelle Rekursion.
5. ☒ Jede strukturell rekursive Funktion ist auch generativ rekursiv.

2 Fragen

1. Was ist der Unterschied zwischen generativer und struktureller Rekursion? Nennen Sie für jede Art von Rekursion ein Anwendungsbeispiel.
2. Welche Vorgehensweise verfolgt ein Backtracking-Algorithmus?

Lösungsvorschlag:

1. *Generative Rekursion ist eine Form der Rekursion, welche ein Problem in mehrere Teilprobleme unterteilt. Diese Teilprobleme sollten dabei so beschaffen sein, dass sie leichter zu lösen sind als das Ausgangsproblem. Die Teilprobleme wiederum werden dann einzeln nach dem gleichen Schema behandelt. Dies geht so lange weiter, bis man ein (Teil-)Problem hat, welches sehr einfach ('trivial') zu lösen ist. Wenn alle Teilprobleme gelöst sind, müssen schließlich die Teillösungen zur Gesamtlösung zusammengesetzt werden.*

   Das trivial lösbare Problem wird als sog. Rekursionsanker benötigt. Ohne diesen würde man die (Teil-) Probleme immer weiter in kleinere Probleme teilen, ohne jemals zu einem Ende zu kommen (siehe hierzu auch die Newton-Aufgabe).

   Strukturelle Rekursion beruht auf einer systematischen Analyse der Eingabedaten. Auch hier wird ein Problem in Teilprobleme aufgeteilt, und zwar gemäß der Struktur der Eingabedaten. Strukturelle Rekursion kann als ein Spezialfall der generativen Rekursion betrachtet werden, der zu naturgemäß

terminierenden Prozeduren führt. Strukturelle Rekursion ist oft einfacher zu entwerfen und zu verstehen als generative Rekursion.

Ein Beispiel für einen strukturell rekursiven Algorithmus, ist eine Prozedur, die zwei Listen miteinander addiert und so eine neue (ebenso langen) Liste zurückgibt. Rekursive Sortierverfahren verwenden häufig generativ rekursive Algorithmen.

2. *Ein Backtracking Algorithmus*
   a) *verfolgt einen potentiellen Lösungsweg, bis die Lösung gefunden wurde (Ende) oder der Weg nicht fortgesetzt werden kann.*
   b) *Wenn der Weg nicht fortgesetzt werden kann, geht er den Weg bis zur nächsten Verzweigungsmöglichkeit zurück, an der es noch nicht gewählte Alternativen gibt. Diese Alternativen werden verfolgt wie in Schritt 1 beschrieben.*
   c) *Wenn der Ausgangspunkt erreicht wird, dann gibt es keine weiteren Alternativen mehr: Der Algorithmus terminiert, ohne eine Lösung zu finden.*

3 Generative vs. strukturelle Rekursion

Die in der Vorlesung vorgestellte Vorlage für rekursive Algorithmen sieht wie folgt aus:

```
1 (define (recursive-fun problem)
2   (cond
3     [(trivially-solvable? problem)
4      (determine-solution problem)]
5     [else
6      (combine-solutions
7       problem
8       (recursive-fun (generate-problem problem)))]))
```

1. Für welche Art von Rekursion ist diese Vorlage gedacht?
2. Die Funktion `recursive-fun` soll folgenden Vertrag haben: `recursive-fun : (listof X) -> number`. Sie soll die Länge der übergebenen Liste berechnen. Ändern Sie dazu nicht die Definition von `recursive-fun`, sondern definieren Sie diese vier Funktionen auf geeignete Weise:
   - `trivially-solvable?`
   - `determine-solution`
   - `combine-solutions`
   - `generate-problem`
3. Welche Art von Rekursion haben Sie letztendlich benutzt?

Lösungsvorschlag:

1. *Die Vorlage wurde für generative Rekursion vorgestellt. Da aber die strukturelle Rekursion ein Spezialfall der generativen Rekursion ist, kann die Vorlage auch für strukturelle Rekursion verwendet werden.*

2. *Hier die Lösung:*

```
;; trivially-solvable : (listof X) -> boolean
;; determines if the problem is trivially solvable.
;; In this implementation the problem is
;; trivially solvable if an empty list is passed
;; example: (trivially-solvable? empty) -> true
(define (trivially-solvable? problem)
  (or (not (cons? problem)) (empty? problem)))

;; determine-solution : (listof X) -> 0
;; Returns the solution for a trivially solvable
;; problem. In this implementtion the solution
;; is 0.
;; example: (determine-solution empty) -> 0
(define (determine-solution problem) 0)

;; combine-solutions : (listof X) number -> number
;; combine the solutions of the current problem
;; and the solution to the newly generated simple
;; problem. In this implementation just adds one
;; to the solution of the simpler problem.
;; example: (combine-solutions empty 3) -> 4
(define (combine-solutions problem n)
  (+ 1 n))

;; generate-problem : (listof X) -> (listof X)
;; generate a simpler problem from a difficult problem
;; In this implementation return the rest of a list.
;; Counting the rest of a list is simpler than counting
;; the whole list.
;; example: (generate-problem '(a b)) -> '(b)
(define (generate-problem problem)
  (rest problem))
```

3. *Es handelt sich um strukturelle Rekursion. (Die Antwort 'generative Rekursion' wäre natürlich auch richtig, da die strukturelle Rekursion ein Spezialfall der generativen Rekursion ist. Die Antwort 'strukturelle Rekursion' ist an dieser Stelle aber genauer.) Dies sieht man leicht an der Verwendung von* `empty?` *als Rekursionsanker und der Anwendung von* `rest` *auf das Argument des rekursiven Aufrufs.*

4 Das Newton Verfahren (K)

Mit dem Newton Verfahren lässt sich näherungsweise eine Nullstelle einer Funktion f berechnen. Die Funktion f muss einigen Bedingungen genügen, die hier nicht weiter erörtert werden sollen. Das Newton Verfahren arbeitet rekursiv: Ausgehend von einer Schätzung x_n wird eine bessere Schätzung x_{n+1} wie folgt berechnet:

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}$$

Das Verfahren startet mit einer beliebigen initialen Schätzung x_0. Das Verfahren wird abgebrochen, sobald die Änderung von x_n zu x_{n+1} kleiner einer Schranke δ ist. Die letzte Schätzung x_{n+1} wird dann als Näherungswert für die Nullstelle zurückgegeben. Schreiben Sie die Prozedur `newton-method`, die

- eine Funktion f (`f`)
- deren Ableitung f' (`dfx`)
- einen Startwert x_0 (`x`)
- und eine Schranke δ (`delta`)

erhält und näherungsweise eine Nullstelle der Funktion f mit Hilfe des Newton Verfahrens bestimmt. Geben Sie zunächst Vertrag, Beschreibung und ein Beispiel für die Prozedur an. Implementieren Sie dann die Prozedur.

Lösungsvorschlag:

```
;; Note: Newton's method is not suitable for all functions,
;; as it will not terminate for certain functions.

;; newton-method : (num -> num) (num -> num) num num  ->  num
;; calculates the root of a function by newton's method
;; example: (newton-method
;;         f:        (lambda (x)(* x x))
;;        df:        (lambda (x)(* x 2))
;;        x0:        1
;;     delta:        0.0001)
(define (newton-method f dfx x delta)
  (local (
    (define next-x (- x (/ (f x) (dfx x))))
    (if (< (abs (- next-x x)) delta)
        next-x
        (newton-method f dfx next-x delta))))

(check-within
 (newton-method
  (lambda (x)(* x x))
  (lambda (x)(* x 2))
  1
  0.01)
 0
 0.1)
```

5 Zahldarstellung (K)

Schreiben Sie eine Funktion `convert: num num -> lon`, die eine Zahl x zur Basis b darstellt. Unter der Darstellung einer Zahl x zur Basis b verstehen wir eine Liste von Zahlen $a_{n-1}...a_0$ mit

$$x = \sum_{i=0}^{n-1} a_i \cdot b^i, a_i \in \mathbb{Z}, 0 <= a_i < b$$

$n \in \mathbb{N}$ ist dabei die kleinste Potenz von b, die größer oder gleich x ist, d.h. für die $x \leq b^n$ gilt, n ist also die Anzahl der Stellen von x in der gesuchten Darstellung.
Beispiel: Soll $x = 10$ im Binärsystem $b = 2$ dargestellt werden gilt: n ist gleich 4, da $2^4 > 10$ und $2^3 < 10$. $10 = 1 \cdot 2^3 + 0 \cdot 2^2 + 1 \cdot 2^1 + 0 \cdot 2^0$. Die Liste, die zurückgegeben werden soll ist (a_3, a_2, a_1, a_0), also `'(1 0 1 0)`.

1. Schreiben Sie eine Funktion `lengthRepresentation: num num -> num`, die x und b konsumiert und die Anzahl der Stellen der resultierenden Darstellung (n) zurückgibt. Sie können die Funktion `expt: num num -> num` verwenden. `(expt x y)` liefert x^y.
   Beispiel: `(lengthRepresentation 10 2)` ergibt 4.

2. Schreiben Sie nun die Funktion `convert`. Sie können folgende in Scheme eingebaute Funktionen verwenden:
   - `expt`, s. oben
   - `floor: num -> num`, rundet eine Zahl ab: `(floor 3.7)` ist 3.
   - `remainder: num num -> num`, gibt den Rest der Division der beiden übergebenen Zahlen zurück: `(remainder 10 3)` ist 1.
   - verwenden Sie nicht append sondern nur cons!

Lösungsvorschlag:

```
1  ;; purpose: calculates the minimal power of b
2  ;; that is greater than x, p is the largest
3  ;; power tried so far.
4  ;; contract: maxPower: num num num -> num
5  ;; example: (maxPower 10 2 0) -> 4, as (expt 2 4)  > 10
6  (define (maxPower x b p)
7    (if (> (expt b p) x)
8        p
9        (maxPower x b (+ p 1))))
10
11 (check-expect
12  (maxPower 10 2 0)
13  4)
14
15 (check-expect
16  (maxPower 0 2 0)
17  0)
18
19 ;; purpose: calculates the length of the representation of x
20 ;; in base b
21 ;; contract: lengthRepresentation: num num -> num
22 ;; example: (lengthRepresentation 10 2) -> 4
23 (define (lengthRepresentation x b)
24   (if (= x 0)
25   1
26   (maxPower x b 1)))
27
28 (check-expect
29  (lengthRepresentation 0 2)
30  1)
31
32 (check-expect
33  (lengthRepresentation 4 2)
34  3)
35
36 ;; purpose: convert the number x to the base n
37 ;; convert: num num -> lon
38 ;; exampel: (convert 4 2) -> (1 0 0)
```

```
39 (define (convert x b)
40   (local (
41     (define (remaining-ciphers rest-of-x p)
42       (if (< p 1)
43       empty
44       (cons
45        (floor (/ rest-of-x p))
46        (remaining-ciphers
47         (remainder rest-of-x p)
48         (/ p b))))))
49   (remaining-ciphers
50    x
51    (expt
52     b
53     (- (lengthRepresentation x b) 1)))))
54
55 (check-expect
56  (convert 12 2)
57  (list 1 1 0 0))
58
59 (check-expect
60  (convert 120 16)
61  (list 7 8))
```

Hausübung

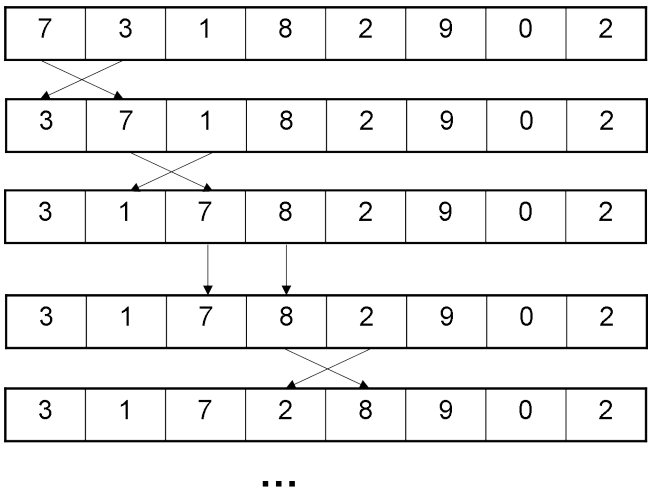
Die Vorlagen für die Bearbeitung werden im GdI1-Portal bereitgestellt. Kommentieren Sie Ihren selbst erstellten Code. Die Hausübung muss bis zum Abgabedatum im GdI1-Portal abgegeben werden. Der Fachbereich Informatik misst der Einhaltung der Grundregeln der wissenschaftlichen Ethik großen Wert bei. Zu diesen gehört auch die strikte Verfolgung von Plagiarismus. Mit der Abgabe Ihrer Hausübung bestätigen Sie, dass Sie bzw. Ihre Gruppe alleiniger Autor des gesamten Materials sind. Falls Ihnen die Verwendung von Fremdmaterial gestattet war, so müssen Sie dessen Quellen deutlich zitieren.

Abgabedatum: Fr, 28.11.08, 16:00 Uhr.

Denken sie daran ihren Code mindestens mit Verträgen und Beschreibungen zu kommentieren, sowie für jede Prozedur 2 Testfälle anzugeben. Wählen sie für Hilfsfunktionen und Parameter sinnvolle Namen. Benutzen sie als Sprachlevel "Zwischenstufe mit Lambda".

6 Bubblesort (6 P)

Ein Sortierverfahren für Listen von Zahlen ist BubbleSort. Beim BubbleSort Verfahren wird eine Liste in mehreren Durchläufen sortiert. Bei einem BubbleSort Durchlauf werden alle in der Liste benachbarten Elemente verglichen. Stehen zwei benachbarte Elemente nicht in der richtigen Reihenfolge, werden Sie vertauscht. Das Bild zeigt ein Beispiel für einen BubbleSort Durchlauf.



Hinweis: Sei $(a_0\, a_1\, a_2 \ldots a_n)$ die zu sortierende Liste. Im ersten Schritt werden a_0 und a_1 betrachtet. Das kleinere der beiden Elemente a_0 und a_1 nennen wir s das andere l. Die Liste sieht nach dem ersten Schritt so aus: $(s\, l\, a_2 \ldots a_n)$. Im nächsten Schritt muss dann l mit a_2 verglichen werden, usw.

1. Überlegen Sie sich ein geeignetes Abbruchkriterium. Wann ist ein BubbleSort Durchlauf für eine Liste trivial und wie sieht das Ergebnis in diesen trivialen Fällen aus? Beanworten Sie diese Frage **in Worten** mit einem Satz!

2. Ergänzen Sie die Funktion `bubblesort-run` aus der Vorlage. Ergänzen Sie auch den Vertrag! `bubblesort-run` konsumiert eine Liste von Zahlen, führt einen BubbleSort-Durchlauf auf dieser Liste durch und gibt das Ergebnis des Durchlauf zurück. Verwenden Sie als Rekursionsanker den von ihnen in der ersten Teilaufgabe gefundenen trivialen Fall. **Beispiel:** `(bubblesort-run '(1 3 2 4 2))` ergibt `'(1 2 3 2 4)`.

Nach jeden Durchlauf BubbleSort steht das größte Element am Ende der Liste, der vordere Teil der Liste ist noch unsortiert. Der BubbleSort Algorithmus führt nun so lange BubbleSort Durchläufe auf dem unsortierten Teil der Liste durch, bis die ganze Liste sortiert ist.

3. Ergänzen Sie die Funktion `bubblesort` aus der Vorlage. Ergänzen Sie auch den Vertrag! `bubblesort` konsumiert eine Liste von Zahlen und gibt sie nach dem BubbleSort Algorithmus sortiert zurück. Sie dürfen die Funktionen `last` und `head` aus der Vorlage verwenden. `last` konsumiert eine Liste und gibt deren letztes Element zurück. `head` konsumiert eine Liste, schneidet das letzte Element ab und gibt die so verkürzte Liste zurück. Beispiel: `(head '(a b c))` ergibt `'(a b)`.

Lösungsvorschlag:
Im falle einer leeren oder einelementigen Liste kann die Liste unverändert zurück gegeben werden.

```
1  ;; Contract: bubblesort-run: list-of-numbers -> list-of-numbers
2  ;; Purpose: takes a list of numbers and switches two adjacent numbers if
       the left one is smaller than the right one
3  ;;          i.e. one iteration of the bubblesort algorithm
4  ;; Example: (bubblesort-run '(4 2 11 1)) -> (2 4 1 11)
5  (define (bubblesort-run alon)
6    (cond
7      ;;trivial: list has less than two elements
8      [(empty? alon) alon]
```

```
9       [(empty? (rest alon)) alon]
10      ;;complex solution
11      [else (local (
12        ;; call the first two elements ai and ai+1
13        (define ai (first alon))
14        (define ai+1 (first (rest alon))) ;; or (second alon)
15        ;; let s be the smaller of a and b
16        (define s (if (< ai  ai+1) ai  ai+1))
17        ;; and let l be the other one
18        (define l (if (< ai  ai+1) ai+1 ai))
19        ;; the new list with reordered elements
20        (define newlist (cons s (cons l (rest (rest alon))))))
21        ;; done with the first element of the list, but recurese with rest
22        (cons (first newlist) (bubblesort-run (rest newlist)))))]))
23
24  (check-expect
25   (bubblesort-run '(4 2 11 1))
26   '(2 4 1 11))
27
28  (check-expect
29   (bubblesort-run '(1 2 1 1))
30   '(1 1 1 2))
31
32
33  ;; Contract: last: list-of-numbers -> number
34  ;; Purpose:  Returns the last element of a list
35  ;; Example:  (last '(4 2 11 1)) -> 1
36  (define (last alon)
37    (first (reverse alon)))
38
39  (check-expect
40   (last '(1 2 3 4))
41   4)
42
43  ;; Purpose:  Takes a list and resturns the same list without the last
      element
44  ;; Contract: head: list-of-numbers -> list-of-numbers
45  ;; Example:  (head '(4 2 11 1)) -> '(4 2 11)
46  (define (head alon)
47    (reverse (rest (reverse alon))))
48
49  (check-expect
50   (head '(1 2 3 4))
51   '(1 2 3))
52
53  ;; Contract: bubblesort: list-of-numbers -> list-of-numbers
54  ;; Purpose:  the main procedure for calling the bubbelsort algorithm
55  ;;           takes a list of numbers and returns the same list in sorted
      order
56  ;; Example:  (4 2 11 1) -> (1 2 4 11)
57  (define (bubblesort alon)
58    (if (empty? alon)
59        ;; trivial: empty list is sorted
60        empty
61        ;: combine solution from simpler problem:
62        ;;          only the unsorted part needs sorting
63        ;; and solution to part problem:
64        ;;          largest element found.
```

```
65        (local (
66         (define unsorted-part (head (bubblesort-run alon)))
67         (define largest-element (last (bubblesort-run alon))))
68         (append (bubblesort unsorted-part) (list largest-element)))))
69
70  (check-expect
71   (bubblesort '(11 7 2 5 12))
72   '(2 5 7 11 12))
73
74  (check-expect
75   (bubblesort '(1 2 1 1))
76   '(1 1 1 2))
```

Bei `bubblesort-run` *und* `bubblesort-run` *handelt es sich um generative Rekursion.*

7 Pythagoras-Baum (7 P)

7.1 Turtle Grafik

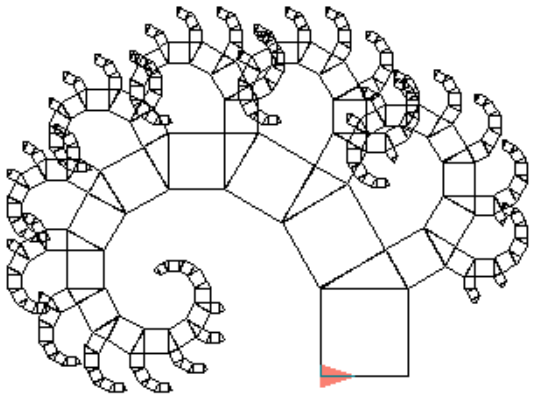
In dieser Übung soll in Scheme mit den `turtle` Zeichenbefehlen gezeichnet werden. Dazu muss das Teachpack `turtle.ss` installiert werden. Hierzu rufen Sie in der Menüleiste die Option "Sprache → Teachpack hinzufügen..." auf. Sollte das Teachpack nicht in der Liste stehen, finden Sie es im DrScheme Installationsverzeichnis unter collects/teachpack/turtle.ss.
Der Befehl `(turtles)` erzeugt eine Turtle-Instanz. Sie können dem Turtle Befehle erteilen. Alle Turtel Befehle werden in DrScheme zu `(void)` ausgewertet haben aber einen Nebeneffekt: Sie zeichnen auf dem Zeichenfenster. Um dem Turtle mehrere Befehle zu erteilen, reicht es diese in eine Liste zu schreiben, `(list (draw 10) (move 5))` wird z.B. zu `(list (void) (void))` ausgewertet, als Nebeneffekt wird aber eine Linie gezeichnet und der Turtle bewegt.
Die folgende Liste zeigt alle Befehle, die notwendig sind, um die Turtle zu bewegen und zu zeichnen:

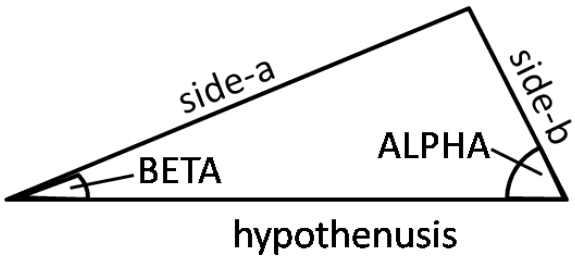
Turtle-Befehl	Semantik
`(draw n)`	Linie der Länge n in aktuelle Richtung zeichnen
`(move n)`	n Schritte in aktuelle Richtung gehen
`(turn a)`	Um a Grad nach links drehen
`(turn -a)`	Um a Grad nach rechts drehen

7.2 Pythagoras-Baum Zeichnen

Nicht nur Funktionen können rekursiv definiert werden, sondern auch Punktemengen. Ein Beispiel dafür ist der rekursiv definierte Pythagoras-Baum. Einen Pythagoras-Baum zeichnet man folgendermaßen:

- Zeichne ein Quadrat mit Seitenlänge a
- Zeichne auf der Oberseite des Quadrats ein rechtwinkliges Dreieck mit Basiswinkeln α und β. Die Länge der Hypothenuse des Dreiecks ist a
- Dies ist der Pythagoras-Baum der Stufe n. Wiederhole diese beiden Schritte auf den beiden Katheten des Dreiecks, a ist dann jeweils die Länge der Katheten. Dies ist der Pythagoras-Baum der Stufe $n + 1$.





1. Schreiben Sie die Funktion `pythagoras-step`, die die Seitenläng a konsumiert und die ersten beiden Schritte beim Zeichnen eines Pythagoras-Baum durchführt. Zum Zeichnen des Dreiecks können Sie die Funktion `paintTriangle` aus der Vorlage verwenden. Diese konsumiert eine Zahl a und zeichnet ein rechtwinkliges Dreieck mit der Hypothenusenlänge a. `paintTriangle` verwendet die ebenfalls in der Vorlage definierten Winkel `ALPHA` und `BETA`. Alle Funktionen in ihrer Abgabe müssen mit Vertrag, Beispiel, Beschreibung und mindestens zwei Testfällen versehen sein!

2. Überlegen Sie sich ein geeignetes Abbruchkriterium, das bestimmt bis zu welcher Stufe ein Pythagorasbaum gezeichnet wird. Beanworten Sie diese Frage **in Worten** mit einem Satz!

3. Schreiben Sie die Funktion `pythagoras-tree` die einen Pythagoras-Baum zeichnet. Verwenden Sie das von Ihnen erdachte Abbruchkriterium. Zur Berechnung der neuen Seitenlänge a können Sie die Funktionen `get-a` und `get-b` verwenden, die die Hypothenusenlänge eines rechtwinkligen Dreiecks konsumieren und jeweils die Länge einer Kathete zurückgeben.

Lösungsvorschlag:

```
1 ;;initializes the turtle-screen
2 (turtles)
3
4 (define ALPHA 30)
5 (define BETA (- 90 ALPHA))
6
7
8 ;; approximations for the length of the cathetus,
9 ;; consumes the length of the hypothenusis
```

```
10 ;; get-b: num -> num
11 ;; get-a: num -> num
12 (define (get-b c)
13   (* c (sin (* (/ ALPHA 180) 3.14))))
14
15 (define (get-a c)
16   (* c (sin (* (/ BETA 180) 3.14))))
17
18 ;; purpose: paints a triangle with the given base length x and given
     angle ALPHA
19 ;; and returns the turtle to its starting position
20 ;; contract: triangle: num -> does not matter
21 (define (paintTriangle a)
22   (list
23    (draw a)
24    (turn (+ 90 ALPHA))
25    (draw (get-b a) )
26    (turn 90)
27    (draw (get-a a))
28    (turn (- 180 ALPHA))))
29
30
31 ;; paints a square of the given width and returns the turtle at its
     starting position
32 ;; aquare num -> does not matter
33 (define (paintSquare a)
34   (list
35    (draw a)
36    (turn 90)
37    (draw a)
38    (turn 90)
39    (draw a)
40    (turn 90)
41    (draw a)
42    (turn 90)))
43
44
45 ;; paints a square and a triangle on top of it with a given base length x
      and with the given angle ALPHA
46 ;; pythagoras-step num -> does not matter
47 (define (pythagoras-step  a)
48   (list
49    (paintSquare a)
50    (turn 90)
51    (move a)
52    (turn -90)
53    (paintTriangle a)
54    (turn -90)
55    (move a)
56    (turn 90)))
57
58 ;; recursively paints a pythagoras tree with given base width x
59 ;; Used abortion criterion: base width is smaller than 5 (as this is no
     more well visible)
60 ;; pythagoras-tree: num -> does not matter
61 (define (pythagoras-tree a)
62   (if (< a 5)
63       empty
```

```
64        (list
65         (pythagoras-step a)
66         (turn 90)
67         (move a)
68         (turn (- ALPHA 90))
69         (pythagoras-tree (get-a a))
70         (move (get-a a))
71         (turn -90)
72         (pythagoras-tree (get-b a))
73         (turn -90)
74         (move (get-a a))
75         (turn (- 90 ALPHA))
76         (move a)
77         (turn 90))))
78
79  ;;TEST
80  (pythagoras-tree 40)
```