Funktionale und Objektorientierte Programmierung —

Dieses Skript richtet sich nach der Vorlesung von

Prof. Dr. rer. nat. Karsten Weihe

Formulierungen sind teils von den Folien übernommen.

Technische Universität Darmstadt

10. November 2017

Inhaltsverzeichnis

1 Grundlagen der Programmierung

Was ist Progrmamieren?

Schauen wir zunächsteinmal, wie einige der „großen Köpfe“ der Informatik das Programmieren definieren.

> „To program is to understand“
> *Kristen Nygaard*

> „Programming is a Good Medium for Expressing Poorly Understood and Sloppily Formulated Ideas“
> *Marvin Minsky, Gerald J. Sussman*

1.1 Strukturierungsmechanismen

1.1.1 ... einer Programmiersprache

Eine Programmiersprache ist mehr als ein Hilfsmittel um einen Computer anzuweisen, Aufgaben durchzuführen. Sie dient auch als **Rahmen**, innerhalb dessen wir **unsere Ideen** über die **Problemdomäne organisieren.**

Wenn wir eine Sprache beschreiben, sollten wir die Hilfsmittel beachten, die sie uns zum Kombinieren von einfachen Ideen anbietet, um komplexere Ideen zu bilden.

Jede vollwertige Programmiersprache hat drei Mechanismen, um Prozessideen zu strukturieren:

- ***Primitive* Ausrücke**

  - Repräsentieren die einfachsten Einheiten der Sprache

  - Im Deutschen: jedes Wort ist ein primitiver Ausdruck

- ***Kombinationsmittel***

  - Zusammengesetzte Elemente werden aus einfacheren Einheiten konstruiert

  - Im Deutschen: Zusammensetzung mehrerer Wörter zu einem Satz.

- ***Abstraktionsmittel***

  - Zusammengesetzte Elemente können benannt und weiter als Einheiten manipuliert werden

  - Im Deutschen: Definition eines Begriffs („Ein Auto ist ...“), so dass der Begriff danach als „Kurzform“ für die Erklärung nutzbar ist

1.1.2 ... in der Elektronik

- ***Primitive* Ausrücke**

  - Widerstände, Kondensatoren, Induktivitäten, Spannungsquellen, ...

- ***Kombinationsmittel***

  - Richtlinien für das Verdrahten der Schaltkreise

  - Standardschnittstellen (z.B. Spannungen, Strömungen) zwischen den Elementen. Diese Schnittstellen können auch Anforderungen an konkrete zulässige Werte oder Einheiten stellen („5 mA“)

- ***Abstraktionsmittel***

  - “Black box” Abstraktion – denke über einen Unter-Schaltkreis als eine Einheit: z.B. Verstärker, Regler, Empfänger, Sender, ...

1.2 Sprachelemente

1.2.1 Die Primitiven

Zahlen

Zahlen sind selbstauswertend: Die Werte der Zifferfolge ist die Zahl, die, die sie bezeichnen.

$$23 \Rightarrow 23$$

$$-36 \Rightarrow -36$$

Boolesche Werte

Boolsche Werte können nur *wahr* oder *falsch* sein. Diese sind ebenfalls selbstauswertend. Sie werden als

$$True \text{ oder } False$$

bezeichnet. Prozeduren sind in der Programmierung auch als „Funktionen“ oder „Methoden“ bekannt. Beispiele für eingebaute Prozeduren sind

$$+,\ *,\ /,\ -,\ =,\ usw.$$

Aber was ist der Wert von so einem Ausdruck? Der Wert von + ist eine Prozedur, die Zahlen addiert. Dies werden wir später als „Higher-Order Procedures“ kennen lernen. Auswertung: Nachschlagen des dem Namen zugewiesenen Wertes.

1.2.2 Besonderheiten bei Zahlen

DrRacket rechnet immer genau, wenn das möglich ist. Ganze und endliche Zahlen berechnet er so wie es „üblich ist“. Brüche mit periodischem Ergebnis werden ebenfalls in

einem Text als Bruch - etwa `7/3` dargestellt. Im Programm werden sie jedoch als Periode angezeigt. Beispielsweise `- 2.3`. Doch kann nicht jede Zahl *exakt* dargestellt werden. Man wird hier durch die Verwendung des Binärsystems eingeschränkt. Es gilt:

> „je mehr Nachkommastellen die Zahl besitzt, desto ungenauer wird in der Regel ihre Darstellung im Binärsystem“

Zahlen unendlicher Länge ohne Periode werden wie folgt dargestellt: `#i „inexact“`
Hier einige Beispiele:

$$e : \boxed{\text{\#i2.718281828459045}}$$

$$pi : \boxed{\text{\#i3.141592653589793}}$$

$$\sqrt{2} : \boxed{\text{\#i1.4142135623730951}}$$

Mit Brüchen oder „inexakten“ Zahlen kann normal gerechnet werden – das Ergebnis ist aber ggf. wieder „inexact“

$$(\sqrt{2})^2 = \boxed{\text{\#i2.0000000000000004}}$$

1.2.3 Kombination

Der Wert einer Kombination wird bestimmt durch die Ausführung der (durch den Operator) angegebenen Prozedur mit den Werten der Operanden. In Racket ist eine Sequenz von Ausdrücken eingeschlossen in Klammern. Die Ausdrücke sind primitiv oder erneut zusammengesetzt.

Hier ein Beispiel: Numerische Ausdrücke können mit Ausdrücken kombiniert werden, die primitive Prozeduren repräsentieren (z.B. + oder *), um einen zusammengesetzten Ausdruck zu erstellen. Kombinationen können verschachtelt werden. Dafür müssen sie die Regeln einfach Rekursiv anwenden.

```
1 (+ 4 (* 2 3)) = (4 + (2 * 3)) = 10
2 (* (+ 3 4) (- 8 2))
3 = ((3 + 4) * (8 - 2))
4 = 42
```

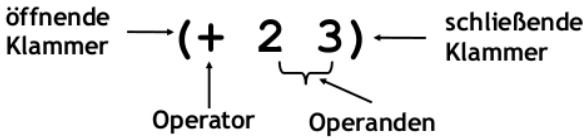


Abbildung 1:

WICHTIG: Eine Kombination bedeutet immer eine Anwendung einer Prozedur. Klammern können nicht eingefügt oder weggelassen werden, ohne die Bedeutung des Ausdrucks zu ändern.

1.2.4 Abstraktion

1.3 Formales

2 Strukturierte Datentypen

3 Rekursive Datentypen und Strukturelle Rekursion

3.1 Listen

Mit Strukturen können Datenobjekte mit einer festen Zahl von Daten gespeichert werden. Häufig wissen wir jedoch nicht, aus wie vielen Datenelementen eine Datenstruktur besteht. Oder die Struktur der Daten ist rekursiv. Mit rekursiven Datentypen können auch beliebig große Datenobjekte strukturiert abgespeichert werden. Die Idee davon ist die Folgende: Ein Element der Datenstruktur speichert (direkt oder indirekt) ein Exemplar der Datenstruktur. Dies nennt man dann eine *rekursive Datenstruktur*. Um eine endliche Datenstruktur zu bekommen benötigt man einen *Rekursionsanker*. Diesen Rekursionsanker modellieren wir mit der Technik zu heterogenen Daten aus dem letzten Kapitel.

Eine Liste ist entweder die leere Liste *the-emtpylst*, oder *(make-lst s r)*, wobei s ein Wert ist und r eine Liste. Im folgenden sehen wir eine Modellierung von Listen mit Struktur.

```
1 ;; a list with 0 elements
2 ;; (define list0 the-emptylst)
3 (define list0 empty)
4
5 ;; a list with 1 element
6 ;; (define list1 (make-lst 'a the-emptylst))
7 (define list1 (cons 'a empty))
```

```
8
9  ;; a list with 2 elements
10 ;; (define list2 (make-lst 'a
11 ;;                (make-lst 'b the-emptylst)))
12 (define list2 (cons 'a (cons 'b empty)))
13
14 ;; get the 2nd element from list2
15 ;; (lst-first (lst-rest list2)) -> 'b
16 (first (rest list2)) ;; -> 'b
```

Listen sind ein wichtiger Datentyp, weshalb es einen eingebauten Datentyp existiert. Der Konstruktor *cons* besitzt zwei argumente. cons entspricht unserem *make-lst* Eigenbeispiel und steht wie es leicht zu vermuten ist für *cons*truktor. Zudem existiert eine leere Liste *emtpy* die unserer leeren Liste *the-emptylst* entspricht. Auf die Sektoren der Liste kann man mit *first* und *rest* zugreifen. Mit *g*reift man auf das erste Element und mit *rest* auf den Rest der Liste zu. Zudem haben beide noch *historische Namen* die da lauten *car* für *first* und *cdr* für rest. Die Prädikate *lst?* und *emtpy?* entsprechen *lst?* und *emptylst?*. *lst?* checkt ob es eine Liste ist und *emptylst?* ob die Liste leer ist. Im Folgenden nun ein Beispiel:

```
1  ; a list with 0 elements
2  ;; (define list0 the-emptylst)
3  (define list0 empty)
4
5  ;; a list with 1 element
6  ;; (define list1 (make-lst 'a the-emptylst))
7  (define list1 (cons 'a empty))
8
9  ;; a list with 2 elements
10 ;; (define list2 (make-lst 'a
11 ;;                (make-lst 'b the-emptylst)))
12 (define list2 (cons 'a (cons 'b empty)))
13
14 ;; get the 2nd element from list2
```

```
15 ;; (lst-first (lst-rest list2)) -> 'b
16 (first (rest list2)) ;; -> 'b
```

Wie sie sehen besteht der einzige Unterschied zwischen *make-lst* und *cons* darin, dass *cons* als zweites argument *empty* oder *(cons ...)*. Zum Beispiel :

```
1 (cons 1 2)
```

ist ein Fehler

```
2 (make-lst 1 2)
```

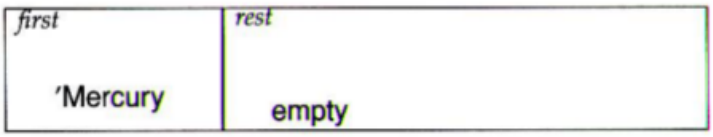
aber nicht.
cons verhindert also inkorrekte Nutzung der Prozedur. In anderen LISP-basierten Dialekten fehlt allerdings diese Überprüfung.

Eine bessere Emulation sähe wie folgt aus:

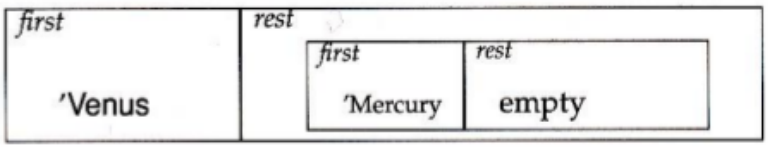
```
1 (define-struct lst (first rest))
2 (define-struct emptylst ())
3 (define the-emptylst (make-emptylst))
4 (define (our-cons a-value a-list)
5   (cond
6     [(emptylst? a-list) (make-lst a-value a-list)]
7     [(lst? a-list) (make-lst a-value a-list)]
8     [else (error 'our-cons "list as second argument expected")]))
```

Dies kann aber nicht verhindern, dass man *make-lst* direkt verwendet. Im folgenden noch einige visuelle Beispiele.

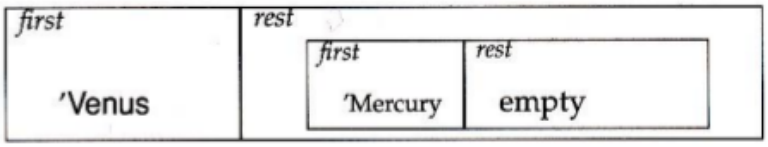
```
1 (cons 'Mercury empty)
```



```
(cons 'Venus
      (cons 'Mercury empty))
```




```
(cons 'Earth
  (cons 'Venus
        (cons 'Mercury empty)))
```



In den folgenden Beispielen sehen wir: Listen speichern beliebige Datentypen, auch gemischte Daten.

```
(cons 0
  (cons 1
    (cons 2
      (cons 3
        (cons 4
          (cons 5
            (cons 6
              (cons 7
                (cons 8
                  (cons 9 empty)))))))))))
```

```
(cons 'RobbyRound
  (cons 3
    (cons true empty)))
```

3.2 Abgeschlossenheitseigenschaft

Eine Operation zum Kombinieren von Daten besitzt die Abgeschlossenheitseigenschaft, wenn die Ergebnisse der Kombination von Datenobjekten wiederum mit der Operation kombiniert werden können. *cons* ist hier ein gutes Beispiel. Solche Kombinationsoperatoren können verwendet werden, um hierarchische Daten aufzubauen.
Doch sind wir der Abgeschlossenheit schon einmal begegnet?
Der Ursprung des Begriffs „Abgeschlossenheit“ (engl. closure) kommt aus der abstrakten Algebra. Es gilt:

> Eine Menge von Elementen ist abgeschlossen bezüglich einer Operation, wenn die Anwendung der Operation an Elementen der Menge wieder ein Element der Menge produziert.

Der Gedanke, dass ein Mittel der Kombination die Abgeschlossenheitseigenschaft besitzen soll, ist intuitiv, jedoch erfüllen Kombinationsoperatoren vieler Programmiersprachen diese nicht. So kann man Elemente kombinieren, indem man diese in Arrays (->T11.84ff) speichert. Man kann aber unter Umständen keine Arrays aus Arrays zusammenstellen bzw. als Rückgabewerte von Prozeduren verwenden. Es fehlt ein eingebauter „Universalkleber“, der es einfacher macht, zusammengesetzte Daten in einer uniformen Art und Weise zu manipulieren.

3.3 list Konstruktor

Längere Listen mit *cons* zu erzeugen ist unhandlich, daher gibt es in den HtDP-TL den Konstruktor *list*. Dieser bekommt eine beliebige Menge von Argumenten und erzeugt eine Liste mit allen Argumenten als Elementen. Als Beispiel:

```
1 (list 1 2 3)
```

statt

```
2 (cons 1 (cons 2 (cons 3 empty)))
```

oder zum Beispiel:

```
(list (list 'a 1) (list 'b 2))
```

Allgemein gilt

```
(list exp-1 ... exp-n)
```

äquivalent zu

```
(cons exp-1 (cons ... (cons exp-n empty))...)
```

3.4 Der ' oder quote Konstruktor

Da Listen wirklich sehr oft eingesetzt werden können einige Listen können mit Hilfe des Quote-Konstruktors ' noch weiter abgekürzt werden.

```
'(1 2 3)
```

ist die Kurzform für

```
(list 1 2 3)
```

Ein weiteres Beispiel:

```
'((1 2) (3 4) (5 6))
```

steht für

```
(list (list 1 2) (list 3 4) (list 5 6))
```

dies sollte man nicht verwechseln mit (list a b c) – list wertet alle Argumente aus, quote jedoch nicht.
Im Allgemeinen bedeutet also

```
'(e-1 ... e-n)
```

steht für

```
(list 'e-1 ... 'e-n)
```

Achtung:
'exp-i ist ein Symbol, wie z.B. *'+'*, *'true*, *'list (!!!)*. Nur für Zahlen gilt *'e-i = e-i*. Beachten Sie, dass diese Regel rekursiv ist! Um list und quote zu verwenden, stellen Sie ab jetzt DrRacket um auf den Sprachlevel "Beginning Student with List Abbreviations" bzw. „Anfänger mit Listen-Abkürzungen"!

3.5 Verarbeitung rekursiver Datentypen

Jetzt kann man sich die Frage stellen wie wir nun unsere Exemplaren rekursiven Datentypen? Die Antwort ist simpel: Wir benutzen unser Designrezept für heterogene Daten.

1. Schritt: Definition des Vertrags, Header etc.
Beachten Sie die Konvention (listof XXX), um im Vertrag zu dokumentieren,
was für Daten als Listenelemente erwartet werden.

```
;; contains-doll?: (listof symbol) -> boolean
;; to determine whether the symbol 'doll occurs
;; in a-list-of-symbols
(define (contains-doll? a-list-of-symbols) ...)
```

2. Schritt: Tabelle erstellen:
Für jeden Datentyp ein cond-Zweig u. Selektoren andeuten

```
(define (contains-doll? a-list-of-symbols)
  (cond
    [(empty? a-list-of-symbols) ...]
    [(cons? a-list-of-symbols)
         ... (first a-list-of-symbols) ...
         ... (rest a-list-of-symbols) ...]))
```

Der Erster Fall (leere Liste) ist trivial

```
[(empty? a-list-of-symbols) false]
```

Mit den verfügbaren Daten können wir direkt das erste Element der Liste überprüfen.

```
[(cons? a-list-of-symbols)
  (cond
    [(symbol=? (first a-list-of-symbols) 'doll) true]
    ... (rest a-list-of-symbols) ...]))
```

Was machen wir mit dem Rest der Liste? Wir brauchen eine Hilfs-Prozedur, die überprüft, ob die (Rest)-Liste das Symbol enthält. Für die Hilfsprozedur nehmen wir *contains-doll?* einfnach selbst. Der folgende Codeteil ist die Lösung.

```
(define (contains-doll? a-list-of-symbols)
  (cond
    [(empty? a-list-of-symbols) false]
    [(cons? a-list-of-symbols)
      (cond
        [(symbol=? (first a-list-of-symbols) 'doll) true]
        [else (contains-doll? (rest a-list-of-symbols))])]))
```

Hier könnte man sich fragen: Wieso ist die Lösung wohldefiniert? (Wohldefiniert bedeutet hier: "die Ausführung der Prozedur endet") Denn nicht jede rekursive Definition ist wohldefiniert z.B.:

```
(define (f a-bool) (f (not a-bool)))
```

Unsere rekursive Definition ist ein Beispiel für strukturelle Rekursion, da Sie Struktur der Prozedur der (rekursiven) Struktur der Daten folgt. Solche rekursiven Definitionen sind stets wohldefiniert, weil die Schachtelungstiefe der Daten in jedem rekursiven Aufruf strikt abnimmt.

3.6 Design von Prozeduren für rekursive Daten

Wie ändert sich also nun unser Designrezept? In der **Datenanalyse und im Design** ändert ich das folgende: Wenn die Problembeschreibung Informationen beliebiger Größe beschreibt, benötigen wir eine rekursive Datenstruktur. Diese Datenstruktur benötigt mindestens zwei Fälle, von denen mindestens einer weder direkt noch indirekt auf die Definition zurückverweist – also eben *nicht* rekursiv ist. Im **Template** müssen die rekursiven Zweige den rekursiven Aufruf andeuten. Im **Prozedurkörper** werden erst die *Basisfälle* (nicht-rekursiv) implementieren, dann die rekursiven Zweige. Für die rekursiven Aufrufe davon ausgehen, dass die Prozedur bereits wie gewünscht funktioniert. Sonst ändert sich nichts weiteres.

3.7 Erzeugen von rekursiven Daten

Prozeduren, die rekursive Daten nach unserem Designrezept verarbeiten, kombinieren üblicherweise die Lösung aus den rekursiven Aufrufen mit den nicht-rekursiven Daten. Diese Kombination besteht häufig in der Konstruktion neuer Daten, deren Struktur analog zu denen der Eingabe ist.

```
1 (define (sum a-list-of-nums)
2   (cond
3     [(empty? a-list-of-nums) 0]
4     [else (+
5   (first a-list-of-nums)
6   (sum (rest a-list-of-nums)))]))
```

Im folgenden Beispiel wird ein Programm zur Lohn für eine oder mehr Personen. Die Schreibweise mit dem "Pfeilïm Namen ist eine Konvention, wenn Werte ümgerechnet"werden. Sie hat *eigene* Semantik.

```
1 ;; wage : number -> number
2 ;; to compute the total wage (at $12 per hour)
3 ;; of someone who worked for h hours
```

```
(define (wage h)
  (* 12 h))
;; hours->wages : (listof number) -> (listof number)
;; to create a list of weekly wages from
;; a list of weekly hours (alon)
(define (hours->wages alon)
  (cond
    [(empty? alon) empty]
                [else (cons
      (wage (first alon))
      (hours->wages (rest alon)))]))
```

3.8 Strukturen die Strukturen enthalten

Strukturen (und Listen) müssen nicht notwendigerweise atomare Daten enthalten. Sie können z.B. Exemplare ihrer eigenen Struktur als Elemente (wie bei der Listenstruktur), aber auch beliebige andere Strukturen/Listen enthalten. Sie können z.B. auch Exemplare ihrer eigenen Struktur als Elemente enthalten (wie bei der Listenstruktur). Zudem können sie beliebige andere Strukturen/Listen enthalten z.B. Liste von Punkten, Liste von Listen von Punkten. Hier ein kleines Beispiel:

Ein Inventareintrag ist eine Struktur *(make-ir s n)*, wobei s ein Symbol ist und n eine (positive) Zahl:

```
(define-struct ir (name price))
```

Eine Inventarliste ist entweder *empty* leer, oder *(cons ir inv)*, wobei ir ein Inventareintrag ist und inv eine *Inventarliste*.

3.8.1 Die Struktur natürlicher Zahlen

Natürliche Zahlen werden häufig als Aufzählungen eingeführt: 0,1,2 etc. Jedoch ist dieses ëtc."nicht besonders hilfreich für die Programmierung. Mit Hilfe einer rekursiven Definition kann man das ëtc."los werden: 0 ist eine natürliche Zahl und falls n eine natürliche Zahl ist, dann auch der Nachfolger (succ n). Diese Definition generiert also

```
(0, (succ 0), (succ (succ 0))
```

usw. Vergleichbar ist dies mit den Peano-Axiomen in der Mathematik. Diese Konstruktion ist analog zu der Konstruktion von Listen.

- **cons** entspricht *succ*
- **rest** entspricht *pred*
- **empty?** entspricht *zero?*

Funktionen auf natürlichen Zahlen können daher analog zu Funktionen auf Listen strukturiert werden. Hier die Definition der Prozeduren pred, succ und zero?:

```
;; pred: N -> N
;; returns the predecessor of n; if n is 0, return 0
(define (pred n)
  (if (> n 0)
    (- n 1)
      0))
;; succ: N -> N
;; returns the successor of n; if negative, return 0
(define (succ n)
  (if (>= n 0)
    (+ n 1)
      0))
;; zero?: N -> boolean
;; returns true if N is 0, else false
;; Already defined in HtDP-TL, no need to do it again!
```

Beispiel 1:

```
;; hellos : N -> (listof symbol)
;; to create a list of n copies of 'hello
(define (hellos n)
  (cond
    [(zero? n) empty]
    [else (cons 'hello (hellos (pred n)))]))
```

Beispiel 2:

```
;; ! : N -> N
;; computes the faculty function
(define (! n)
  (cond
    [(zero? n) 1]
    [else (* n (! (pred n)))]))
```

3.9 Intermezzo: Design von Hilfsprozeduren

3.9.1 Wunschlisten)

Größere Programme bestehen aus vielen verschiedenen (Hilfs-)Prozeduren. Ein sinnvolles Vorgehen hierbei ist das Anlegen und Pflegen einer Wunschliste von Hilfsprozeduren. Eine Wunschliste ist eine Liste der Prozeduren, die noch entwickelt werden müssen, um ein Programm fertigzustellen. Die Prozeduren auf der Wunschliste können wir dann schrittweise mit Hilfe der Designrezepte implementieren. So eine Wunschliste ändert sich häufig im Laufe der Implementierung, etwa weil man entdeckt, dass bestimmte neue Hilfsprozeduren erforderlich sind um andere Prozeduren zum aufen zu bekommen. Die Reihenfolge der Abarbeitung der Wunschliste ist wichtig. Es gibt zwei Lösungsansetze die die Abarbeitung für den Entwickler vereinfachen.

"Bottom-UpAnsatz:

Hier werden erst die Prozeduren implementieren, die von keiner anderen Prozedur auf der Wunschliste abhängen.
Der Vorteil: Wir können sofort und jederzeit testen
Der Nachteil: Am Anfang ist oft noch nicht klar, welche Prozeduren auf der untersten Ebene benötigt werden; es kann der Gesamtüberblick verloren gehen

"Top-DownAnsatz:

Hier wird erst die Hauptprozedur implementieren, dann die dort aufgerufenen Funktionen usw.
Vorteil: Inkrementelle Verfeinerung der Programmstruktur
Nachteil: Man kann erst sehr spät testen; manchmal findet man erst “unten” einen konzeptuellen Fehler öben".

Oft ist aber eineMischung aus Top-Down und Bottom-Up sinnvoll. Hier nun ein Beispiel zum Sortieren einer Liste von Zahlen. Aka. Ïnsertion Sort"

```
1  ;; sort : (listof number) -> (listof number)
2  ;; to create a sorted list of numbers from all
3  ;; the numbers in alon
4  ;; Examples:
5  ;;(sort empty)     empty
6  ;; (sort (cons 1297.04 (cons 20000.00 (cons -505.25 empty))))
7  ;; -> (cons -505.25 (cons 1297.04 (cons 20000.00 empty)))
8  (define (sort alon)
9    (cond
10     [(empty? alon) ...]
11     [else ... (first alon) ... (sort (rest alon)) ...]))
```

Der erste Fall (leere Liste) ist trivial. Der Aufruf *(sort (rest alon))* im rekursiven Fall gibt uns eine sortierte Liste zurück. In diese Liste muss *(first alon)* einsortiert werden. Die

Prozedur, die dies erledigt, kommt auf unsere Wunschliste, die *sort*-Prozedur können wir schon mal komplettieren.

```
;; insert : number (listof number) -> (listof number)
;; to create a list of numbers from n and the numbers
;; in alon that is sorted in descending order; alon is
;; already sorted
(define (insert n alon) ...)
(define (sort alon)
  (cond
    [(empty? alon) empty]
    [else (insert (first alon) (sort (rest alon)))]))
```

Die *insert*-Prozedur kann nun unabhängig von sort implementiert werden. Hierbei ist genaue Analyse der unterschiedlichen Fälle wichtig.

```
;; insert : number (listof number) (sorted)
;;                -> (listof number) (sorted)
;; to create a list of numbers from n and the numbers in
;; alon that is sorted in ascending order; alon is sorted
(define (insert n alon)
  (cond
    [(empty? alon) (cons n empty)]
    [else (cond
    [(<= n (first alon))(cons n alon)]
    [(> n (first alon))(cons
        (first alon)(insert n (rest alon)))])]))
```

3.9.2 Veralgemeinerung von Problemen

Häufig ist ein allgemeineres Problem als das, was man eigentlich lösen möchte einfacher zu verstehen als das eigentliche Problem. Die Abstraktion kann also durchaus die Lösung leichter machen. :)

Doch wie gehen wir hier vor? Die Hilfsprozedur löst ein allgemeineres Problem, doch Hauptprozedur spezialisiert die Hilfsprozedur für das eigentliche Problem. Als Beispiel nehmen wir einen Münzwechsel.
Auf wie viele Arten kann man einen Euro-Betrag a wechseln (unter Verwendung von Münzen à 1,2,5,10,20,50 Cent)? Verallgemeinern wir das Problem zunächst. Auf wie viele Arten kann man einen Euro-Betrag a wechseln, unter Verwendung der ersten n Münzarten aus 1,2,5,10,20,50 Cent? Spezialisierung: n=6
Wechselmöglichkeiten zählen: Einfache Formulierung des allgemeineren Problems als rekursive Prozedur: Die Anzahl der Möglichkeiten, den Betrag a unter Verwendung von n Sorten von Münzen zu wechseln, ist: § 1 falls a = 0 („eine Möglichkeit für Betrag 0: keine einzige Münze“) § 0 falls a<0 oder n = 0 („negativer Betrag / keine Münzen erlaubt“) ansonsten: § Die Anzahl von Möglichkeiten a zu wechseln unter Verwendung aller außer der letzten Münzsorte, plus § Die Anzahl der Möglichkeiten a-d zu wechseln unter Verwendung aller n Münzsorten, wobei d der Nennwert der letzten Münzsorte ist.

4 Auswertungsreihenfolge und Lexikalisches Scoping

4.1 Intermezzo: Syntax und Semantik der HtDP-TL

4.1.1 Syntax

Ähnlich wie die natürlichen Sprachen haben auch die Programmiersprachen ein Vokabular und eine Grammatik. Das Vokabular ist eine Sammlung der "Wörter", aus denen wir Sätzeïn unserer Sprache bilden können. Ein Satz in einer Programmiersprache ist ein Ausdruck oder eine Funktion. Die Grammatik der Sprache sagt uns, wie wir ganze Sätze aus Wörtern bilden. Der Ausdruck Syntax bezieht sich auf Vokabular und Grammatik von Programmiersprachen.

4.1.2 Semantik

Nicht alle grammatikalisch richtigen Sätze sind sinnvoll, weder in Deutsch noch in Programmiersprachen. "Die Katze ist schwarzïst ein sinnvoller Satz. "Der Ziegel ist ein Auto"macht wenig Sinn, auch wenn der Satz grammatikalisch richtig ist.

Um herauszufinden ob ein Satz sinnvoll ist, müssen wir die Bedeutung (Semantik) der Wörter und Sätze verstehen. Für Programmiersprachen gibt es verschiedene Wege,

um den Sinn von einzelnen Sätze/Ausdrücken zu erklären. Den Sinn von HtDP-TL-Programmen diskutieren wir mit einer Erweiterung der bekannten Gesetze aus Arithmetik und Algebra (Substitutionsmodell).

Es existieren vier Kategorien von Wörtern und jedes Wort ist durch eine Zeile definiert.

- Variablen <var>: Namen von Werten
- Funktionen <fct>: Namen von Funktionen
- Konstanten <con>: boolean, Symbole, numerische Konstanten
- Primitive Operationen <prm>: Die Grundfunktionen, die HtDP-TL von Anfang an zur Verfügung stellt

Die Notation geschieht wie folgt: Zeilen zählen einfache Beispiele auf, getrennt durch ein „|“ und Punkte bedeuten, dass es noch mehr Dinge derselben Art in der Kategorie gibt

```
<var> = x | a | alon | ...
<fct> = area-of-disk | perimeter | ...
<con> = true | false | 'a | 'doll | 'sum | ... | 1 | -1 | 3/5 / 1.22 | ...
<prm> = + | - | * | ...
```