Alle heute gebräuchlichen Computer-Sprachen fallen unter den Oberbegriff problemorientierte Sprachen, womit eine weitgehende Unabhängigkeit gekennzeichnet wird von der Maschine auf der die Software läuft.
Bei den problemorientierte Sprachen unterscheidet man:

- *Prozedurale bzw. imperative Sprachen*
  Bei diesen Sprachen stehen Prozeduren und Anweisungen im Mittelpunkt, wobei Prozeduren zur Zusammenfassung mehrerer Anweisungen dienen. Zu dieser Sprachgruppe gehören u.a. die Sprachen Fortran, Pascal, Basic und C.
- *Objektorientierte Sprachen*
  Smalltalk war die erste reine objektorientierte Sprache (OO-Sprache). Die meisten heute üblichen Sprachen sind Mischformen, die sowohl die objektorientierte, als auch die imperative Programmierung erlauben. Bei den OO-Sprachen stehen Klassen, Objekte und die Vererbung im Vordergrund. Zu diesen Sprachen gehören (neben Smalltalk) Java, Delphi, C++, Eiffel, Python, Ruby und auch Visual-Basic.
- *Deklarative Sprachen*
  Bekannt ist aus diesem Bereich eigentlich nur PROLOG (Programming Logic), ein System, das hauptsächlich im Bereich der künstlichen Intelligenz eingesetzt wird. Expertensysteme und wissensbasierte Systeme werden gern in Prolog realisiert.
- *Funktionale Sprachen*
  Der bekannteste Vertreter dieser Gruppe ist LISP (List Processing), eine Sprache, bei der jedes Programm eine Funktion darstellt. Lisp hat einen sehr streng logischen Aufbau und als zentralen Datentyp die Liste. Alle anderen Datentypen leiten sich vom Typ Liste ab. Auch das Programm selber lässt sich als Liste auffassen und modifizieren. Neben Lisp spielen vor allem Lisp-Dialekte wie Scheme eine Rolle.

1. DrScheme

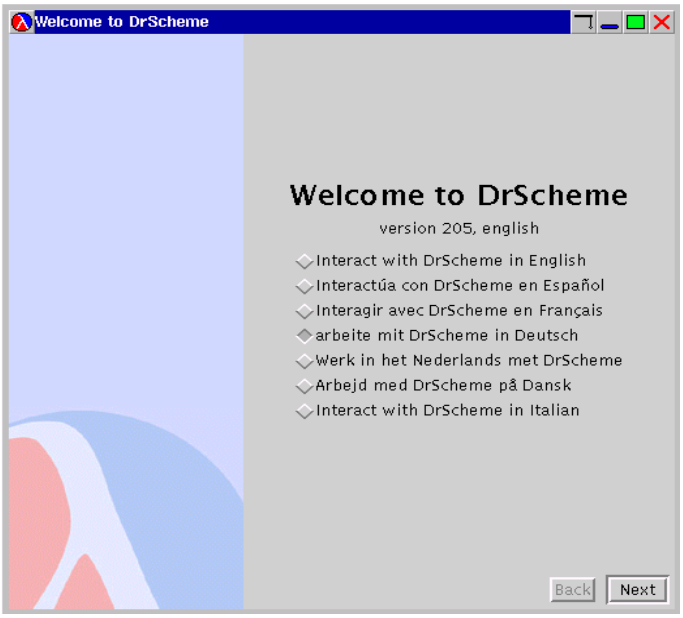
Die Programmiersprache Scheme ist ein spezieller Dialekt von LISP (List Processing) und damit eine funktionale Sprache. Wie der Name LISP schon andeutet, sind beide Sprachen auf die Verarbeitung von Listen spezialisiert.

DrScheme wiederum ist eine Implementierung von Scheme, die man kostenlos unter *http://www.drscheme.org* beziehen kann. Es gibt dort Versionen für Unix, MacOS und verschiedene Windows-Versionen.

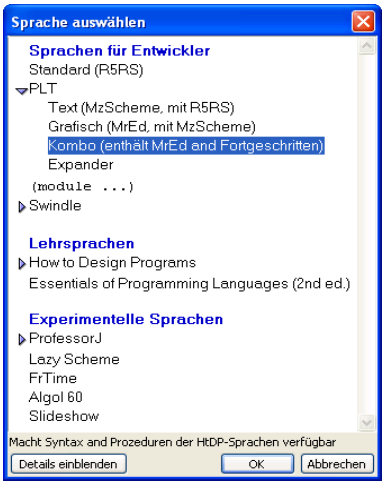
1.1. Umgang mit DrScheme

Bei manchen Versionen ist bei ersten Start von DrScheme die Sprache auszuwählen. Es stehen hier mehrere Sprachen zur Auswahl, u.a. auch Deutsch. Nach der Änderung der Sprache muss man DrScheme leider neu starten.

Die Spracheinstellung kann bei allen Versionen unter *Hilfe* geändert werden, danach ist dann aber ein Neustart des Programms notwendig..

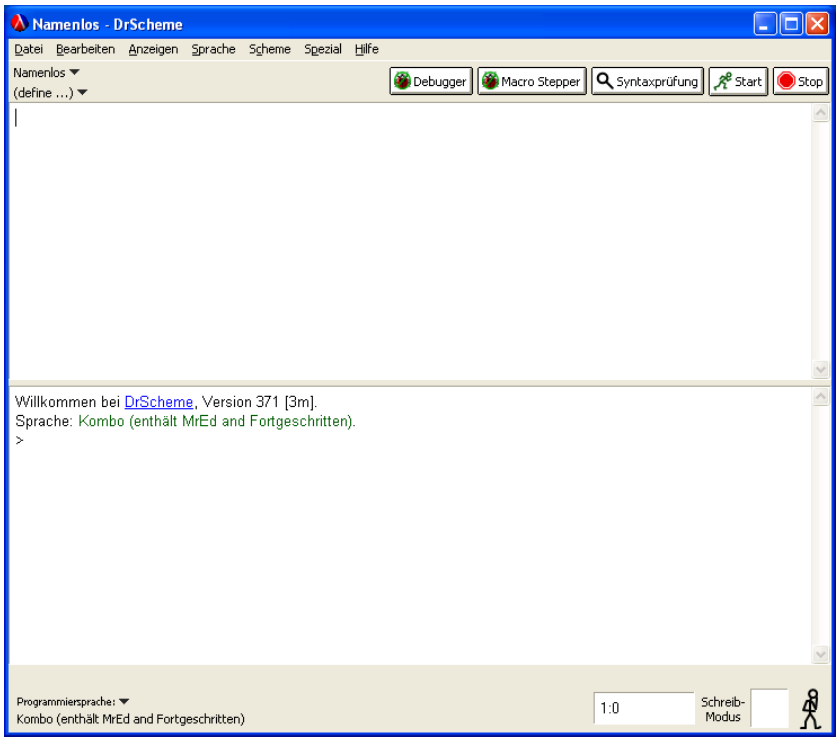


Wichtig ist die Wahl des Scheme-Sprachumfanges, der zur Verfügung stehen soll.
Es ist am sinnvollsten hier gleich den vollen Sprachumfang zu aktivieren, also die Einstellung *Kombo* ....
Bei den anderen Einstellungen steht nur ein eingeschränkter Sprachumfang zur Verfügung, mit dem die folgenden Beispiele nicht alle funktionieren werden.



Der Sprachumfang kann jederzeit über den Menüpunkt *Sprache · Sprache Auswählen...* nachträglich verändert werden.

DrScheme startet mit einem zweigeteilten Fenster:
Im oberen Teil des Fensters, dem Editor stehen die Texte, die gesichert werden sollen. Hier wird man also Programmtexte schreiben. Eventuelle Eingabe führt DrScheme nur dann aus, wenn der Button *Ausführen* aktiviert wird.



Das untere Fenster wird als REPL (read-eval-print loop) bezeichnet, alle Eingaben hier werden direkt ausgeführt.

DrScheme meutert, wenn Eingaben im REPL-Fenster erfolgen und im Editor-Fenster Änderungen erfolgt sind, ohne dass Ausführen betätigt wurde. Das hängt damit zusammen, dass die Funktionen aus dem Editor ja auch im REPL zur Verfügung stehen sollen.

1.2. Erste Beispiele mit Scheme

Die ersten Schritte mit Scheme erfolgen im REPL-Fenster. Scheme soll den Ausdruck 3+5 berechnen. Das ist hier in einer ungewöhnlichen Notation geregelt, bei der der Operator (hier „+“) vorne steht und dann die Operanden folgen. In Scheme schreibt man also:

```
(+ 3 5)
```

Nach dem Betätigen der Enter-Taste erhält man das Ergebnis 8.
Einen Ausdruck wie 3+(5*8) stellt man in Scheme als

```
(+ 3 (* 5 8))
```

dar. Scheme benötigt hier keine Vorrangregeln, die Reihenfolge der Berechnung ergibt sich aus den Klammern.
Schon an diesen ersten Beispielen sieht man, das Scheme alles als Liste betrachtet. Ungewöhnlich sind auch Schreibweisen wie:

```
(+ 3  4  5)
```

für 3+4+5.
Entsprechend ist für 1/5 die Kurz-Eingabe `(/ 5)` möglich.

Interessant ist das Ergebnis von `(/ 2 3)`, Scheme liefert hier `2/3` als Ergebnis. Das hängt damit zusammen, dass DrScheme normalerweise exakt rechnet und damit keine Näherungszahl liefert. Benötigt man die Näherungszahl, hier also 0,66666, so muss man Scheme zur Umwandlung des Ergebnisses in eine Näherungszahl anweisen:

```
(exact->inexact (/ 2 3))
```

Jede höhere Programmiersprache kennt Bezeichner, so auch Scheme. Der einfachste Mechanismus zur Benennung von Objekten besteht in der Verwendung von *define*:

```
(define x 42)
```

Danach steht der Bezeichner zur Verfügung und wird beim Aufruf durch den Wert des Ausdruckes ersetzt, der ihm zugeordnet wurde. Als Ergebnis von `(/ x 2)` erhält man also `21`.
Die Bezeichner können auch sprechend sein `(define mehrwertsteuer 16)` und der Ausdruck darf wiederum eine Berechnung enthalten
`(define zwischenergebnis (+ 17 4))`.

Einige Standard-Bezeichner sind schon vordefiniert, so `pi` (Kreiszahl) und `e` (Eulersche Zahl)

Scheme kennt nicht nur Zahlen als Werte für seine Bezeichner, sondern auch Texte. Bei Textsymbolen muss man aber deutlich machen, dass sie nicht als Bezeichner zu interpretieren sind, deshalb muss man sie quoten. Entweder mit `(define x (quote Hallo))` oder etwas kürzer `(define x 'Hallo)`. In beiden Fällen ergibt der Aufruf von `x` den Wert `hallo`,
Scheme unterscheidet nicht zwischen Groß- und Kleinschreibung.

2. Scheme-Sprachelemente

In diesem ersten Abschnitt über Scheme geht es primär um die Möglichkeiten im Zusammenhang mit numerischen Daten.
Das wohl wichtigste Sprach-Element von Scheme sind die Funktionen.

2.1. Funktionen

Funktionen in Scheme bekommen üblicherweise einen Namen, eine Parameterliste und den eigentlichen Prozedurkörper. Ein einfaches Beispiel ist eine Funktion zum Quadrieren einer Zahl.

```
(define (quadrat x)
    (* x x))
```

Aufrufen kann man die so definierte Funktion anschließend z.B. mit `(quadrat 5)`. Selbst definierte Funktionen haben den gleichen Stellenwert wie die vordefinierten.
Vor allem lassen sich Funktionen auch übersichtlich schachteln. Das folgende Beispiel dient zur Berechnung des Volumens eines Zylinders. Wenn der Radius des Grundkreises und die Höhe gegeben sind, dann gilt hier V=Fläche des Grundkreises * Höhe. Wenn man das schrittweise angeht, dann ergibt sich folgendes Programm:

```
(define (quadrat x)
  (* x x))

(define (kreisfläche radius)
  (* pi (quadrat radius)))

(define (zylinder-volumen radius höhe)
  (* (kreisfläche radius) höhe))
```

wobei die Reihenfolge der Funktionsdefinitionen hier keine Rolle spielt. Erst beim Aufruf der Hauptfunktion müssen die anderen Funktionen auch vorhanden sein.
Hinweis: Die folgenden Definitionen sollten auf alle Fälle im Editor-Fenster vorgenommen wer-

den, da dies Veränderungen erleichtert und die Listings auch einfach gespeichert werden können.

2.2. Fallunterscheidungen

Jede Programmiersprache kennt die Möglichkeit, verschiedene Fälle zu unterscheiden, so auch Scheme. Die einfachste Möglichkeit der Fallunterscheidung erfolgt mit if, die Nutzung zeigt das folgende Beispiel:

```
(define (betrag zahl)
     (if (>= zahl 0)
     zahl
     (- zahl)))
```

Hinter dem `if` folgt eine Bedingung, ein Test, hier `(>= zahl 0)`. Danach folgt dann der Ausdruck, der ausgewertet wird, wenn die Bedingung erfüllt ist, hier einfach `zahl`. Zuletzt folgt dann der Ausdruck, hier `(- zahl)`, der ausgewertet wird, wenn der Test `false` liefert.

Für Unterscheidungen mit mehr als zwei Fällen bzw. unterschiedlichen Bedingungen dient `cond`.

```
(define (betrag2 zahl)
  (cond ( (< 0 zahl)  zahl)
        ( (= 0 zahl) 0)
        ( else (- zahl))))
```

Hier können beliebig weitere Bedingungen (*Prädikate*) überprüft werden, wie z.B. hier die zusätzliche Abfrage auf 0. Für das letzte Prädikat kann man auch das Schlüsselwort `else` angeben, die Bedingung tritt immer dann ein, wenn keine der bisherigen zutraf.

Für den Fall, dass man die zu unterscheidenden Werte aufzählen kann kennt Scheme noch die sehr übersichtlich Möglichkeit mit `case`.

```
(define (zahlwort zahl)
  (case zahl
    ((1) 'Eins)
    ((2) 'Zwei)
    ((3) 'Drei)
    ((4 5 6) "mehr als drei")
    (else "ich kann nur bis drei Zählen")))
```

Die Funktion *zahlwort* erwartet eine Zahl als Parameter und gibt das zugehörige Zahlwort aus, zumindest bis zur drei. Das erste Element für jeden der Fälle ist eine Liste mit den zugehörigen Werten, danach folgt dann der Ausdruck, der als Ergebnis genommen wird.

2.3. Prädikate I

Bei den Fallunterscheidungen taucht ein spezieller Typ von Funktionen auf, die Prädikate. Prädikate sind Funktionen, die nicht Zahlen zurückliefern, sondern Wahrheitswerte, also `#t` (*true*) oder `#f` (*false*). Daten dieses Typs werden auch als *boolesche* Daten bezeichnet. Scheme kennt u.a. die folgenden numerischen Prädikate:

Prädikat	***Argument(e)***	***Erläuterung***
<	2 Zahlen	#t, wenn die erste Zahl kleiner ist
>	2 Zahlen	#t, wenn die erste Zahl größer ist
=	2 Zahlen	#t, wenn beide Zahlen gleich sind
<=	2 Zahlen	#t, wenn die erste Zahl kleiner oder gleich ist
>=	2 Zahlen	#t, wenn die erste Zahl größer oder gleich ist
zero?	1 Zahl	#t, wenn die Zahl gleich 0 ist
integer?	1 Zahl	#t, wenn die Zahl ganz ist

Prädikat	*Argument(e)*	*Erläuterung*
real?	1 Zahl	#t, wenn die Zahl zu den reellen Zahlen gehört
rational?	1 Zahl	#t, für eine rationale Zahl
complex?	1 Zahl	#t, für eine komplexe Zahl
number?	1 Zahl	#t, für jede Zahl
odd?	1 Zahl	#t, wenn die Zahl ungerade ist
even?	1 Zahl	#t, wenn die Zahl gerade ist
negative?	1 Zahl	#t, wenn die Zahl negativ ist
positive?	1 Zahl	#t, wenn die Zahl positiv ist
zero?	1 Zahl	#t, wenn die Zahl Null ist
exact?	1 Zahl	#t, wenn die Zahl exakt dargestellt ist, #f z.B. bei pi
inexact?	1 Zahl	#t, wenn die Darstellung genähert ist

Neben diesen vordefinierten Prädikaten kann man jederzeit eigene Prädikate definieren. Im Unterschied zu normalen Funktionen muss man hier ein Fragezeichen an den Bezeichner anhängen. Das folgende Prädikat untersucht, ob die übergebene Zahl eine Quadratzahl ist oder nicht.

```
(define (quadratzahl? zahl)
  (integer? (sqrt zahl)))
```

Gerade dieses Beispiel wäre mit vielen anderen Programmiersprachen kaum realisierbar, da aufgrund der genäherten Zahlen-Darstellung kaum feststellbar wäre, ob eine Zahl ganz ist oder nicht, vor allem, wenn es sich dann auch noch um die Wurzel einer anderen Zahl handelt. Da Scheme exakt rechnet, ist dies realisierbar.

Auch das folgende Beispiel, das auf Teilbarkeit untersucht, ist nur aufgrund der exakten Zahlendarstellung so einfach realisierbar.

```
(define (teiler? zahl1 zahl2)
  (integer? (/ zahl2 zahl1)))
```

2.4. Numerische Funktionen in Scheme

Auch eine Vielzahl von Funktionen, die keine Prädikate sind, ist zur Arbeit mit Zahlen in DrScheme bereits vorhanden.

Funktion	*Argument(e)*	*Erläuterung*
round	1 Zahl	rundet mathematisch (round 4.6) ergibt 5
truncate	1 Zahl	schneidet ab, (truncate 4.6) ergibt 4
remainder	2 Zahlen	Rest beim Teilen (remainder 7 3) ergibt 1
modulo	2 Zahlen	entspricht modulo
exact->inexact	1 Zahl	wandelt zur genäherten Darstellung
inexact->exact	1 Zahl	wandelt in die naheliegendste exakte Zahl
ceiling	1 Zahl	kleinste ganze Zahl, die nicht kleiner ist
floor	1 Zahl	größte ganze Zahl, die nicht größer ist
exp	1 Zahl	Potenz zur Basis e
log	1 Zahl	Logarithmus zur Basis e
sin, cos, tan, atan	1 Zahl	trigonometrische Funktionen
sqrt	1 Zahl	Wurzelfunktion
string->number	String [Zahl]	liefert den zugehörigen Zahlenwert, die optionale Zahl gibt das Stellenwertsystem an
number->string	1 Zahl [Zahl]	wandelt die Zahl in einen String um

2.5. Wiederholungen und Rekursion

Nahezu alle Programmiersprachen kennen Schleifenkonstrukte, so auch Scheme.

```
(define (zähle grenze)
  (do ((i 0 (+ i 1)))
      ((> i grenze) "fertig")
    (display i)
    (newline)))
```

ein kleines Beispiel, welches einfach die Ausgabe hochzählt.

Scheme erlaubt sogar mehrere Schleifenvariablen. Im folgenden Beispiel zur Berechnung der Fakultät tauchen i und prod auf. Der eigentliche Schleifenkörper bleibt leer, sämtliche Berechnungen erfolgen in den Schrittanwisungen.

```
(define (fakultät zahl)
  (do ((i 1 (+ i 1))
    (prod 1 (* prod i)))
    ((> i zahl) prod)))
```

Vermutlich sehr elegant gelöst, aber nicht ganz leicht nachvollziehbar.

Etwas übersichtlicher ist eventuell die folgende Darstellung:

```
(define (fakultät zahl)
  (define prod 1)
  (do ((i 1 (+ i 1)))
    ((> i zahl) prod)
    (set! prod (* i prod))))
```

oder

```
(define (fac zahl)
  (let schleife
    ((i 1)
     (prod 1))
    (if (> i zahl) prod
       (schleife (add1 i) (* i prod)))))
```

So oder ähnlich würde man auch in imperativen Sprachen dieses Problem lösen. In Scheme ist dieser Ansatz eher ungewöhnlich, die Iteration ist hier eher verpönt.
Naheliegend für Scheme ist der rekursive Ansatz.

In den beiden bisherigen Beispielen haben wir die folgende Definition von Fakultät benutzt:
`n! = 1*2*...*n`
Es gibt eine weitere Möglichkeit die Fakultät zu definieren und zwar:
`n! = n*(n-1)! für n>1 und n!=1 für n=1`
Dies lässt sich direkt in ein Scheme-Programm umsetzen.

```
(define (fakultät zahl)
  (if (> zahl 1)
```

```
        (* zahl (fakultät (- zahl 1)))
        1))
```

3. Paare und Listen

Ein grundlegendes Element in Scheme sind Listen und deren Sondertyp, die Paare. Die Funktion zum Aufbau von Paaren heißt *cons*, was eine Abkürzung für *construct* (Erzeugnis) sein soll. Die Funktion verknüpft zwei Objekte zu einem *Paar* (dotted pair). Den grundsätzlichen Umgang zeigt das folgende Beispiel:

```
(define paar1 (cons 3 5))
```

Definiert die Variable paar1 als Paar von zwei Zahlen. Die Abfrage von paar1 liefert das Paar:

```
paar1
  (3 . 5)
```

Man kann auch auf die einzelnen Komponenten von paar1 zugreifen, mit *car* auf die erste Komponente und mit *cdr* auf die zweite.

```
(car paar1)
  3
(cdr paar1)
  5
```

Elemente eines Paares dürfen nicht nur Atome sein (z.B. Zahlen), sondern auch zusammengesetzte Objekte, wie z.B. Paare.

```
(define paar2 (cons paar1 7))
paar2
  ((3 . 5) . 7)
```

Auch auf die Komponenten dieses Doppelpaares kann man ganz normal zugreifen.

```
(car paar2)
  (3 . 5)
(cdr paar2)
  7
```

Die Zugriffe lassen sich natürlich auch kaskadieren, da (car paar2) wieder ein Paar ist.

```
(car (car paar2))
  3
```

Für das Kaskadieren gibt es auch eine Kurzschreibweise, die sich beliebig fortsetzen läßt.

```
(caar paar2)
  3
```

Für die Arbeit mit Paaren gibt es auch neue Prädikate:
`pair?` und `null?`

Häufiger als Paare benötigt man Listen, die aus mehr als zwei Elementen bestehen.

```
(define liste1 (list 1 2 3 5 7))
```

und die Abfrage:

```
liste1
  (1 2 3 5 7)
```

Auch hier stehen car und cdr zur Verfügung, wobei car das erste Element und cdr den Rest der Liste ohne das erste Element liefert.

```
(car liste1)
  1
(cdr liste1)
  (2 3 5 7)
```

Zusätzlich gibt es ein Prädikat, um zu untersuchen, ob eingegebenes Objekt eine Liste darstellt:
`liste?`
Für den Umgang mit Listen und Ausdrücken kann das folgende Beispiel ganz hilfreich sein.

```
(define a 1)
```

```
(define b 2)
(define c 3)
(define d 5)
(define liste1 (list a b c d))
(define liste2 '(a b c d))
```

Im Fall von liste1 wertet Scheme die Symbole aus, im Fall von liste2 nicht, hier wird gequotet. Die gequotete Darstellung ist eine Kurzform von

```
(define liste2 (list 'a 'b 'c 'd))
```

Die Listen haben dann folgende Werte:

```
liste1
  (1 2 3 5)
liste2
  (a b c d)
```

Listen kann man mittels *cons* verlängern. Dazu muss man aber wissen, dass der erste Parameter von *cons* ein beliebiges Objekt sein kann, der zweite aber eine Liste sein muss.

```
(cons 7 liste1)
```

liefert eine korrekte Liste

```
(7 1 2 3 5)
```

während

```
(cons liste1 7)
```

eine *improper-list* liefert

```
((1 2 3 5) . 7)
```

die auch keine Liste ist (Testen mit list?).

3.1. Rekursion mit Listen

Im Zusammenhang mit Listen lässt sich sehr schön rekursiv arbeiten. Für die Arbeit mit Listen hilfreich wäre z.B. eine Funktion, die die Länge einer gegebenen Liste ermittelt. Die Länge einer Liste ergibt sich aus der Länge von (cdr liste), indem man 1 dazu zählt. Das ergibt folgendes Scheme Programm:

```
(define (länge liste)
  (cond ((null? liste) 0)
        (else (+ 1 (länge (cdr liste))))))
```

Aufgabe: Weitere Funktionen mit Listen, die rekursiv programmierbar sind

- Verketten zweier Listen zu einer Liste
- Test ob ein gegebenes Objekt in einer Liste vorhanden ist
- entfernen eines gegebenen Objektes aus einer Liste, falls vorhanden
- umdrehen der Reihenfolge einer Liste
- ein Objekt in der Liste durch ein neues ersetzen (ersetzen)
- das erste Element einer Liste liefern (head)
- das letzte Element einer Liste liefern (tail)
- das i-te Element einer Liste liefern

Lösungen:

```
(define (verketten liste1 liste2)
  (cond ((null? liste1) liste2)
        (else
         (cons (car liste1)
               (verketten (cdr liste1) liste2)))))


(define (enthalten? objekt liste)
  (cond  ((null? liste) #f)
         ((equal? (car liste) objekt) #t)
         (else (enthalten? objekt (cdr liste)))))


(define (entfernen objekt liste)
  (cond ((null? liste) liste)
        ((equal? objekt (car liste)) (entfernen objekt (cdr liste)))
        (else (cons (car liste) (entfernen objekt (cdr liste))))))
```

3.2 Funktionen mit Listen

Die meisten der im vorherigen Abschnitt selbst erstellten Funktionen sind in Scheme schon vorhanden, können also direkt benutzt werden.

`(length liste)` liefert die Anzahl der Elemente in einer Liste.

Man kann Listen auch nachträglich verlängern, mittels *append*, aber alle Parameter müssen wieder Listen sein:

```
(define liste3 (append liste2 (list 7)))
```

ergibt dann

```
liste3
(a b c d 7)
```

Die Reihenfolge der Elemente in einer Liste lässt sich mit `(reverse liste)` vertauschen.

Eine Teilliste lässt sich mit `(list-tail liste position)` abfragen, hier tauchen alle Elemente nach der angegebenen Position auf.
Will man die Restliste nicht ab einer vorgegebenen Position, sondern ab einem vorgegebenen Element haben, so dient dazu `(member objekt liste)`.

Ein einzelnes Element der Liste lässt sich mit `(list-ref liste position)` abfragen.

Falls die Liste aus Paaren besteht, dann kann man mittels `(assoc objekt liste)` das erste Paar erhalten, dessen car-Feld objekt entspricht.

```
(define liste '((1 hund) (muku 5) (2 17) (muku 'ne!)))
(assoc 'muku liste)
(muku 5)
```

4. Weitere Scheme-Funktionen

In diesem Abschnitt folgen nun weitere Informationen über Scheme, die für die Arbeit mit Automaten bzw. Compilern nützlich sein können.

4.1. Zufall

Gelegentlich, benötigt man Zahlen, die in gewisser Weise zufällig sind. Scheme kennt dazu die Funktion random, die natürliche Zahlen liefert und als Parameter eine Zahl erwartet, die größer ist als die größte zulässige Zufallszahl.
`(random 6)` liefert eine Zufallszahl zwischen 0<=Zahl<6, immer nur eine zur Zeit.
Aufbauend auf random liefert die folgende kleine Funktion eine Liste von Zufallszahlen.

```
(define (zufall grenze anzahl)
  (define liste ())
  (do ((i 1 (+ i 1)))
       ((> i anzahl) liste)
    (set! liste (cons (random grenze) liste))
    ))
```

Der Aufruf

```
(zufall 6 40)
```

liefert dann eine Liste von 40 Zufallszahlen aus dem Bereich 0..5.
Scheme kennt die Möglichkeit mehrere Zufallzahlengeneratoren unabhängig voneinander zu nutzen (siehe Hilfe-Funktion).

4.2. Funktionen mit variabler Parameterzahl

Viele der vordefinierten Funktionen verfügen über eine variable Zahl an Parametern. Die arithmetischen Funktionen z.B. erlauben neben dem Operator eine variable Zahl an Operanden, z.B. (/ 5) (/ 1 5) (/ 1 5 6)... Diese Möglichkeit besteht auch für eigene Funktionen.

```
(define (beispiel a b . c)
    <Funktionsrumpf>)
```

definiert eine Funktion beispiel, die zwei oder mehr Argumente benötigt. Das erste Argument wird zu a, das zweite zu b und alle übrigen Argumente gehen an die Liste c. Das folgende Beispiel zeigt die Funktionsweise

```
(define (zeige-argumente a b . c)
  (display "a: ")
  (display a)
  (newline)
  (display "b: ")
  (display b)
  (newline)
  (display "c: ")
  (display c))
```

Der Aufruf `(zeige-argumente 1 2 3 4 5)` liefert dann die Ausgabe

```
a: 1
b: 2
c: (3 4 5)
```

Eine Funktion mit keinem oder mehreren Parametern hätte dann den folgenden Aufbau.

```
(define (beispiel . a)
    <Funktionsrumpf>)
```

Im Zweifelsfall finden sich also alle Parameter in der Liste.

Als Beispiel für die variable Parameterzahl soll eine rekursive Funktion zur Ermittlung einer Liste mit Zufallszahlen dienen:

```
(define (zh anzahl bereich)
  (cond ((= 0 anzahl) empty)
        (else (cons (random bereich) (zh (- anzahl 1) bereich)))))

(define (zufall . parameter)
  (cond ((null? parameter) (zh 10 6))
        ((= 1 (length parameter)) (zh (car parameter) 6))
        ((= 2 (length parameter)) (zh (car parameter) (cadr parameter)))
        (else "Fehler: Falsche Zahl an Parametern")))
```

Dieses Listing arbeitet der Übersichtlichkeit halber mit einer Hilfsfunktion *zh*, die immer mit Zwei Parametern aufgerufen wird.
Ruft man (zufall) ohne Parameter auf, so „denkt“ sich die Funktion die Parameter 10 und 6 dazu, ruft man (zufall 10) mit einem Parameter auf, so „denkt“ sich die Funktion 6 dazu. Bei mehr als zwei Parametern erfolgt eine Fehlermeldung.

Die beiden Funktionen kann man auch miteinander verbinden, zu einer etwas kompakteren Lösung.

```
(define (zufall . parameter)
  (cond ((null? parameter) (zufall 10 6))
        ((= 1 (length parameter)) (zufall (car parameter) 6))
        ((= 2 (length parameter))
              (cond ((= 0 (car parameter)) empty)
              (else (cons (random (cadr parameter)) (zufall (- (car
              ➥ parameter) 1) (cadr parameter))))))
        (else "Fehler: Falsche Zahl an Parametern")))
```

(Hinweis: das Zeichen ➥ deutet einen Zeilenumbruch nur für das Layout des Textes an)

4.3. Funktionen mit Zustand

Für die Arbeit mit Automaten wäre es praktisch, wenn sich der Automat seinen eigenen Zustand merken könnte. Das wäre im Sinne der Objektorientierung sicherer, als mit globalen Variablen zu arbeiten.
Scheme kennt natürlich auch Funktionen mit Zustand. Vor der Beschreibung muss man aber leider auf eine Vereinfachung verzichten, die wir bisher immer für Funktionen benutzt haben.
(define (bezeichner parameter)
  <Funktionsrupf>)
ist nur eine abkürzende Schreibweise für:
(define bezeichner
 (lambda (parameter)
  <Funktionsrumpf>))

Zum Vergleich die Quadratfunktion in beiden Versionen:

```
(define (quadrat1 x)
    (* x x))

(define quadrat2
  (lambda (x)
    (* x x)))
```

Die lambda-Ausdrücke werden oft benutzt um Hilfsfunktionen ohne eigenen Funktionsnamen einsetzen zu können. Im vorliegenden Beispiel wird dann eine solche anonyme Funktion an den Funktionsnamen gebunden.

Bringt man nun zwischen die ersten beiden Zeilen eine Variablendefinition, die den Funktionsrumpf umfasst, so kann die Funktion sich ihren Zustand „merken“.

```
(define summe
  (let ((zustand 0))
    (lambda (differenz)
      (set! zustand (+ zustand differenz))
      zustand)))
```

Die Bindung der 0 an den Bezeichner zustand erfolgt nur beim allerersten Aufruf der Funktion. Bei allen weiteren Aufrufen bleibt der bisherige Zustand erhalten und dient als Ausgangsbasis für die Veränderungen. Zur Veranschaulichung dienen die folgenden fortlaufenden Aufrufe:

```
(summe 50)
50
(summe 5)
55
(summe -3)
52
```

Deutlich wird hier auch die Datenkapselung, ich komme an die Variable innerhalb der Funktion von außen nicht heran, nur über die vordefinierten Mechanismen.

Gelegentlich braucht man mehrere Instanzen eines derartigen Objektes, auch das ist mit Scheme realisierbar. Wir müssen nur den Zustand als Parameter betrachten:

```
(define summe-objekt
  (lambda (zustand)
    (lambda (differenz)
      (set! zustand (+ zustand differenz))
      zustand)))
```

Nun ist der Zugriff etwas anders zu gestalten.

```
(define w1 (summe-objekt 50))
(define w2 (summe-objekt 5))
(w1 7)
57
(w2 7)
12
```

Damit stehen zwei vollkommen unabhängige Zähler-Objekte zur Verfügung, die aber mit der gleichen Methode arbeiten.

Etwas knapper wird die Darstellung übrigens, wenn man auf das doppelte Lambda verzichtet und die abkürzende Schreibweise benutzt.

```
(define (summe-objekt zustand)
    (lambda (differenz)
      (set! zustand (+ zustand differenz))
      zustand))
```