Grundlagen der Informatik 1
WS 2008/09

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

Übung 6 - Lösungsvorschlag Version: 1.2 23.11.2008

---

1 Mini Quiz

Die abstrakte Laufzeit eines rekursiven Algorithmus...

- ☒ lässt sich mit einer Kostenfunktion $T(n)$ abschätzen.
- ☐ kann durch mehrere Testeingaben exakt ermittelt werden.
- ☐ kann nie einer Komplexitätsklasse zugeordnet werden.
- ☐ ist der Quotient aus Eingabegröße und der Zahl der Rekursionsschritte.
- ☐ kann (ungefähr) durch die Zeilenanzahl des Codes bestimmen werden.

Komplexität:

- ☒ Die O-Notation ist generell nur für große Eingaben aussagekräftig.
- ☐ Algorithmus $A \in O(n)$ terminiert für alle Eingaben schneller als Algorithmus $B \in O(n^2)$.
- ☐ QuickSort liegt in $O(n\,log(n))$.
- ☒ $n^3 \in O(n^n)$
- ☒ $O(log(n^{1000000})) \subset O(n)$
- ☒ $O(n) = O(0,0123456789n)$
- ☒ $10000000000000000000001 \in O(1)$

Akkumulatoren:

- ☐ Funktionen im Akkumulator-Stil sind immer schneller als Funktionen ohne Akkumulator.
- ☒ Manche Funktionen lassen sich nur mit Akkumulatoren implementieren.
- ☐ Ein Akkumulator wird nur nur bei Problemen, die generative Rekursion nutzen, benötigt.
- ☒ Akkumulator-Stil führt unter Umständen zu einem Effizienzgewinn.

2 Fragen

1. Welche Schritte sind nötig, um eine Funktion im Akkumulator-Stil zu implementieren?
2. Was versteht man unter der Akkumulator-Invarianten?

3. Zeigen oder widerlegen Sie folgende Behauptung: Man kann einen Sortieralgorithmus, dessen Mechanismus auf dem Vergleich zweier Zahlen beruht, in der Komplexitätsklasse $O(log(n))$ implementieren. **Hinweis**: Dieses Problem soll Ihnen zeigen, an welche Grenzen Algorithmen manchmal stossen. Den Beweis für die Aufgabe können Sie informell (also: mathematisch, zeichnerisch oder mit eigenen Worten) beschreiben. Stellen Sie sich eine Liste von Zahlen vor, die sortiert werden müssen. Wie gross ist der Aufwand, wenn jede Zahl in der Liste wenigstens einmal "angefasst" werden muss?

4. Angenommen, wir haben eine Liste mit 30 identischen Elementen. Wie verhält sich die Laufzeit von QuickSort in diesem Fall? Wäre es ratsamer, hier ausnahmsweise einen Algorithmus mit schlechterer Komplexität zu wählen?

5. Fassen Sie bitte in Ihren eigenen Worten, wofür $\Theta(..)$, $\Omega(..)$ und $O(..)$ stehen.

6. Welche der folgenden Aussagen gelten?
   a) $O(3\,n\,log_4(n)) \subseteq \Theta(9\,n\,log_{10}(n))$
   b) $O(3\,n\,log_4(n)) \subseteq O(9\,n\,log_{10}(n))$
   c) $\Theta(3\,n\,log_4(n)) \subseteq O(9\,n\,log_{10}(n))$?

7. Zeigen Sie, dass aus $f(n) \in O(log_a(n))$ folgt $f(n) \in O(log_b(n))$ für beliebige a, b.

Lösungsvorschlag:

1. *Definition und Bereitstellung des Akkumulators (welches Wissen über die Parameter muss im Akkumulator gespeichert werden?).*

2. *Eine Akkumulator-Invariante beschreibt die Beziehung zwischen dem eigentlichen Argument der Funktion, dem aktuellen Argument der Hilfsfunktion und dem Akkumulator.*

3. *Es ist definitiv* **NICHT** *möglich ein Sortieralgorithmus, der vergleichsbasiert arbeitet, in weniger als $O(nlog(n))$ zu implementieren.*

   Ein Widerspruch dazu kann in einem Satz (informell) erklärt werden: Sie müssen ja jedes zu sortierende Element mindestens einmal anfassen, um es richtig einzuordnen. Das allein ist schon in $\Omega(n)$. Mit anderen Worten die untereste Schranke ist schon bereits in **n** *und bekanntlich ist $n > log(n)$ für alle $n > 0$.*

4. *Tatsächlich würde QuickSort die Liste 30 mal unterteilen die Laufzeit wäre also $O(n^2)$. InsertionSort benötigt nur einen Durchlauf durch die Liste, die Laufzeit wäre also $O(n)$.*

5. - *Ω ist die* **untere** *Schranke,*
   - *O ist die* **obere** *Schranke,*
   - *Θ ist widerum die* **genaue** *Schranke*

   zur Abschätzung der Komplexität eines Algorithmus.

6. *Welche der folgenden Aussagen gelten?*
   a) $O(3\,n\,log_4(n)) \not\subseteq \Theta(9\,n\,log_{10}(n))$
   b) $O(3\,n\,log_4(n)) \subseteq O(9\,n\,log_{10}(n))$
   c) $\Theta(3\,n\,log_4(n)) \subseteq O(9\,n\,log_{10}(n))$?

7.

$$\begin{aligned} f(n) \in O(log_a(n)) &\Rightarrow \exists n_0, C. \forall n > n_0. f(n) < C log_a(n) \\ &\Rightarrow \exists n_0, C. \forall n > n_0. f(n) < C \frac{log_b(n)}{log_b(a)} \\ &\Rightarrow \exists n_0, C. \forall n > n_0. f(n) < \frac{C}{log_b(a)} log_b(n) \\ \Rightarrow (C' = \frac{C}{log_b(a)}) &\exists n_0, C'. \forall n > n_0. f(n) < C' log_b(n) \\ &\Rightarrow f(n) \in O(log_b(n)) \end{aligned}$$

3 Rekursion vs. Iteration: Fakultät (K)

Die Fakultät von n (geschrieben $n!$) ist wie folgt definiert:

$$n! = \begin{cases} 1 & falls\, n = 0 \\ n \cdot (n-1)! & sonst \end{cases}$$

1. Implementieren Sie die Funktion `fak-rec`, die eine Zahl n konsumiert und $n!$ zurückliefert. Verwenden Sie Rekursion! Markieren Sie die Stelle in Ihrer Lösung, in der die Rekursion terminiert (Rekursionsanker) und die Stelle an der die Funktion sich selbst rekursiv aufruft.
   Lösungsvorschlag:

```
1  ;; contract: fak-rec: number->number
2  ;; purpose: calculate the factorial of the given number
3  ;; example: (fak-rec 3) should be 6
4  (define (fak-rec n)
5    (if (= n 0)
6        1 ;; Rekursionsanker
7        (* n (fak-rec (- n 1))))) ;; rekursiver Aufruf von fak-rec
```

2. Veranschaulichen Sie nun Ihre Lösung (genauso wie in der Folie T7.7) indem Sie als Parameter für die Fakultät die Zahl 6 nehmen.

   Lösungsvorschlag:

```
1  (fak-rec 6)
2  (* 6 (fak-rec 5))
3  (* 6 (* 5 (fak-rec 4)))
4  (* 6 (* 5 (* 4 (fak-rec 3))))
5  (* 6 (* 5 (* 4 (* 3 (fak-rec 2)))))
6  (* 6 (* 5 (* 4 (* 3 (* 2 (fak-rec 1))))))
7  (* 6 (* 5 (* 4 (* 3 (* 2 1)))))
8  (* 6 (* 5 (* 4 (* 3 2))))
9  (* 6 (* 5 (* 4 6)))
10 (* 6 (* 5 24))
11 (* 6 120)
12 ;; Ergebniss: 720
```

3. Implementieren Sie nun die iterative Variante `fak-iter` der Fakultät unter Verwendung eines Akkumulators. Der Akkumulator soll das bisher aufmultiplizierte Produkt enthalten. Initialisieren Sie den Akkumulatorwert mit 1, multiplizieren Sie mit n, das Ergebnis multiplizieren Sie mit $n-1$, das neue Ergebnis mit $n-2$, solange bis Sie bei $n-n+1=1$ angelangt sind.

   Lösungsvorschlag:

```
;; contract: fak-iter-akk: number number -> number
;; purpose: calculate the factorial of n using an accumulator akk
;; example: (fak-iter-akk 3 1) should be 6
(define (fak-iter-akk n akk)
  (if (= n 1)
      akk
      (fak-iter-akk (- n 1) (* n akk))))

;; contract: fak-iter: number -> number
;; purpose: calculate the factorial of n
;; example: (fak-iter 3) should be 6
(define (fak-iter n)
  (fak-iter-akk n 1))
```

4. Veranschaulichen Sie Ihre Lösung wieder mit der Zahl 6.

   Lösungsvorschlag:

```
;; Der Aufruf: (fak-iter 6) würde folgenden Rechenprozess erzeugen:
(fak-iter 6)
(fak-iter-akk 6 1)
(fak-iter-akk 5 6)
(fak-iter-akk 4 30)
(fak-iter-akk 3 120)
(fak-iter-akk 2 360)
(fak-iter-akk 1 720)
;; Ergebniss: 720
```

5. Vergleichen Sie beide Varianten hinsichtlich der Länge des Codes, der Komplexität der Berechnung und der Anzahl der im Speicher zu haltenenden Elemente pro Rechenschritt.

   Lösungsvorschlag:

   Hinweis: *Die Definition der Prozeduren ist in beiden Fällen rekursiv aber der durch sie entstehende Prozess nicht immer.*

   Die rekursive Variante benötigt weniger Zeilen Code. Sie ist von der Berechnungskomplexität (Anzahl der benötigten Multiplikationen etc) äquivalent zur iterativen Variante (beide haben eine Laufzeit von O(n)). Die rekursive Variante muss aber mehr Elemente im Speicher halten als die iterative Variante.

   Merken Sie sich einen interessanten Grundsatz für rekursive Funktionen: Viele Probleme lassen sich rekursiv sehr elegant (mit wenig Codezeilen) lösen, bezahlen dafür aber mit einer hohe Komplexität, Abhilfe hierfür sind iterative Prozeduren, diese jedoch sind meist komplexer in der Implementierung als ihre rekursiven Pendants.

Eine ausführliche Darstellung dieses Sachverhalts finden Sie in Abschnitt 1.2 des Buches: **Structure and Interpretation of Computer Programs**, *das Sie aus der Vorlesung bzw. GDI1 Portal (zumindest vom Namen her) kennen sollten.*

4 Rekursion vs. Iteration: Fibonacci (K)

In dieser Aufgabe sollen Sie, wie in Aufgabe 3, die Fibonacci Funktion erst rekursiv, dann iterativ implementieren. Die Fibonacci Funktion ist wie folgt definiert:

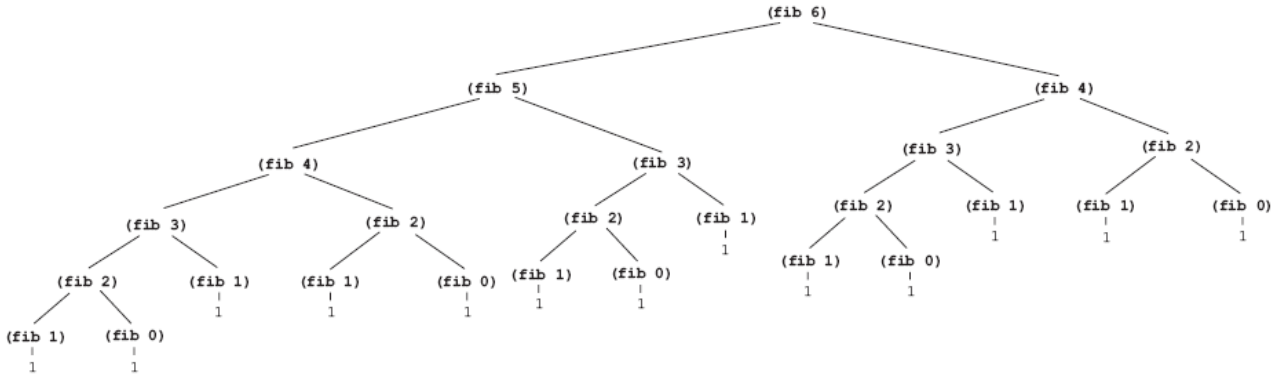
$$fib(n) = \begin{cases} fib(n-1) + fib(n-2) & n \geq 2 \\ 1 & sonst \end{cases}$$

1. Implementieren Sie die Funktion `fib-rec: number -> number`, die die n-te Fibonaccizahl rekursiv berechnet. Zeichnen Sie einen Rekursionsbaum für die 6. Fibonacci Zahl und beurteilen Sie die Effizienz Ihrer Lösung. Zur Erinnerung, die ersten acht Fibonacci Zahlen lauten (beginnend mit der 0ten): 1, 1, 2, 3, 5, 8, 13 und 21.

   Lösungsvorschlag:

```
;; contract: fib-rec: number -> number
;; purpose: calculate the n-th fibonacci number
;; example: (fib-rec 4) => 5
(define (fib-rec n)
   (if (or (= n 0) (= n 1))
       1
       (+ (fib-rec (- n 1)) (fib-rec (- n 2)))))

;; (fib-rec 6) ergibt somit: 13
```



Das Problem der rekursiven Berechnung von Fibonaccizahlen ist, dass Teile der Berechnung doppelt ausgeführt werden. In den oberen Grafik ist das z.B. (fib-rec 2), bei größeren Fibonaccizahlen wiederholt sich dieser Effekt entsprechend. Dies lässt sich durch iterative Berechnung vermeiden.

2. Implementieren Sie nun die Fibonacci-Funktion iterativ und beurteilen Sie einen Vorteil der iterativen Lösung gegenüber der rekursiven Lösung. Verwenden Sie Akkumulatoren in Ihrer Lösung.

Lösungsvorschlag:

```
;; contract: fib-iter: number -> number
;; purpose: calculate the nth fibonacci number
;; example: (fib-iter 3) should be 3
(define (fib-iter n)
  (local (
      ;; contract: fib-iter-akk: number number number -> number
      ;; purpose: calculate the fibonacci number using the counter i
        and the two last fibonacci numbers fibi-2 and fibi-1
    (define (fib-iter-akk i fibi-2 fibi-1)
      (if (<= i 1)
        fibi-1
        (fib-iter-akk (- i 1) fibi-1 (+ fibi-2 fibi-1)))))

  (fib-iter-akk n 1 1)))
```

Betrachten Sie nun folgende Aufrufe:

```
(fib 6)
(fib-iter-akk 6 1 1)
(fib-iter-akk 5 1 2)
(fib-iter-akk 4 2 3)
(fib-iter-akk 3 3 5)
(fib-iter-akk 2 5 8)
(fib-iter-akk 1 8 13)
13
```

Bei der iterativen Berechnung müssen **keine** *Berechnungen doppelt ausgeführt werden, weil deren Ergebnisse bereits vorliegen, wenn sie benötigt werden. Da stets nur die Werte der zwei vorherigen Fibonaccizahlen zur Berechnung der nächsten Fibonaccizahl benötigt werden, reicht es aus, die letzten zwei Fibonaccizahlen als Parameter zu übergeben. Sie können somit in der folgenden Berechnung wiederverwendet werden und müssen nicht neu berechnet werden.*
Genau so geht fib-iterative vor. Die letzten zwei Parameter enthalten die vorherigen zwei Fibonaccizahlen. Der erste Parameter zählt, wie viele Schritte noch durchgeführt werden müssen. Initialisiert wird der letzte Parameter mit 1 so dass bei n = 1 oder n = 0 korrekt 1 zurückgegeben wird.

Noch cleverer (O(log(n))) kann man generative Rekursion verwenden:

$$fib(2 * n) = fib(n) * fib(n) + fib(n + 1) * fib(n + 1)$$

$$fib(2 * n - 1) = (2 * fib(n + 1) - fib(n)) * fib(n)$$

5 Akkumulatoren

Lesen Sie in T8.24 über die Funktion `invert` mit Akkumulator.

1. Schreiben Sie jetzt eine Funktion `invert2`, die *ohne* Akkumulator auskommt. Vervollständigen Sie dazu folgenden Code, ohne `append` oder eine ähnliche bereits verfügbare Funktion zu verwenden:

```
;; invert2 : (listof X) -> (listof X)
;; construct the reverse of list lst
;; example: (invert2 (list 'A 'B 'C)) should return (list 'C 'B 'A)
(define (invert2 lst)
```

```
5      ;; rcons : (listof X) X -> (listof X)
6      ;; appends el to the end of list lst.
7      ;; example: (rcons (list 'A 'B) 'C) should be (list 'A 'B 'C)
8     (local (
9       (define (rcons lst el)
10                (...) ))
11      (...) ))
```

2. Vergleichen Sie Ihre Implementierung `invert2` mit `invert`. Welcher Algorithmus liegt in welcher Komplexitätsklasse?

Lösungsvorschlag:

```
1  ;; invert2 : (listof X) -> (listof X)
2  ;; construct the reverse of list lst
3  ;; example: (invert2 (list 'A 'B 'C)) should return (list 'C 'B 'A)
4  (define (invert2 lst)
5      ;; rcons : listofx X -> listofx
6      ;; appends el to the end of list lst.
7      ;; example: (rcons (list 'A 'B) 'C) should be (list 'A 'B 'C)
8      (local
9          ((define (rcons lst el)
10              (if (empty? lst)
11                  (list el)
12                  (cons (first lst) (rcons (rest lst) el)))))
13        (if (empty? lst)
14           empty
15           (rcons (invert2 (rest lst)) (first lst)))))
16
17 (invert2 (list 'A 'B 'C 'D))
```

Die Funktion `invert` aus der Vorlesung durchläuft die gesamte Liste genau ein Mal und ist daher in $O(n)$. Die neue Funktion `invert2` ohne Akkumulator durchläuft ebenfalls die gesamte Liste ein Mal. Allerdings muss in jedem Schritt *ein Element an eine Hilfsliste angehängt werden. Um ein Element an eine Liste anhängen zu können, muss diese ebenfalls einmal durchlaufen werden. Hier gibt es also eine Komplexität von $O(n^2)$. Man sieht, dass diese Implementierung mit Akkumulatoren eine bessere (geringere) Komplexität hat.*

6 Rekursion und Komplexität

Die Ackermannfunktion ist eine 1926 von Wilhelm Ackermann gefundene mathematische Funktion, die in der theoretischen Informatik eine wichtige Rolle bezüglich Komplexitätsgrenzen von Algorithmen spielt. Sie wird als Benchmark zur Überprüfung rekursiver Prozeduraufrufe verwendet, wenn man die Leistungsfähigkeit von Programmiersprachen oder Compilern überprüfen will. Implementieren Sie die vereinfachte Ackermann Funktion nach Péter rekursiv anhand folgender Definition:

$$\begin{aligned} Ack(0,m) &= m+1 \\ Ack(n,0) &= Ack(n-1,1) \\ Ack(n,m) &= Ack(n-1,Ack(n,m-1)) \end{aligned}$$

Berechnen Sie dann den Aufruf: Ack(3,4). Berechnen Sie nun die ersten Schritte des Aufrufs Ack(4,2). Was fällt Ihnen hierbei sehr schnell auf? Ist diese Funktion bezüglich der Komplexität von praktischem Nutzen?

Lösungsvorschlag:

```
1 ;; contract: Ack: number number -> number
2 ;; purpose: calculates the Ackermann number
3 ;; example: Ack(2,3) should be 9
4 (define (Ack n m)
5   (cond
6   [(= n 0) (+ m 1)]
7   [(= m 0) (Ack (- n 1) 1)]
8   [else (Ack (- n 1) (Ack n (- m 1)))]))
```

Folgende Tabelle veranschaulicht den Aufruf von Ack(3,4) sowie Ack(4,2). Es lässt sich ebenfalls hier beobachten, wie schnell diese scheinbar triviale Funktion wächst...

n,m	0	1	2	3	4	5	6	7	8	9
0	1	2	3	4	5	6	7	8	9	10
1	2	3	4	5	6	7	8	9	10	11
2	3	5	7	9	11	13	15	17	19	21
3	5	13	29	61	**125**	253	509	1021	2045	4093
4	13	65533	**Ack(4, 2)**	...	...	...	...	...	...	...
5	65533	...	...	...	...	...	...	...	...	...

Der Aufruf von Ack(4,2) erzeugt eine gigantische Zahl (insgesammt 19729 Stellen), von der hier nur ein kleiner Bruchteil gezeigt ist...

Ack(4, 2) = 2003529930406846464979072351560255750447825475569751419265016973710894059556311453089506130880933348101038234342907263181822949382118812668869506364761547029165041871916351587966347219442930927982084309104855990570159318959639524863372367203002916969592156108764948889254090805911457037675208500206671563 7...

Hinsichtlich des praktischen Nutzen können wir also folgendes festhalten:

- *Sie ist sinnvoll zum Testen der Effizienz von Prozeduraufrufmechanismen.*
- *Sie ist sinnvoll zur Beschreibung der Komplexitat mancher Algorithmen.*
- *Sonst keinen weiteren praktischen Nutzen !*

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, 05.12.08, 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”.

7 Zum Aufwärmen: Palindrome (K) (4 P)

Implementieren Sie eine Funktion `palindrome`, die eine Liste konsumiert und sie zu einem Palindrom erweitert. Geben Sie mindestens 2 Testfälle an. Ein Palindrom ist eine Liste, die von vorn und von hinten gelesen gleich bleibt. Der Vertrag, Beschreibung und Beispiele sehen wie folgt aus:

```
;; Contract: palindrome: (listof x) -> (listof x)
;; Purpose:  to build a list which transforms the given
;;           list into a palindrome list..
;; Examples 1.) (palindrome (list 'a 'b 'c))
;;                  should produce: (list 'a 'b 'c 'b 'a)
;;          2.) (palindrome (list 1 2 3 4))
;;                  should produce: (list 1 2 3 4 3 2 1)
```

Lösungsvorschlag:

```
;; Contract: palindrome: (listof x) -> (listof x)
;; Purpose:  to build a list which transforms the given
;;           list into a palindrome list..
;; Examples 1.) (palindrome (list 'a 'b 'c))
;;                  should produce: (list 'a 'b 'c 'b 'a)
;;          2.) (palindrome (list 1 2 3 4))
;;                  should produce: (list 1 2 3 4 3 2 1)
(define (palindrome alox)
   (if (empty? alox)
       empty
       (append alox (rest (reverse alox)))))
;;statt reverse kann auch invert oder invert2 verwendet werden.
;;rest kann auch weggelassen werden, dann ergeben sich auch
;;Palindrome, in denen das mittlere Element doppelt vorkommt.
(check-expect
 (palindrome (list 'a 'b 'c 'd))
 (list 'a 'b 'c 'd 'c 'b 'a))

(check-expect
 (palindrome (list 1 2 3 4))
 (list 1 2 3 4 3 2 1))
```

Tips für Tutoren:
1P für Rekursionsanker empty
0.5P pro Testfall
2P else Zweig: (1P) append, (1P) reverse

8 Sinus-Approximation (K) (4P)

Im Folgenden soll eine Approximation der Sinus-Funktion $sin(x)$ implementiert werden. x soll dabei im Bogenmaß angegeben werden. Für kleine x kann man $sin(x)$ durch x abschätzen. Als "klein" sollen im Rahmen dieser Aufgabe Werte von $|x| \leq 0.1$ angesehen werden. Der Sinus für größere Werte lässt sich dann nach folgender Gleichung abschätzen:

$$sin(x) = \begin{cases} x & |x| \leq 0.1 \\ 3sin(\frac{x}{3}) - 4(sin(\frac{x}{3}))^3 & sonst \end{cases}$$

1. Implementieren Sie die Abschätzungsfunktion für den Sinus in Scheme. Vermeiden Sie, dass $sin(\frac{x}{3})$ mehrfach berechnet wird, indem Sie die lambda-Notation verwenden. Vergessen Sie nicht den Vertrag anzugeben, Beschreibung, Beispiel und Testfälle können Sie vom Übungsblatt übernehmen.

```
;; Purpose: calculate sinus x using the approximation for small
  numbers
;; Example: (sin-approx 3.14) should be close to 0

(define (sin-approx angle)
  ...)
;;Test
(sin-approx 1.2)
;;should be
0.93218229417886703406711722434709627405353802620137...

;;Test
(sin-approx 3.14)
;;should be
0.00080567746742247618824 65...
```

Lösungsvorschlag:

```
;; Contract: sin-approx: number -> number
;; Purpose: calculate sinus x using the approximation for small
  numbers
;; Example: (sin-approx 3.14) should be close to 0
(define (sin-approx angle)
    (if (<= (abs angle) 0.1)
        angle
        ((lambda (x) (- (* 3 x ) (* 4 (* x x x ))))
          (sin-approx (/ angle 3)))))
```

2. Wie oft ruft sich Ihre Funktion rekursiv selbst auf, wenn Sie den Sinus von 12.15 berechnen? Begründen Sie Ihre Antwort!

Lösungsvorschlag:
Es sind fünf Aufrufe. In jedem Aufruf wird `angle` *durch drei dividiert, nach fünf Aufrufen ist* `angle` *kleiner 0.1.*

3. Da sin periodisch ist, gilt $sin(x) = sin(x \ mod \ 2\Pi)$, $x \ mod \ 2\Pi$ ist dabei der Rest, der bei der Teilung durch 2Π entsteht. Jedes x kann so im ersten Schritt auf das Intervall $[0, 2\Pi]$ abgebildet werden, das größte x für das $sin(x)$ also rekursiv berechnet werden muss, ist $x = 2\Pi$. Was ist daher die Komplexität des Algorithmus?

Lösungsvorschlag:
Der Algorithmus terminiert also nach spätestens 4 rekursiven Aufrufen, d.h. die Anzahl der maximal benötigten Aufrufe ist konstant und somit ist der Algorithmus in $O(1)$.

Tips für Tutoren:
Vertrag: 0.5P

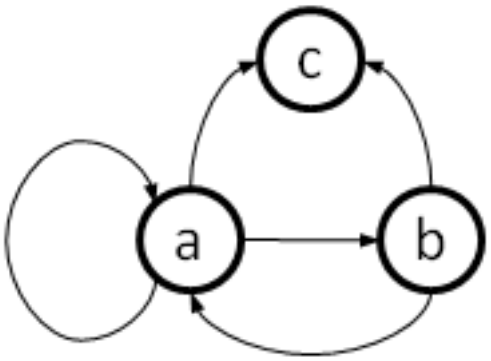
Korrekte Implementierung: 1P (auch ohne abs) mit Lambda + 1P

Korrekte Anzahl der Aufrufe: 0.5P

Korrekte Komplexität: 1P

9 Graphen (5 Punkte)

In der Vorlesung haben Sie bereits verschiedene Graphen kennengelernt (T6,T8). Im Folgenden möchten wir nun auf gerichteten Graphen arbeiten. Wir repräsentieren solche Graphen als `listof edge`. Anders als in der Vorlesung soll eine Kante (`edge`) **eine Struktur** aus Startknoten und Endknoten sein. Startknoten und Endknoten sollen wie in der Vorlesung Symbole sein. In der Vorlage ist die Struktur und ein kleiner Graph `samplegraph` definiert, der foldenden Graphen repräsentiert.



Beachten Sie: In dieser Übung dürfen Graphen Zyklen enthalten! Ein Zyklus ist ein Pfad in einem Graph, der beim gleichen Knoten startet und endet. Kette von mit Kanten verbundenen Knoten, die in sich geschlossen sind, z.B. die Kette a, b oder die einelementige Kette a im Beispielgraph.

- Schreiben Sie eine Prozedur `neighbors: node graph -> (listof node)`, die einen Knoten n und einen Graphen G konsumiert und die Nachbarknoten dieses Knotens als Liste zurückliefert. Nachbarknoten sind diejenigen Knoten in G die Endknoten zu einer Kante in G sind bei der n Startknoten ist. Zum Beispiel sind a,b und c Nachbarknoten von a, a und c von b und c hat keine Nachbarknoten. Beachten Sie, dass Kanten Strukturen vom Typ `edge` sind!
- Wir wollen nun für unsere Graphen eine Prozedur `find-route: node node graph -> (listof node)` erstellen, die zwei Knoten und einen Graphen konsumiert, und einen beliebigen zyklenfreien Pfad vom ersten zum zweiten Knoten ausgibt (vgl. T6.53ff). Falls kein Pfad existiert soll `false` zurückgegeben werden. Stellen Sie dabei mit Hilfe von Akkumulatoren sicher, dass die Prozedur auch auf Graphen, **die Zyklen enthalten**, terminiert! Sie dürfen die Funktion `contains?` aus der Vorlage verwenden. `contains?` gibt für ein Symbol s und eine Liste von Symbolen los zurück, ob s in los enthalten ist.

Vergessen Sie nicht Vertrag, Beschreibung, Beispiel und mindestens zwei Tests anzugeben.

Lösungsvorschlag:

```
1 ;;struct for storing (directed) graph edges
2 (define-struct edge (start end))
3
```

```
(define samplegraph
  (list
   (make-edge 'a 'b)
   (make-edge 'b 'a)
   (make-edge 'a 'a)
   (make-edge 'a 'c)
   (make-edge 'b 'c)))


;;helper function contains?
;;returns whether sym is contained in a list of symbols
;;contract: contains: symbol list-of-symbols -> boolean
::example: (contains-symbol? 'a (1 2 3 4)) -> false
::example: (contains? 'a '(a b c d)) -> true
(define (contains? sym alos)
  (foldl (lambda (x c) (or (symbol=? sym x) c)) false alos))



(define (neighbors n G)
  (local (
    ;; outgoing-edge?: edge -> boolean
    ;; returns whether this edge starts in n
    (define (outgoing-edge? e)
     (symbol=? (edge-start e) n))
    ;; add-end-node: edge list-of-nodes -> list-of-nodes
    ;; adds the end of an edge to a list-of-nodes
    (define (add-end-node e list-of-nodes)
      (cons (edge-end e) list-of-nodes))
    ;; add-if-neighbor: edge list-of-nodes -> list-of-nodes
    ;; adds the end of an edge to a list-of-nodes
    ;; if the edge starts in n
   (define (add-if-neighbor e neighbors)
      (if (outgoing-edge? e)
          (add-end-node e neighbors)
          neighbors)))
  ;;add all end-nodes of edges in G that start in n
  (foldl add-if-neighbor empty G)))

;(neighbors 'a samplegraph)
;(neighbors 'b samplegraph)
;(neighbors 'c samplegraph)



(define (find-route orig dest graph)
  (find-route-accu orig dest graph empty))

;; Contract: find-route-accu: node node graph (listof node) -> (listof
  node)
;; Extended version of find-route in slide T6.61
;; new: visited is a list of symbols representing the nodes that have
  already been visited
(define (find-route-accu orig dest graph visited)
  (cond
    ;;trivial case: origin == destination
    [(symbol=? orig dest) (list orig)]
```

```
60      ;;trivial case: origin has already been visited
61      ;;this route contains a cycle!
62      [(contains? orig visited) false]
63      [else
64       ;;complex case: get a possible route
65       ;;starting from ANY neighbor. Mark
66       ;;the current node as visited
67       (local ((define possible-route
68                 (find-route/list
69                  ;;start at any neighbor
70                  (neighbors orig graph)
71                  ;;find route to same destination
72                  dest
73                  ;;in the same graph
74                  graph
75                  ;;this node is now visited!
76                  (cons orig visited))))
77         (cond
78           ;;no route found, return false
79           [(boolean? possible-route) false]
80           ;;okay! Prepend origin to route
81           [else (cons orig possible-route)])))]))
82
83  ;; Contract: find-route/list: (listof node) node node (listof node) -> (
      listof node) or false
84  ;; Extended version of find-route/list in slide T6.63
85  ;; new: visited contains the nodes which have been inserted in find-route
      -accu
86  (define (find-route/list lo-Os dest graph visited)
87    (cond
88      ;;trivial: no origins -> no route
89      [(empty? lo-Os) false]
90      [else
91       (local
92           ;;try to find a route from the first origin
93           ((define possible-route (find-route-accu (first lo-Os) dest
               graph visited)))
94         (cond
95           ;;this is not a route? "Turn around" and try with the rest of
               origins
96           [(boolean? possible-route)
97            (find-route/list (rest lo-Os) dest graph visited)]
98           ;;this is a route, so return
99           [else possible-route]))]))
100
101 (check-expected
102   (find-route 'a 'b samplegraph)
103   '(a b))
```

Tips für Tutoren:

Korrekte Implementierung von neighbors: 2P Korrekte Implementierung von find-route: 3P Für fehlenden Vertrag, Beschreibung, Beispiel, Tests kann pro Prozedur 0.5P. abgezogen werden