Informatik I

Zusammenfassung

Inhaltsverzeichnis

1. Programmieren mit Ausdrücken 3
1.1 Programmieren mit arithmetischen Ausdrücken 3
1.2 Arithmetik mit nicht-numerischen Werten 4
1.3 Auftreten und Umgang mit Fehlern 5
1.4 Kommentare 6
1.5 Bedeutung von BSL Ausdrücken 7
2. Programmierer entwerfen Sprachen! 8
2.1 Funktionsdefinitionen 8
2.2 Funktionen die Bilder produzieren 8
2.3 Bedeutung von Funktionsdefinitionen 9
2.4 Konditionale Ausdrücke 9
2.4.1 Motivation 9
2.4.2 Bedeutung konditionaler Ausdrücke 10
2.4.3 Auswertung konditionaler Ausdrücke (2.4.5 im Skript) 11
2.4.4 Redundanz again(??) (2.4.6 Im Skript) 11
2.5 Definitionen von Konstanten 11
2.6 DRY 11
2.6.1 Bedeutung von Funktions- und Konstantendefinitionen 12
3 Systematischer Programmentwurf 13
3.6 Funktionale Dekomposition 13
3.7 Vom Problem zum Programm 13
3.8 Systematischer Entwurf mit Entwurfsrezepten 14
3.8.1 Testen 14
3.8.2 Informationen und Daten 14
3.8.3 Entwurfsrezept zur Funktionsdefinition 15
3.8.4 Programme mit vielen Funktionen 17
3.9 Information Hiding 17
4. Batchprogramme und interaktive Programme 18
4.1 Batchprogramme 18
4.2 Interaktive Programme 20
4.2.1 Das Universe Teachpack 20
5. Datendefinitionen durch Alternativen: Summentypen 21
5.1 Aufzählungstypen 21
5.2 Intervalltypen 21

5.3 Summentypen 22
5.3.1 Entwurf mit Summentypen 23
5.4 Unterscheidbarkeit der Alternativen 24
6. Daten Definitionen durch Zerlegung: Produkttypen 24
6.1 Die posn Struktur 24
6.2 Strukturdefinitionen 25
6.3 Verschachtelte Strukturen 26
6.4 Datendefinitionen für Strukturen 26
6.5 Ein Ball in Bewegung 27
6.6 Tagged Unions und Maybe 27
6.7 Erweiterung des Entwurfsrezepts 27
7. Datendefinitionen durch Alternativen und Zerlegung: Algebraische Datentypen 28
7.1 Beispiel: Kollision zweier Shapes 28
7.2 Programmentwurf mit ADTs 28
7.3 Refactoring von algebraischen Datentypen 29
8. Bedeutung von BSL 30
8.1 Wieso? 30
8.2 Kontextfreie Grammatiken 30
8.2.1 Beispiele einer Kontextfreien Grammatik 31
8.2.2 Syntax von EBNF in EBNF 32
8.2.3 Konstruktion von Ableitungsbäumen 32
8.3 Syntax von BSL 33
8.4 Die BSL Kernsprache 33

1. Programmieren mit Ausdrücken

1.1 Programmieren mit arithmetischen Ausdrücken

- Algorithmus: Abfolge von Schritten, um eine Aufgabe zu erledigen
- Programmiersprache: formale Sprache, in der Algorithmen programmiert werden können (z.B. BSL)
  - ➢ Viele einfache Algorithmen sind schon in der Programmiersprache enthalten. Z.B. solche zur Arithmetik mit Zahlen
- Ausdruck: z.B. (+ 1 1), aber auch nur Zahlen
- Funktionen: Ausdrücke wie „+“
  - ➢ Beispielfunktionen: +, -, *, /, sqr, expt, sin, …
- Argumente: Operanden, wie z.B. Zahlen
- Zahlenliteral: Zahl, die direkt in einem Ausdruck verwendet wird
- #i -> das Ergebnis ist gerundet
- **Atomare Ausdrücke:** Zahlen
- **Zusammengesetzte Ausdrücke:** Ausdrücke, die mit einer Klammer starten
- Reihenfolge der Auswertungen: Argumente, die zusammengesetzte ausdrücke sind, werden zunächst berechnet; Ist der Unterausdruck selbst geschachtelt, wird das gleiche Verfahren angewendet **-> rekursive Anwendung**

1.2 Arithmetik mit nicht-numerischen Werten

- Datentypen (Texte, Bilder, Wahrheitswerte, …) werden aus Konstruktoren aufgebaut und durch Funktionen angewendet
  - Konstruktoren sind numerische Werte (Zahlenliterale), Funktionen sind z.B. +,-, …
  - Beispiel: „Konzepte der Programmiersprache“ ist ein Stringliteral. Der Konstruktor sind die Anführungszeichen. Eine Funktion auf diesen Datentyp ist string-append.
- Arithmetik Strings: verknüpfen Texte, kehren die Reihe der Buchstaben um, konvertieren von Groß- und Kleinbuchstaben
- Es gibt auch Funktionen die Werte des einen Datentyps als Argumente erwarten und die eines anderen Datentyps als Antwort liefern. Z.B. string-length
  > (+ (string-length „Programmiersprache“) 5)
  24
- Es gibt auch Funktionen, die unterschiedliche Datentypen als Argumente annehmen:
  > (replicate 3 „hi“)
  „hihihi“
- Es gibt auch Funktionen die Datentypen ineinander umwandeln (number->string) und (string->number)
- String= Sequenz von Zeichen
- **Boolsche Werte (=Wahrheitswerte):**
  - Ihre einzigen Konstruktoren sind #true und #false
  - Funktionen: and, or, not
  - Werden auch von Vergleichsfunktionen zurückgegeben; z.B. <, >, =, …
  - Können auch verschachtelt werden
- Bilder als Datentyp:
  - in BSL als „ganz normale Werte“ mit Arithmetik (also Funktionen)
  - Funktionen: image-width, image-hight, Bilder konstruieren, overlay, place-image, …

1.3 Auftreten und Umgang mit Fehlern

1) Vor der Programmausführung auftretende Fehler
   - =statische Fehler
   - Z.B. ob inkorrekte Anzahl von Argumenten eingegeben sind
   - **Syntaxfehler** (Grammatikfehler)
     - ➢ Zahlen nach Funktion, Klammern da, …
   - Auch andere Fehler werden vor dem Ausführen erkannt:

```
> (number->string "asdf")
number->string: expects a number, given "asdf"
> (string-length "asdf" "fdsa")
string-length: expects only 1 argument, but found 2
> (/ 1 0)
/: division by zero
> (string->number "asdf")
#false
```



   - Die Menge der BSL Programme, die eine Bedeutung haben sind nur eine Teilmenge der syntaktisch korrekten BSL Programme

2) Während der Programmausführung auftretende Fehler
   - =Laufzeitfehler

   - Tritt ein Laufzeitfehler auf, wird das Programm abgebrochen & man kann reagieren

   - **Typfehler**
     - ➢ Funktion erwartet eine andere Art von Datentyp als Argument

   - **Aritätsfehler**
     - ➢ Die Anzahl der angegebenen Argumente passt nicht zur Funktion

   - **Nicht-Definierte Funktionen**
     - ➢ Z.B. (/1 0)

   - Nicht alle Fehler führen zum Abbruch der Programmausführung
     - ➢ Z.B. (string-> number „asdf“) führt zur Antwort #false
     - ➢ Programm ist „wohldefiniert“ da sie ihre Aufgabe danach weiter fortsetzten könnte

1.4 Kommentare

- Haben keinen Einfluss auf die Bedeutung eines Programmes

1.5 Bedeutung von BSL Ausdrücken

- Programmieren= Aufschreiben arithmetischer Ausdrücke
  - ➢ Arithmetik umfasst Zahlen, Strings, Boolsche Werte und Bilder
- Programme = syntaktisch korrekt, wenn sie nach den obigen Regeln aufgebaut sind
  - ➢ Nur syntaktisch korrekte Programme haben in BSL eine Bedeutung
- Die Bedeutung eines Ausdrucks in BSL ist ein Wert, der wie folgt ermittelt wird:

  - Zahlen, Strings, Bilder und Wahrheitswerte sind Werte. Wir benutzen im Rest dieser Vorschrift Varianten des Buchstaben *v* als Platzhalter für Ausdrücke die Werte sind und Varianten des Buchstaben *e* für beliebige Ausdrücke (Merkhilfe: Wert = *v*alue, Ausdruck = *e*xpression).
  - Ist der Ausdruck bereits ein Wert so ist seine Bedeutung dieser Wert.
  - Hat der Ausdruck die Form `(f` e_1 ... e_n`)`, wobei `f` ein Funktionsname und e_1,..., e_n Ausdrücke sind, so wird dieser Ausdruck wie folgt ausgewertet:
    - Falls e_1,..., e_n bereits Werte v_1,...,v_n sind und `f` auf den Werten v_1,...,v_n definiert ist, so so ist der Wert des Ausdrucks die Anwendung von `f` auf v_1,...,v_n
    - Falls e_1,...,e_n bereits Werte v_1,...,v_n sind aber `f` ist *nicht* auf den Werten v_1,...,v_n definiert, dann wird die Auswertung mit einer passenden Fehlermeldung abgebrochen.
    - Falls mindestens eines der Argumente noch kein Wert ist, so werden durch Anwendung der gleichen Auswertungsvorschriften die Werte aller Argumente bestimmt, so dass dann die vorherige Auswertungsregel anwendbar ist.

- Diese Vorschriften können auch zur Reduktion von Programmen genutzt werden (werte können nicht reduziert werden)
  - ➢ Reduktion ist wie folgt definiert:

    - Falls der Ausdruck die Form `(f` v_1 ... v_n`)` hat und die Anwendung von `f` auf `v1`,...,`vN` den Wert `v` ergibt, dann `(f` v_1 ... v_n`)` → `v`.
    - Falls ein Audruck e_1 einen Unterausdruck e_2 in einer *Auswertungsposition* enthält, der reduziert werden kann, also $e_2 \rightarrow e_2$', dann gilt $e_1 \rightarrow e_1$', wobei e_1' aus e_1 entsteht indem e_2 durch e_2' ersetzt wird.

    Letztere ist die **Kongruenzregel** mit folgenden Konventionen:

    - $e_1 \rightarrow e_2 \rightarrow e_3$ ist eine Kurzschreibweise für $e_1 \rightarrow e_2$ und $e_2 \rightarrow e_3$
    - Wenn wir schreiben $e \rightarrow^* e$' so bedeutet dies dass es ein $n \geq 0$ und e_1, ...,e_n gibt so dass $e \rightarrow e_1 \rightarrow ... \rightarrow e_n \rightarrow e$' gilt. Insbesondere gilt $e \rightarrow^* e$. Man sagt, $\rightarrow^*$ ist der reflexive und transitive Abschluss von $\rightarrow$.

    Beispiele:

    - `(+ (* 2 3) (* 4 5))` → `(+ 6 (* 4 5))` → `(+ 6 20)` → 26.
    - `(+ (* 2 3) (* 4 5))` → `(+ (* 2 3) 20)` → `(+ 6 20)` → 26.
    - `(+ (* 2 (+ 1 2)) (* 4 5))` → `(+ (* 2 3) (* 4 5))`.
    - `(+ (* 2 3) (* 4 5))` →* 26 aber nicht `(+ (* 2 3) (* 4 5))` → 26.

- **Konfluenz**???
- Zwei Ausdrücke sind Äquivalent, geschrieben $e_1 \equiv e_2$, falls es einen Ausdruck e gibt, so dass $e_1 \rightarrow^* e$ & $e_2 \rightarrow^* e$
- Beispiele:

  - `(+ (* 2 3) 20)` ≡ `(+ 6 (* 4 5))` ≡ 26.
  - `(overlay`  `(rectangle 20 20 "solid" "blue"))` ≡ `(overlay (circle 5 "solid" "red")` `)`.

2. Programmierer entwerfen Sprachen!

- Bisher wurden nur schon vorprogrammierte Definitionen verwendet -> Vokabular
- Nun können wir das Vokabular auch selbst erweitern (= Namen definieren)

2.1 Funktionsdefinitionen

- (define (…)) definiert Namen und ihre Funktion
- Eine Funktionsdefinition **ist kein Ausdruck**!
  - ➢ Kann nach dem definieren aber in Ausdruck verwendet werden
- Sind abstrakter und leichter verständlich
- **DRY-Prinzip**: Wiederholungen vermeiden!
  - ➢ Redundante Programme sind nicht nur anstrengender zu schreiben, sondern auch schwerer zu verstehen
  - ➢ Wiederholende Muster können in Funktionsdefinitionen zusammengefasst werden
- Bodyexpressions = Implementationen der Funktion; hier werden die Parameter der Funktion benutzt
- **Benutzerdefinierte Funktionen** und **primitive Funktionen** (fest eingebaute Funktionen) sehen gleich aus und funktionieren auch gleich

2.2 Funktionen die Bilder produzieren

- Werten auch Bilder als Eingaben aus
- Z.B. place image
- Auch auf Redundanz achten!
  z.B.:

```
(define (create-rocket-scene height)
  (place-image  50 height (empty-scene 100 100)))
```

  Höhe als Parameter, der Rest bleibt gleich, da sie nur „nach oben fliegt“
  - ➢ Der Höhenparameter könnte auch als **Zeitparameter** aufgefasst werden
  - ➢ Die Funktion bildet also Zeitpunkte auf Bildern ab
  - ➢ Das gleiche wie bei Animationen und Filmen
  - ➢ Film kann entstehen durch (animate …), wenn Funktion Parameter davor definiert
    - ➔ Beim Start der Funktion wird eine Stoppuhr gestartet; 28-mal pro Sekunde wird der Zählerwärt um eins erhöht und die Funktion neu ausgewertet

2.3 Bedeutung von Funktionsdefinitionen

- Durch Funktionsdefinitionen können Ausdrücke nicht mehr isoliert (ohne Berücksichtigung des Rests des Programms) ausgewertet werden
  - ➔ werden im Kontext der Menge von Funktionen ausgewertet
  - ➔ Kontext umfasst alle Definitionen, die vor auszuwertendem Ausdruck stehen

> • Falls der Ausdruck die Form `(f` v_1 ... v_n`)` hat und `f` eine primitive (eingebaute) Funktion ist und die Anwendung von `f` auf v_1,...,v_n den Wert `v` ergibt, dann `(f` v_1 ... v_n`)` → `v`.
>
> • Falls der Ausdruck die Form `(f` v_1 ... v_n`)` hat und `f` *keine* primitive Funktion ist und der Kontext die Funktionsdefinition
>
> `(define (f` x_1 ... x_n`) BodyExpression)`
>
> enthält, dann `(f` v_1 ... v_n`)` → `NewBody`, wobei `NewBody` aus `BodyExpression` entsteht, indem man alle Vorkommen von x_i durch v_i ersetzt (für i=1...n).

2.4 Konditionale Ausdrücke

2.4.1 Motivation

- Es wird eine Fallunterscheidung benötigt
- Z.B. Notengebung in Abhängigkeit der Punkte:

```
(define (note punkte)
  (cond
    [(>= punkte 90) 1]
    [(>= punkte 80) 2]
    [(>= punkte 70) 3]

    [(>= punkte 60) 4]
    [(>= punkte 50) 5]
    [(< punkte 50) 6]))
```

2.4.2 Bedeutung konditionaler Ausdrücke

- Allgemein:

```
(cond
  [ConditionExpression1 ResultExpression1]
  [ConditionExpression2 ResultExpression2]
  ....
  [ConditionexpressionN ResultExpressionN])
```

  - ➔ Links Bedingung, rechts Resultat
  - ➔ Fängt oben an; Ist die Erste Bedingung korrekt, ist der Wert des Gesamten cond Ausdrucks der, der 1. Bedingung
  - ➔ Ist die erste Bedingung #false, geht Dr.Racket weiter
  - ➔ Ist keine Bedingung erfüllt, erfolgt eine Fehlermeldung

- Wichtig: e0…en-1 sind in Auswertungsposition; e1…en nicht

```
(cond [e0 e1]
      [e2 e3]
      ....
      [en-1 en])
```

- **else**: wenn ein Zweig immer wahr sein soll, wenn alles andere unwahr ist:

```
(define (note punkte)
  (cond
    [(>= punkte 90) 1]
    [(>= punkte 80) 2]
    [(>= punkte 70) 3]
    [(>= punkte 60) 4]
    [(>= punkte 50) 5]
    [else 6]))
```

≙

```
(cond [e0 e1]
      [e2 e3]
      ....
      [#true en])
```

→ else existiert zwar #true, vereinfacht es aber eig. nicht
⇒ Syntaktischer Zucker!

  - ➔ (darf nur im letzten Zweig verwendet werden)

- **Syntaktischer Zucker**: Fügt einem Sprachkonstrukt nichts neues hinzu, kürzt es aber ab

- **if:** nur eine Bedingung wird überprüft; je nachdem ob erfüllt oder nicht, wird ein Wert ausgegeben

```
(define (aggregatzustand temperatur)
  (if (< temperatur 0) "gefroren" "flüssig"))

> (aggregatzustand -5)
"gefroren"
```

Im Allgemeinen hat ein `if` Ausdruck folgende Form:

```
(if CondExpression ThenExpression ElseExpression)
```

≙

```
(cond [CondExpression ThenExpression]
      [else ElseExpression])
```

↳ if ist auch Syntaktischer Zucker

Cond Ausdruck kann auch als if geschrieben werden, ist halt umständlich

- Cond und if sind gleichmächtig, da sie ineinander umgewandelt werden können

2.4.3 Auswertung konditionaler Ausdrücke (2.4.5 im Skript)

- Cond ist **nicht-strikt**: Die Auswertung bestimmter Argumente ist nicht erlaubt (nur von denen, die auf Auswertungsposition sind)
- Wichtig um Vorgänge terminieren zu können
- **strikt** = alle Argumente werden ausgewertet bevor die Funktion angewendet wird

2.4.4 Redundanz again(??) (2.4.6 Im Skript)

Ausdruck		Vereinfachter Ausdruck	Bedingung
`(if e #true #false)`	≡	`e`	*e* ≡ *#true oder e* ≡ *#f*
`(if e #false #true)`	≡	`(not e)`	–
`(if e e #false)`	≡	`e`	*e* ≡ *#true oder e* ≡ *#f*
`(if (not e-1) e-2 e-3)`	≡	`(if e-1 e-3 e-2)`	–
`(if e-1 #true e-2)`	≡	`(or e-1 e-2)`	*e-2* ≡ *#true oder e-2* ≡
`(if e-1 e-2 #false)`	≡	`(and e-1 e-2)`	*e-2* ≡ *#true oder e-2* ≡
`(if e-1 (f ... e-2 ...) (f ... e-3 ...))`	≡	`(f ... (if e-1 e-2 e-3) ...)`	–

2.5 Definitionen von Konstanten

➔ Siehe Tutorium 3 (I think)

2.6 DRY

- Redundanz vermeiden
- DRY durch Konstanten Definitionen (again: Tut. 3)
  - ➔ Für das Programm wird z.B. „HIGHT“ dann ein fester gültiger Ausdruck, der auch den ihm bei der Definition zugeteilten Wert hat
- **Refactorings**: ändern die Struktur eines Programms, aber nicht sein Verhalten
- **Die Bedeutung von Magic numbers** werden durch **reverse engineering** erkannt
- Bei Konstantendefinitionen ist ihr Standort egal, sie werden im ganzen Programm als bekannt angenommen -> sie haben **globalen Scope**
- Durch eine gute Namenswahl von Funktionen wird die Bedeutung eines Programms viel leichter ersichtlich
- Gibt auch magic images
- Ganz krass:

```
(define WIDTH 100)
(define HEIGHT 100)
(define MIDDLE (/ WIDTH 2))
(define MTSCN (empty-scene WIDTH HEIGHT))
(define ROCKET )
(define ROCKET-CENTER-TO-BOTTOM (- HEIGHT (/ (image-
height ROCKET) 2)))
(define (create-rocket-scene-v6 height)
  (if
    (<= height ROCKET-CENTER-TO-BOTTOM)
    (place-image ROCKET MIDDLE height MTSCN)
    (place-image ROCKET MIDDLE ROCKET-CENTER-TO-
BOTTOM MTSCN)))
```

=

```
(define WIDTH 100)
(define HEIGHT 100)
(define MIDDLE (/ WIDTH 2))
(define MTSCN (empty-scene WIDTH HEIGHT))
(define ROCKET )
(define ROCKET-CENTER-TO-BOTTOM (- HEIGHT (/ (image-
height ROCKET) 2)))
(define (create-rocket-scene-v7 height)
  (place-image
    ROCKET
    MIDDLE
    (if (<= height ROCKET-CENTER-TO-BOTTOM)
        height
        ROCKET-CENTER-TO-BOTTOM) MTSCN))
```

2.6.1 Bedeutung von Funktions- und Konstantendefinitionen

- Die Reihenfolge in der Konstanten definiert werden ist wichtig
- Dr.Racket wertet von oben nach unten aus; bei auftreten einer noch nicht definierten Konstante
  - ➔ Fehlermeldung
  - ➔ Konstantendefinitionen dürfen nur solche Konstanten (und Funktionen) verwenden, die oberhalb der Definition bereits definiert wurden

```
> (define A (+ B 1))
B: this variable is not defined
> (define B 42)
```

-> A hängt von B ab, darum muss B definiert sein

- Die Reihenfolge von Funktionsdefinitionen ist hingegen egal
- Allgemein:

- Ein Programm ist eine Sequenz von Ausdrücken, Konstantendefinitionen und Funktionsdefinitionen. Diese können in beliebiger Reihenfolge auftreten.
- Ein Kontext ist eine Menge von Funktions- und Konstantendefinitionen. Der Kontext ist am Anfang der Programmausführung leer.
- Ein Programm wird von links nach rechts (bzw. oben nach unten) ausgewertet. Hier sind nun drei Fälle zu unterscheiden.
  - Ist das nächste Programmelement ein Ausdruck, so wird dieser gemäß der bekannten Reduktionsregeln im aktuellen Kontext zu einem Wert ausgewertet. Für die Auswertung von Konstanten gilt hierbei `x` → `v`, falls der Kontext die Definition `(define x v)` enthält.
  - Ist das nächste Programmelement eine Funktionsdefinition, so wird diese Funktionsdefinition dem aktuellen Kontext hinzugefügt.
  - Ist das nächste Programmelement eine Konstantendefinition `(define CONSTANTNAME CONSTANTExpression)`, so wird `CONSTANTExpression` im aktuellen Kontext zu einem Wert `v` ausgewertet und zum Kontext die Definition `(define CONSTANTNAME v)` hinzugefügt.

3 Systematischer Programmentwurf

➔ Wie kann systematisch eine Problembeschreibung in ein qualitativ hochwertiges Programm umgeformt werden?

3.6 Funktionale Dekomposition

- Programme bestehen meist aus vielen Funktionen
- Diese sind meist voneinander abhängig, weil sie sich ineinander aufrufen
- Programme sollten so geschrieben werden, dass es pro Aufgabe eine Funktion gibt
  - ➔ Die Aufgaben sollten hierarchisch angeordnet sein (Gesamtaufgaben oben, Verfeinerungen folgen)
  - ➔ Die Funktionen sollten selbstständig die Verfeinerungen zu den jeweiligen Aufgaben aufrufen
  - ➔ Vergleichbar mit einem Baum: Hauptfunktion als Wurzel, Implementierungen, die nur primitive Funktionen aufrufen, als Blätter

  TOP-DOWN: Man beginnt der Hauptfunktion, dann alle Funktionen, die von der Hauptfunktion aufgerufen werden, usw.
- **BOTTOM-UP**: Zunächst die einfachsten Funktionen programmieren, die primitive Funktionen aufrufen, dann solche, die die eben programmierten Funktionen aufrufen usw.
- **hierarchische Abstraktion**: Funktionen können nur Funktionen aus der Schicht genau unter ihnen aufrufen. Dies ist sinnvoll, da man so eine Schicht verstehen kann, ohne alle darunterliegenden Schichten verstehen zu müssen.
  - ➔ **Ermöglicht es hoch komplexe Programme zu verstehen**
  - ➔ Die Grundidee der hierarchischen Abstraktion: Programmierer sollten anhand des Namens und der Dokumentation (Spezifikation) imstande sein, diese Funktion effektiv zu nutzen, ohne alle einzelnen Schritte verstehen zu müssen
  - ➔ Informelle Spezifikationen: Kommentare

3.7 Vom Problem zum Programm

- Unteraufgaben verdienen eine eigene Funktion
- Beispiel S.36 Eintrittspreise
- Ein längeres Programm, das aber besser lesbar und einfacher zu ändern ist, ist besser als ein kompaktes, alles vereinendes Programm

3.8 Systematischer Entwurf mit Entwurfsrezepten

- Bisherig gelernte Sprachkonstrukte (Funktions- und Konstantendefinitionen, Ausdrücke, …) reichen allein nicht aus, um systematisch aus einer Problemstellung ein Programm zu entwerfen
- Relevanz, Konsum und Ausgabe eines Programmes und einzelner Ausdrücke müssen ermittelt werden und einem klar sein
- Checken: sind passende Funktionen schon vorhanden, oder muss alles neu programmiert werden?
- Gute Programme haben kurze Beschreibungen ihrer Funktion, welche Eingaben sie erwarten und welche Ausgaben sie produzieren und sollten leicht neuen Anforderungen anpassbar sein
- Programme sind niemals „fertig“, sie werden immer weiter erneuert und angepasst

3.8.1 Testen

- Systematischer und automatischer Test von Programmen ist fester Teil des Entwurfsrezeptes
  - ➔ Man schaut anhand von Testdaten, ob das Programm den erwarteten Wert ausgibt
- (check-expect (zu testendes Programm) erwarteter Wert)
  - ➔ Funktion selbst fordert zwei gleiche Argumente
- Z.B. (check-expect (+3 2) 5) -> korrekt
  - ➔ Funktion gibt detaillierte Fehlermeldung aus
- Es gibt Varianten wie (check-within) und (check-range)
- Check- expect nimmt beliebige Ausdrücke als Parameter, z.B. auch Bilder
- Beispiel:

```
(check-expect (image-width (ring 5 10 "red")) 20)
```
→ Testet, ob d. Bild wirklich 20 breit ist

3.8.2 Informationen und Daten

- Ein Programm beschreibt eine Berechnung, die Informationen aus der Domäne des Programms verarbeitet und produziert
  - ➔ Problem: Programm kann mit meisten Informationen z.B. Namen nichts anfangen
  - ➔ Wir repräsentieren deshalb Informationen als Daten
  - ➔ Diese können wiederum als Informationen interpretiert werden
  - ➔ Die Beziehung zwischen Daten und Informationen wird in Form von **Datendefinitionen** festgehalten:

```
; Distance is a Number.
; interp. the number of pixels from the top margin of a canvas

; Speed is a Number.
; interp. the number of pixels moved per clock tick

; Temperature is a Number.
; interp. degrees Celsius
```

  - ➔ Oft ist es sinnvoll Datenbeispiele zu den Definitionen anzugeben

```
; Temperature is a Number.
; interp. degrees Celsius
; Examples:
(define sunny-weather 25)
(define bloody-cold -5)
```

3.8.3 Entwurfsrezept zur Funktionsdefinition

- Auf Basis der Trennung zwischen Informationen und Daten kann man den Entwurf einzelner Funktionen als Abfolge von Schritten beschreiben:

1. **Relevante Informationen für die Funktion in Form von Datendefinitionen präsentieren**
   - ➔ Eingabe und Ausgabe
   - ➔ Beispiele für Daten angeben

2. **Signatur, Aufgabenbeschreibung und Funktionskopf schreiben**
   - ➔ **Signatur**: Kommentar, der angibt, wie viele und welche Eingaben die Funktion konsumiert und welche Ausgabe sie produziert
     z.B. string konsumieren, Zahl produzieren
   - ➔ **Aufgabenbeschreibung**: Kommentar, der den Zweck in einer Zeile zusammenfasst; liefert die einfachste Antwort auf die Frage: Was berechnet die Funktion
   - ➔ **Funktionskopf/Header/Stub**: Funktionsdefinition, passend zur Signatur, allerdings nur mit Dummy-Werten eingesetzt (z.B. 0, falls Zahl ausgegeben werden soll, oder (empty-scene 100 100) falls Bild rauskommen soll.
     Trotzdem müssen Name und Eingabeparameter schon festgelegt werden
   - ➔ Beispiel:

```
; Number String Image -> Image
; add s to img, y pixels from top, 10 pixels to the left

(define (add-image y s img)
  (empty-scene 100 100))
```

3. **Tests schreiben**
   Zwischen Aufgabenbeschreibung und Funktionskopf, sie beschreiben anhand von Beispielen, was die Funktion eigentlich macht.

```
; Number -> Number
; compute the area of a square whose side is len
(check-expect (area-of-square 2) 4)
(check-expect (area-of-square 7) 49)

(define (area-of-square len) 0)
```

4. **Templates erstellen**
   Entstehen aus dem Vorhergegangenen. Die Dummy-Werte werden durch Template ersetzt. Man muss sich überlegen, welche Eingabedaten, Hilfsfunktionen und Variablen einem zur Verfügung stehen und daraus dann das Template basteln

```
; Number -> Number
; compute the area of a square whose side is len
(check-expect (area-of-square 2) 4)
(check-expect (area-of-square 7) 49)

(define (area-of-square len) (... len ...))
```

   - ➔ Häufig wird „…“ verwendet, um den Platz frei zu halten

5. **Funktionsbody implementieren**

Template nach und nach durch Ausdrücke ersetzen, die die Spezifikationen (Signatur, Aufgabenbeschreibung und Tests) erfüllen.

```
; Number -> Number
; compute the area of a square whose side is len
(check-expect (area-of-square 2) 4)
(check-expect (area-of-square 7) 49)

(define (area-of-square len) (* len len))
```

Oder

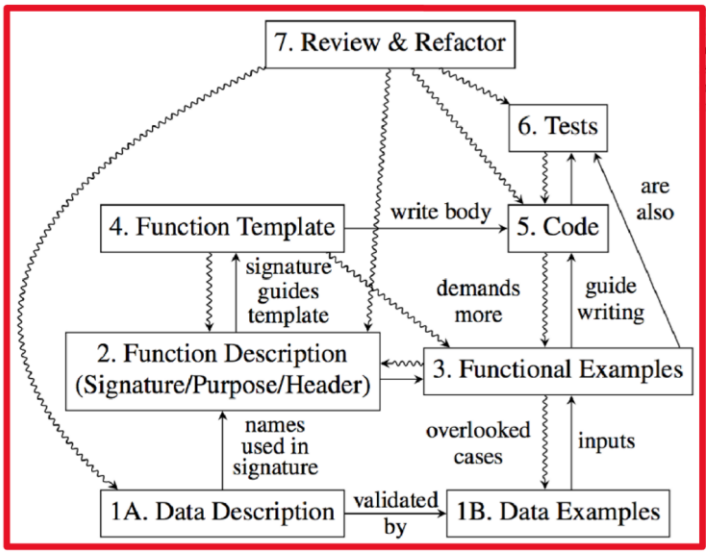
```
; Number String Image -> Image
; add s to img, y pixels from top, 10 pixels to the left
(check-expect (add-image 5 "hello" (empty-scene 100 100))
              (place-image (text "hello" 10 "red") 10 5 (empty-
scene 100 100)))
(define (add-image y s img)
  (place-image (text s 10 "red") 10 y img))
```

6. **Funktion Testen**

Testen ist durch check-expect automatisiert und läuft, sobald Start gedrückt wird. Bei Fehlern -> Reparieren

7. **Nachbearbeitung und Refactoring**

- ➔ Korrektheit der Signatur und Aufgabenbeschreibung überprüfen; Stimmt alles mit der Implementierung überein? (z.B. Anzahl und Aufgabe der Parameter)
- ➔ Wird der Code vollständig durch Tests abgedeckt? Gibt es ein Beispiel für jeden interessanten Fall? (z.B. Randwerte)
- ➔ Entspricht die Funktionsdefinition dem vorgeschlagenen Template?
- ➔ Redundante Codes suchen und ggf. verändern/entfernen
- ➔ Suchen nach Konstanten oder Funktionen, die nicht mehr benötigt werden und entfernt werden können
- ➔ Konditionale Ausdrücke suchen, in denen verschiedene Fälle zusammengefasst werden können
- ➔ Nach jedem Refactoring das Programm testen!



3.8.4 Programme mit vielen Funktionen

- Meist bestehen Programme nicht aus einer, sondern vielen Funktionen
- Für jede Funktion das Entwurfsrezept nutzen
- Template auch für Überfunktion nutzen
- Zwei Herangehensweisen:
  - ➔ **Top-Down Entwurf:** erst Hauptfunktionen programmieren, diese dann zweckmäßig in weitere Hilfsfunktionen zerlegen.
    Da diese Hilfsfunktionen ja am Anfang noch nicht vorhanden sind, bietet es sich an eine **Wunschliste** mit den benötigten Funktionen zu erstellen. Einträge in der Wunschliste bestehen aus Namen, Signatur und Funktionsbeschreibung.
    Dann: Die Liste abarbeiten. Fehlt etwas, das der Liste hinzufügen.
    Vorteile: Großes Entwurfsproblem wird in immer kleinere, direkt lösbare Probleme zerlegt.
    Nachteil: spätes programmieren der Details der Hilfsfunktionen (evtl. können sie gar nicht programmiert werden) und das vollständige Testen der Funktion ist erst zum Schluss möglich. Letzteres Problem kann durch Tests mit **Stubs** (=Dummy-Funktionsdefinition, die eine vordefinierte Antwort zurückliefert) vermieden werden. Der Test-Stub kann dann später einfach durch die Funktionsimplementierung ersetzt werden:

```
; Number -> Number
; computes the area of a cube with side length len


(check-expect (area-of-cube 3) 54)
(define (area-of-cube len) (* 6 (area-of-square len)))

; Number -> Number
; computes the area of a square with side length len
(check-expect (area-of-square 3) 9)
(define (area-of-square len) (if (= len 3) 9 (error "not yet
implemented")))
```

    ->nutzt ERROR, um einen zu erinnern, dass die Funktion noch nicht implementiert ist.

3.9 Information Hiding

- Die Spezifikationen einer Funktion sind: Name, Signatur, Aufgabenbeschreibung und Tests
- Die Spezifikation sollte ausreichend viele Informationen besitzen, dass ein fremder Programmierer die Funktion versteht
- **Information Hiding/Geheimnisprinzip**
  - ➔ Wichtiges Prinzip, um mit der Komplexität großer Programme umzugehen
  - ➔ Programme sind besser lesbar und wartbar, wenn alle Aufrufer von Funktionen sich nur auf die Spezifikationen verlassen
  - ➔ Es sollte einen Unterschied zwischen Spezifikation und Implementation geben: Es kann viele verschiedene Implementationen einer einzelnen Spezifikation geben, diese bleibt dabei aber immer gleich
  - ➔ So können Implementationen geändert und modifiziert werden, die Spezifikation und damit die Aufgabe der Funktion bleibt aber dieselbe

4. Batchprogramme und interaktive Programme

➔ Aus der Perspektive der Benutzung eines Programms können wir mehrere Unterarten dieser Programme unterscheiden

➔ Z.B. Batchprogramme und interaktive Programme

4.1 Batchprogramme

- Bestehen aus einer Hauptfunktion, die Hilfsfunktionen nutzt, die wiederum Hilfsfunktionen nutzen können
- Ein Batchprogramm wird durch die Eingabe der Hauptfunktion aufgerufen und gibt dann ein Ergebnis aus
  - ➔ Während der Auswertung gibt es keine Interaktion mit dem Benutzer
- Eingabe wird oft in Kommandozeilenparametern oder Standardströmen übergeben
- Eingaben auch aus Dateien, deren Name z.B. als Kommandozeilenparameter übergeben wurde
- Ausgabe über **Standardausgabeströme (stdout)**
- Verhalten sich wie die von uns definierten Funktionen
- Beispiel aus BSL: die Funktion „letter“
- Beispiel in Dr.Racket:

```
(require racket/base)

(define (letter fst lst signature-name)
  (string-append
    (opening lst)
    "\n"
    (body fst lst)
    "\n"
    (closing signature-name)))

(define (opening lst)
  (string-append "Sehr geehrte(r) Herr/Frau " lst ","))

(define (body fst lst)
  (string-append
   "Herr Gadafi aus Libyen ist gestorben.\n"
   "Er hat Sie, "  fst  ", in seinem Testament als Alleinerben
eingesetzt.\n"
   "Lieber " fst " " lst ", gegen eine kleine Bearbeitungsge-
buehr\n ueberweise ich das Vermoegen."))

(define (closing signature-name)
  (string-append
   "Mit freundlichen Gruessen,\n"
   signature-name))

(define args (current-command-line-arguments))

(if (= (vector-length args) 3)
    (display (letter (vector-ref args 0)
                     (vector-ref args 1)
                     (vector-ref args 2)))
    (error "Bitte genau drei Parameter uebergeben"))
```

- Ist dieses Programm in der Datei letter.rkt gespeichert, kann s z.B. von Unix oder DOS Shell aufgerufen werden:

```
$ racket letter.rkt Tillmann Rendel Klaus
```

und erhalten dann die folgende Ausgabe:

```
Sehr geehrte(r) Herr/Frau Rendel,
Herr Gadafi aus Libyen ist gestorben.
Er hat Sie, Tillmann, in seinem Testament als Alleinerben
eingesetzt.
Lieber Tillmann Rendel, gegen eine kleine Bearbeitungsgebuehr
 ueberweise ich das Vermoegen.
Mit freundlichen Gruessen,
Klaus Ostermann
```

Dieses Batchprogramm können Sie nun auch mit anderen Kommandozeilenprogrammen verknüpfen; beispielsweise können Sie es mit dem *wc* Programm verknüpfen um die die Anzahl der Wörter in dem generierten Text zu bestimmen:

```
$ racket letter.rkt Tillmann Rendel Klaus | wc -w
     35
```

  - ➔ Es wird die Kommandoversion von Racket verwendet
  - ➔ Racket Programme können aber auch ganz ohne Racket Umgebung ausgeführt werden
  - ➔ Z.B. Menüeintrag: Racket -> Programmdatei; erzeugt eine auf jedem Rechner mit jedem Betriebssystem ausführbare Datei
  - ➔ Teachpack zum lesen oder schreiben von Dateien: batch-io
  - ➔ **Die Stärke von Batchprogrammen ist, dass sie sich leicht und vielfältig mit anderen Batchprogrammen kombinieren**
  - ➔ Verknüpfung über Kommandozeile oder automatisch innerhalb von Batchfiles oder Shell-Skripten

4.2 Interaktive Programme

- Bestehen aus mehreren Hauptfunktionen sowie einem Ausdruck, der den Computer informiert, welche dieser Funktionen welche Art von Eingaben verarbeitet und welche Funktion Ausgaben produziert
    - ➔ Jede der Hauptfunktionen kann natürlich wieder Hilfsfunktionen nutzen

4.2.1 Das Universe Teachpack

- Unterstützt die Konstruktion interaktiver Programme
    - ➔ **Interaktive Programme**: Programme, die auf Zeitsignale, Mausklicks, Tastatureingaben oder Netzwerkverkehr reagieren und graphische Ausgaben erzeugen
- Typisches Konzept **Worldstate**: Zustand einer interaktiven Anwendung
- Der Zustand eines interaktiven Programms ändert sich, wenn bestimmte Ereignisse eintreten
    - ➔ Ereignisse sind z.B. das Drücken auf eine Tastatur oder Maus oder das Ablaufen bestimmter Zeitintervalle (**Timer**)
- Auf Ereignisse wird in Form von **Event Handlern** reagiert
    - ➔ Ein Eventhandler ist eine Funktion, die der gegenwärtigen Worldstate und weitere Informationen über das eingetretene Ereignis als Eingabe erhält und einen neuen Worldstate als Ausgabe produziert
    - ➔ Z.B. on-mouse-event und on-tick-event

    - ➔ Ausführliches Beispiel unter diesem Unterpunkt im Skript

5. Datendefinitionen durch Alternativen: Summentypen

- Datentypen bisher: Zahlen, Strings, Wahrheitswerte sowie in informellen Datendefinitionen definierte Datentypen, wie z.B. Temperatur
  - ➔ Für realistische Anwendungen braucht man aber ein größeres und flexibleres Repertoire zu haben

5.1 Aufzählungstypen

- Zählen die unterschiedlichen Elemente des Typs Wert für Wert auf
- **Aufzählungstypen oder Enumerationstypen**: Datentyp mit einer endlichen Menge von Werten

```
; A TrafficLight shows one of three colors:
; - "red"
; - "green"
; - "yellow"
; interp. each element of TrafficLight represents which col-
ored
; bulb is currently turned on
```

  - ➔ Wenn eine Funktion einen Parameter mit Aufzählungstypen hat, dann enthält das Programm eine **Fallunterscheidung**, in der alle Möglichkeiten des Aufzählungstyps separat voneinander behandelt werden

```
; TrafficLight -> TrafficLight
; given state s, determine the next state of the traffic light

(check-expect (traffic-light-next "red") "green")

(define (traffic-light-next s)
  (cond
    [(string=? "red" s) "green"]
    [(string=? "green" s) "yellow"]
    [(string=? "yellow" s) "red"]))
```

  - ➔ Gibt auch Funktionen, die zur Fallunterscheidung eine Hilfsfunktion aufruft
  - ➔ In Tamplate am besten trotzdem die Funktionsunterscheidung machen

```
(define (traffic-light-next s)
  (cond
    [(string=? "red" s) ...]
    [(string=? "green" s) ...]
    [(string=? "yellow" s) ...]))
```

5.2 Intervalltypen

- Unterscheiden unterschiedliche Teilmengen der Zahlen (oder anderer geordneter Werte)
- Unendliche Zahl möglicher Werte
  - ➔ Lösung: **Intervalle**
  - ➔ Intervall wird durch seine Grenzen definiert
  - ➔ Hat untere und obere Grenze oder ist zu einer Seite offen
  - ➔ WorldState kann auch in Intervallen ausgedrückt werden
  - ➔ Beispiel: 5.2
  - ➔ Intervalle sind nicht für alle Funktionen interessant
  - ➔ Funktionen, für die die Intervalle wichtig sind, beinhalten typischerweise einen cond Ausdruck
  - ➔ Tests für alle Grenzfälle der Intervalle machen

5.3 Summentypen

- Manchmal auch union types genannt
- Verallgemeinern Intervall- und Aufzählungstypen
- Existierende Datentypen und individuelle Werte können beliebig miteinander kombiniert werden
- Ein Summentyp gibt verschiedene Alternativen an, von denen jeder Wert des Typs genau eine Alternative erfüllt
- Z.B. string->number

```
; A MaybeNumber is one of:
; - #false
; - a Number
; interp. a number if successful, else false.
```

Damit können wir für `string->number` folgende Signatur definieren:

```
; String -> MaybeNumber
; converts the given string into a number;
; produces #false if impossible
(define (string->number str) ...)
```

➔ Definition eines Summentyps, der das Verhalten dieser Funktion beschreibt

- Wie bei Intervall- und Aufzählungstypen: Formulierung z.B. durch cond

```
; MaybeNumber -> MaybeNumber
; adds 3 to a if it is a number; returns #false otherwise
(check-expect (add3 5) 8)
(check-expect (add3 #false) #false)
(define (add3 a)
  (cond [(number? a) (+ a 3)]
        [else #false]))
```

- Unterscheiden sich in der Entwurfsmethodik:

5.3.1 Entwurf mit Summentypen

- Das Entwurfsrezept muss ergänzt werden, in Bezug auf Summentypen:
  Beispiel: *the tax on an item i seither an absolute tax of 5 or 10 currency units, or a linear tax. Design a function that coumputes the price of n item after applying ist tax*

1. Falls die Problemstellung Werte in unterschiedliche Klassen unterteilt, so sollten diese Klassen durch eine Datendefinition explizit definiert werden.

   Beispiel: Die Problemstellung oben unterscheidet drei verschiedene Arten der Steuer. Dies motiviert folgende Datendefinition:

   ```
   ; A Tax is one of
   ; - "absolute5"
   ; - "absolute10"
   ; - a Number representing a linear tax rate in percent
   ```

2. Im zweiten Schritt ändert sich nichts, außer dass typischerweise der definierte Datentyp in der Signatur verwendet wird.

   Beispiel:

   ```
   ; Number Tax -> Number
   ; computes price of an item after applying tax
   (define (total-price itemprice tax) 0)
   ```

3. Bzgl. der Tests ist es ratsam, pro Alternative des Datentypen mindestens ein Beispiel zu haben. Bei Intervallen sollten die Grenzen der Intervalle getestet werden. Gibt es mehrere Parameter mit Summentypen, so sollten alle Kombinationen der Alternativen getestet werden.

   Beispiel:

   ```
   (check-expect (total-price 10 "absolute5") 15)
   (check-expect (total-price 10 "absolute10") 20)
   (check-expect (total-price 10 25) 12.5)
   ```

4. Die größte Neuerung ist die des Templates für die Funktion. Im Allgemeinen *folgt die Struktur der Funktionen aus der Struktur der Daten*. Dies bedeutet, dass in den meisten Fällen der Funktionskörper mit einem cond Ausdruck startet, der die unterschiedlichen Fälle des Summentypen unterscheidet.

   ```
   (define (total-price itemprice tax)
     (cond [(number? tax) ...]
           [(string=? tax "absolute5") ...]
           [(string=? tax "absolute10") ...]))
   ```

5. Im fünften Schritt werden die ... Platzhalter durch den korrekten Code ersetzt. Hier ist es sinnvoll, jeden Zweig des cond Ausdrucks einzeln und nacheinander durchzugehen und zu implementieren. Der Vorteil ist, dass Sie sich bei dieser Art der Implementierung immer nur um einen der Fälle kümmern müssen und alle anderen Fälle ignorieren können.

   Möglicherweise stellen Sie während dieses Schritts fest, dass es sinnvoll ist, den cond Ausdruck nach innen zu ziehen (ähnlich wie in `create-rocket-scene-v6` in Abschnitt §2.6.2 "DRY Redux"), oder vielleicht müssen Sie gar nicht alle Fälle unterscheiden, oder vielleicht möchten Sie die Unterscheidung auch in eine Hilfsfunktion auslagern – dennoch ist es in der Regel sinnvoll, mit diesem Template zu starten, selbst wenn am Ende ihre Funktion anders aussieht.

   Beispiel:

   ```
   (define (total-price itemprice tax)
     (cond [(number? tax) (* itemprice (+ 1 (/ tax 100)))]
           [(string=? tax "absolute5") (+ itemprice 5)]
           [(string=? tax "absolute10") (+ itemprice 10)]))
   ```

6. Im letzten Schritt ändert sich nichts, aber überprüfen Sie, dass Sie Testfälle für alle Alternativen definiert haben.

5.4 Unterscheidbarkeit der Alternativen

- Wollten wir statt zwei absoluten Steuersätzen ein kontinuierliches Spektrum, in unserem letzten Beispiel, sähe das so aus:

```
; A Tax is one of
; - a Number representing an absolute tax in currency units
; - a Number representing a linear tax rate in percent
```

- Damit können wir noch nicht umgehen, da wir *tags* dafür benötigen -> nächstes Kapitel

6. Daten Definitionen durch Zerlegung: Produkttypen

- Mehrere Werte so zusammenstellen, dass wir sie wie einen einzelnen Wert behandeln können
  - ➔ Strukturen (*structs*)
- Strukturdefinitionen werden auch Produkte genannt, weil sie dem Kreuzprodukt von Mengen in der Mathematik entsprechen

6.1 Die posn Struktur

- Position in einem Bild definiert durch die Distanz vom linken und die vom rechten Rand
  - ➔ X- und y- Koordinate
- In BSL werden solche Positionen mit Hilfe der **posn** Struktur repräsentiert
  - ➔ Eine **posn** (Position) ist also *ein* Wert, der *zwei* Werte enthält
  - ➔ Wir können eine **Instanz** der posn Struktur mit der Funktion **make-posn** erzeugen
    Die Signatur dieser Funktion ist *Number Number -> Posn*
- Beispiel:

```
(make-posn 3 4)
(make-posn 8 6)
(make-posn 5 12)
```

- Posn hat die gleiche Struktur wie Zahlen oder Strings in dem Sinne, dass Funktionen Instanzen von Strukturen konsumieren oder produzieren können
- Beispiel einer Funktion, die die Distanz einer Position von der oberen linken Bild Ecke berechnet:

```
; Posn -> Number
; to compute the distance of a-posn to the origin
(define (distance-to-0 a-posn) 0)


(check-expect (distance-to-0 (make-posn 0 5)) 5)
(check-expect (distance-to-0 (make-posn 7 0)) 7)
(check-expect (distance-to-0 (make-posn 3 4)) 5)
(check-expect (distance-to-0 (make-posn 8 6)) 10)
```

  - ➔ Kann mit Pythagoras- Funktion berechnet werden
    Funktionsdefinition für x- und y-Koordinate

```
> (posn-x (make-posn 3 4))
3
> (posn-y (make-posn 3 4))
4
```

➔ Können wir distance-to-0 implementieren: (zuerst Template)

```
(define (distance-to-0 a-posn)
  (... (posn-x a-posn) ...
   ... (posn-y a-posn) ...))
```

```
(define (distance-to-0 a-posn)
  (sqrt
    (+ (sqr (posn-x a-posn))
       (sqr (posn-y a-posn)))))
```

6.2 Strukturdefinitionen

- Strukturen wie posn werden normalerweise nicht fest in eine Programmiersprache eingebaut
  - ➔ Es werden nur Mechanismen der Sprache zur Verfügung gestellt, um eigene Strukturen zu definieren
- Strukturen definieren:

```
(define-struct posn (x y))
```

Im Allgemeinen hat eine Strukturdefinition diese Form:

```
(define-struct StructureName (FieldName ... FieldName))
```

  - ➔ Man definiert nicht nur eine Strukturdefinition, sondern gleich einen ganzen Satz an Funktionen:
    - Ein **Konstruktor**: Eine Funktion mit so vielen Parametern, wie die Struktur Felder hat und eine Instanz der Struktur zurückliefert.
      Für posn heißt diese Funktion z.B. make-posn;
      Allgemein heißt sie: **make-Structurename**
    - Pro Feld der Struktur einen **Sektor**: Eine Funktion, die den Wert eines Feldes aus einer Instanz der Struktur ausliest.
      Bei posn heißt diese Funktion posn-x und posn-y;
      Allgemein: StructureName-FieldName
    - **Strukturprädikat**: Boolsche Funktion, die berechnet, ob ein Wert eine Instanz dieser Struktur ist
      Bei posn heißt das Prädikat posn? ;
      Allgemein: StructureName?
  - ➔ Der Rest des Programms nutzt die Funktion, als wäre sie eine primitive Funktion

6.3 Verschachtelte Strukturen

- Position eines Balles kann ja mit posn angegeben werden
- Es gibt verschiedene Möglichkeiten, die Geschwindigkeit und die Richtung eines Objekts zu repräsentieren: z.B. mit dem Bewegungsvektor

```
(define-struct vel (deltax deltay))
```

➔ Deltax und deltay beschreiben, um wie viele Punkte auf der x und y-Achse sich das Objekt bewegt

➔ So können wir berechnen, wie sich der Ort des Objekts verändert

```
; posn vel -> posn
; computes position of loc after applying v
(check-expect (move (make-posn 5 6) (make-vel 1 2)) (make-
posn 6 8))
(define (move loc v)
  (make-posn
   (+ (posn-x loc) (vel-deltax v))
   (+ (posn-y loc) (vel-deltay v))))
```

➔ Ball durch seinen Ort und seinen Bewegungsvektor beschreiben:

```
(define-struct ball (x y deltax deltay))
```

Hier ist ein Beispiel für einen Ball in dieser Repräsentation:

```
(define some-ball (make-ball 5 6 1 2))
```

➔ Unübersichtlich, da Vektoren und Position vermischt sind, besser wäre so:

```
(move (make-posn (ball-x some-ball) (ball-y some-ball))
      (make-vel (ball-deltax some-ball) (ball-deltay some-
ball)))
```

Eine bessere Repräsentation *verschachtelt* Strukturen ineinander:

```
(define-struct ball (loc vel))
```

➔ Noch nicht verschachtelt. Die Schachtelung sieht man, wenn man Instanzen erzeugt:

```
(define some-ball (make-ball (make-posn 5 6) (make-vel 1 2)))
```

➔ Alles bleibt intakt, move wird einfacher:

```
(move (ball-loc some-ball) (ball-vel some-ball))
```

6.4 Datendefinitionen für Strukturen

- Datendefinitionen für Strukturen sollen veranschaulichen, welche Art von Daten jedes Feld enthalten darf

```
(define-struct posn (x y))
; A Posn is a structure: (make-posn Number Number)
; interp. the number of pixels from left and from top
```

Hier sind zwei plausible Datendefinitionen für `vel` und `ball`:

```
(define-struct vel (deltax deltay))
; a Vel is a structure: (make-vel Number Number)
; interp. the velocity vector of a moving object

(define-struct ball (loc vel))
; a Ball is a structure: (make-ball Posn Vel)
; interp. the position and velocity of a ball
```

6.5 Ein Ball in Bewegung

➔ Siehe Skript

6.6 Tagged Unions und Maybe

➔ Habe ich leider nicht wirklich verstanden

6.7 Erweiterung des Entwurfsrezepts

1. Wenn in einer Problembeschreibung Information auftauchen, die zusammengehören oder ein Ganzes beschreiben, benötigt man Strukturen. Die Struktur korrespondiert zu dem "Ganzen" und hat für jede "relevante" Eigenschaft ein Feld.

   Eine Datendefinition für ein Feld muss einen Namen für die Menge der Instanzen der Struktur angeben, die durch diese Datendefinition beschrieben werden. Sie muss beschreiben, welche Daten für welches Feld erlaubt sind. Hierzu sollten nur Namen von eingebauten Datentypen oder von Ihnen bereits definierten Daten verwendet werden.

   Geben Sie in der Datendefinition Beispiele für Instanzen der Struktur, die der Datendefinition entsprechen, an.

2. Nichts ändert sich im zweiten Schritt.

3. Verwenden Sie im dritten Schritt die Beispiele aus dem ersten Schritt, um Tests zu entwerfen. Wenn eines der Felder einer Struktur, die Eingabeparameter ist, einen Summentypen hat, so sollten Testfälle für alle Alternativen vorliegen. Bei Intervallen sollten die Endpunkte der Intervalle getestet werden.

4. Eine Funktion die Instanzen von Strukturen als Eingabe erhält wird in vielen Fällen die Felder der Strukturinstanz lesen. Um Sie an diese Möglichkeit zu erinnern, sollte das Template für solche Funktionen die Selektorausdrücke (zum Beispiel `(posn-x param)` falls `param` ein Parameter vom Typ `Posn` ist) zum Auslesen der Felder enthalten.

   Falls der Wert eines Feldes selber Instanz einer Struktur ist, sollten Sie jedoch *nicht* Selektorausdrücke für die Felder dieser verschachtelten Strukturinstanz ins Template aufnehmen. Meistens ist es besser, die Funktionalität, die diese Unterstruktur betrifft, in eine neue Hilfsfunktion auszulagern.

5. Benutzen Sie die Selektorausdrücke aus dem Template um die Funktion zu implementieren. Beachten Sie, dass Sie möglicherweise nicht die Werte aller Felder benötigen.

6. Testen Sie, sobald Sie den Funktionsheader geschrieben haben. Überprüfen Sie, dass zu diesem Zeitpunkt alle Tests fehlschlagen (bis auf die bei denen zufällig der eingesetzte Dummy-Wert richtig ist). Dieser Schritt ist wichtig, denn er bewahrt Sie vor Fehlern in den Tests und stellt sicher, dass ihre Tests auch wirklich eine nicht-triviale Eigenschaft testen.

   Testen Sie so lange, bis alle Ausdrücke im Programm während des Testens mindestens einmal ausgeführt wurden. Die Codefärbung in DrRacket nach dem Testen unterstützt Sie dabei.

7. Datendefinitionen durch Alternativen und Zerlegung: Algebraische Datentypen

➔ Die Kombination von Summen- und Produkttypen

7.1 Beispiel: Kollision zweier Shapes

➔ Wenn die Alternativen eines Summentyps Produkte sind, so lagert man sie auf Grund ihrer Komplexität besser durch Hilfsfunktionen aus
➔ Beispiel im Skript

7.2 Programmentwurf mit ADTs

➔ Entwurfsrezept ergänzen in den Schritten:

1. Falls ADTs verwendet werden, diese im 1. Schritt definieren, wenn sie nicht schon im Kontext einer anderen Funktion definiert wurde

➔ ADT dann sinnvoll, wenn die Problemstellung unterschiedliche Arten von Infos unterscheidet, die jeweils als Produkttypen formuliert werden können, aber auch ein gemeinsames Konzept haben.
➔ Daten können mit ADTs hierarchisch organisiert werden

2. Alles gleich, Namen der ADTS können aber natürlich definiert werden
3. Bei den Tests bei Summentypen mindestens einen Test pro Alternative definieren

➔ Durch die Schachtelung kann es viel mehr Alternativen geben

4. Bei den Templates gibt es nur zwei Dimensionen: Summen- und Produkttypen

➔ Für Summentypen immer cond ausdrücke
➔ Für Produkttypen meist eine Hilfsfunktion nötig
➔ Kein cond, wenn die Fälle nicht unterschieden werden, sondern bereits existierende Funktionen aufrufen -> siehe overlap/3 Funktion im Bsp. Von oben

5. Funktionen für Templates implementieren (auch für mögl. Hilfsfunktionen)
6. Testen wie immer
7. ADTs auf das DRY Prinzip untersuchen

➔ ADTs zusammenfassen, z.B. mit der posn Funktion

7.3 Refactoring von algebraischen Datentypen

➔ Beispiel:

```
(define-struct student1 (lastname firstname matnr))
; a Student1 is: (make-student1 String String Number)
; interp. lastname, firstname, and matrikel number of a student

(define-struct student2 (matnr lastname firstname))
; a Student2 is: (make-student2 Number String String)
; interp. matrikel number, lastname, and firstname of a student

(define-struct fullname (firstname lastname))
; a FullName is: (make-fullname String String)
; interp. first name and last name of a person

(define-struct student3 (fullname matnr))
; a Student3 is: (make-student3 FullName Number)
; interp. full name and matrikel number of a student
```

```
; Student1 -> Student2
(define (student1->student2 s)
  (make-student2 (student1-matnr s) (student1-lastname s) (student1-firstname s)))

; Student2 -> Student1
(define (student2->student1 s)
  (make-student1 (student1-lastname s) (student1-firstname s) (student1-matnr s)))
```

➔ Den Rest checke ich nicht

8. Bedeutung von BSL

Fast alle Sprachkonstrukte von BSL werden in diesem Kapitel zusammengefasst und die Bedeutung klar gemacht.

Dies geschieht in zwei Schritten:

- ➔ Zuerst die **Syntax der Sprache** definieren. Sie wird in form einer Grammatik definiert und definiert, welche Texte BSL-Programme sind und zerlegt BSL-Programme in seine Teile.
- ➔ Danach definieren wir, was die **grammatikalisch korrekten BSL-Programme bedeuten**. Die Bedeutung wird durch die Definition von Reduktionsschritten festgelegt, die das BSL-Programm zur Auswertung verwendet.

Wir definieren die Bedeutung der Programmiersprache mit dem **Reduktionssemantik** oder **strukturellen operationellen Semantik.**

8.1 Wieso?

Damit man alle Programme ausführen kann und nicht nur dran rumdoktort

8.2Kontextfreie Grammatiken

- ➔ Mit Hilfe von Grammatiken lässt sich präzise beschreiben, wie BSL Programme aussehen
- ➔ Es gibt viele Grammatiken, meist nutzt man aber sogenannte **kontextfreie Grammatiken**
- ➔ Wir verwenden die EBNF (Erweiterte Backus-Naur Form) Notation

8.2.1 Beispiele einer Kontextfreien Grammatik

```
‹Zahl›          ::= ‹PositiveZahl›
                  | -‹PositiveZahl›
‹PositiveZahl›  ::= ‹GanzeZahl›
                  | ‹KommaZahl›
‹GanzeZahl›     ::= ‹ZifferNichtNull› ‹Ziffer›*
                  | 0
‹Kommazahl›     ::= ‹GanzeZahl› . ‹Ziffer›+
‹ZifferNichtNull› ::= 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
‹Ziffer›        ::= 0 | ‹ZifferNichtNull›
```

Beispiele für Texte, die der ‹Zahl› Definition dieser Grammatik entsprechen, sind: `0`, `420`, `-87`, `3.1416`, `-2.09900`.

Beispiele für Texte, die nicht der ‹Zahl› Definition dieser Grammatik entsprechen, sind: `007`, `-.65`, `13.`, `zwölf`, `111Nonsense222`.

- ➔ Die mit spitzen Klammern markierten Bezeichner heißen **Nichtterminale**
- ➔ Die farblich markierten Symbole heißen **Terminalsymbole**
- ➔ Eine Klausel, wie die ersten beiden Zeilen, heißt **Produktion**
  Sie besteht aus einem Nichtterminal links und aus einer Menge von Alternativen, die durch den Strich voneinander getrennt werden.
  Zu jeden Nichtterminal gibt es eine Produktion.
- ➔ Zu jedem Nichtterminal kann man eine Menge an **Ableitungsbäumen** bilden
  Ableitungsbäume entstehen durch das Ersetzen der Nichtterminale in einer der Alternativen der dazugehörigen Produktion durch Ableitungsbäume für diese Nichtterminale -> Die Konstruktion der Ableitungsbäume ist also ein rekursiver Prozess
  Der Prozess stoppt dort, wo man eine Alternative wählt, die nur aus Terminalsymbolen besteht.
- ➔ Ist ein Nichtterminal durch ein Sternchen markiert , so bedeutet das 0 oder mehr Wiederholungen des Nichtterminals (<Ziffer>*)
- ➔ Ist ein Nichtterminal durch ein Plus markiert , so bedeutet das 1 oder mehr Wiederholungen des Nichtterminals (<Ziffer>+)
- ➔ Jeder Ableitungsbaum steht für einen Text, nämlich die Sequenz der Terminalsymbole, die in dem Baum vorkommen, von links nach rechts gelesen.
- ➔ **Die durch eine Grammatik definierte Sprache ist die Menge aller Worte, für die man Ableitungsbäume bilden kann**
- ➔ Bsp. Für das Nichtterminal <Zahl>

Der Ableitungsbaum für `0` ist:

‹Zahl›
  ‹PositiveZahl›
    ‹GanzeZahl›
      0

Der Ableitungsbaum für `420` ist:

‹Zahl›
  ‹PositiveZahl›
    ‹GanzeZahl›
      ‹ZifferNichtNull›
        4
      ‹Ziffer›
        ‹ZifferNichtNull›
          2
      ‹Ziffer›
        0

8.2.2 Syntax von EBNF in EBNF

➔ Selbstanwendung von EBNF, um zu untersuchen wie die Grammatikdefinitionen in EBNF aussehen

➔ Man muss bei diesem Prozess darauf achten zwischen **Metaebene** (die Grammatik der EBNF Grammatiken, auch **Metagrammatik**) und **Objektebene** (eine beliebige EBNF Grammatik, die durch Metagrammatik beschrieben wird) zu unterscheiden
Z.B. wird ::= in der Metagrammatik als Terminalsymbol verwendet, kann aber auch als Trenner zwischen linker und rechter Seite von Produktionen verwendet werden

➔ Beispiel:

Wir definieren die folgenden Nonterminale: ‹*G*› für Grammatik, ‹*P*› für Produktion, ‹*S*› für Sequenz, ‹*E*› Expression ‹*I*› für Item, ‹*V*› für Nichtterminalsymbole (nicht explizit definiert) und ‹*T*› für Terminalsymbole (nicht explizit definiert).

‹*G*› ::= ‹*P*›*
‹*P*› ::= < ‹*V*› > ::= {‹*S*› |}* ‹*S*›
‹*S*› ::= ‹*E*›+
‹*E*› ::= ‹*I*› | ‹*I*› * | ‹*I*› +
‹*I*› ::= < ‹*V*› > | ‹*T*› | { ‹*S*› }

8.2.3 Konstruktion von Ableitungsbäumen

➔ Eine Sequenz von Ableitungsbäumen (**SALB**) ist eine geordnete Menge an Ableitungsbäumen. (benötigt um mit + und * Operanden umzugehen)
Bei Nichtterminalen wird diese Sequenz immer die Länge 1 haben -> am Ende kommt also wieder genau ein Ableitungsbaum heraus

Im folgenden stehe *S* für eine beliebige Sequenz ‹*S*›, *E* für eine beliebige Expression ‹*E*›, *I* für eine beliebiges Item ‹*I*›, *V* für eine beliebiges Nonterminal ‹*V*›, *T* für eine beliebige Terminal ‹*T*›, *Z* für einen beliebigen Ableitungsbaum und *B* für eine beliebige SALB. Wenn wir mehr dieser Platzhalter (auch genannt "Metavariablen", s.u.) benötigen, verwenden wir Indizes wie *B-1*. Für SALB *B-1* bis *B-n* verwenden wir die Schreibweise *B-1*,...,*B-n* zur Konkatenation der Sequenzen zu einer neuen SALB.

Für *S*, *E* und *I* können wir nun die Konstruktion einer SALB wie folgt definieren:

- Falls *S* = *E-1* ... *E-n* und für alle i ist *B-i* ein SALB von *E-i*, so ist *B-1*,...,*B-n* ein SALB für *S*.
- Falls *E* = *I* und *B* ein SALB von *I*, so ist *B* ein SALB für *E*.
- Falls $E = I^+$ oder $E = I^*$, dann ist für alle $k >= 1$ (für +) bzw. $k >= 0$ (für *) *B-1*,...,*B-k* ein SALB für *E*, falls alle *B-i* SALBs für *I* sind.
- Falls *I* = *V* und ‹*V*› ::= ... | S | ... ist eine Produktion der Grammatik und *B* = *Z-1* ... *Z-n* ist ein SALB für *S*, so ist

  V
  Z-1
  ...
  Z-n

  ein SALB (und ein ALB) für *I*.
- Falls *I* = *T* , dann ist *T* ein SALB für *I*.
- Falls *I* = { *S* } und *B* ist ein SALB für *S* , dann ist *B* ein SALB für *I*.

8.3 Syntax von BSL

- ➔ Die Syntax ist durch das bisher besprochene schon fast vollständig
- ➔ Es fehlen noch **lambda-Ausdrücke, Quoting, Zeichendaten, Bibliotheksimporte**
- ➔ **Aspekte, die für die Bedeutung der Sprache irrelevant sind, werden außeracht gelassen**
  z.B. Kommentare, Zeilenumbrüche, Leerzeichen,…
- ➔ Da man diese Aspekte weglässt, handelt es sich um eine **abstrakte Syntax** und nicht um eine **konkrete Syntax**, die diese Aspekte mit einbezieht
- ➔ Die Ableitungsbäume werden dann auch oft **Abstrakte Syntaxbäume** genannt (**AST**)

```
‹program›        ::= ‹def-or-expr›*
‹def-or-expr›    ::= ‹definition›  |  ‹e›
‹definition›     ::= ( define ( ‹name› ‹name›+ ) ‹e› )
                   | ( define ‹name› ‹e› )
                   | ( define-struct ‹name› ( ‹name›* ) )
‹e›              ::= ( ‹name› ‹e›* )
                   | ( cond {[ ‹e› ‹e› ]}+ )
                   | ( cond {[ ‹e› ‹e› ]}* [ else ‹e› ] )
                   | ( if ‹e› ‹e› ‹e› )
                   | ( and ‹e› ‹e›+ )
                   | ( or ‹e› ‹e›+ )
                   | ‹name›
                   | ‹v›
‹v›              ::= < make-‹name› ‹v›* >
                   | ‹number›
                   | ‹boolean›
                   | ‹string›
                   | ‹image›
```

8.4 Die BSL Kernsprache

Idk was er mir sagen will