

ARRGH! MY MAP OF LISTS OF MAPS  
TO STRINGS IS TOO HARD TO  
ITERATE THROUGH! I'LL JUST ASSIGN  
EVERYTHING A NUMBER AND USE  
A \*!\*@ ARRAY



# Informatik II – SS 2014

## Scheme Überblick zu den Vorlesungen

Martin Butz, [martin.butz@uni-tuebingen.de](mailto:martin.butz@uni-tuebingen.de)



# Dr. Racket - Programmierumgebung

- Dr. Racket installieren...
  - <http://racket-lang.org>
  - Download
- Dr. Racket starten...
  - Grundaufbau
    - Definitionsfenster
      - Wo Prozeduren und der Programmablauf definiert werden.
    - Interaktionsfenster (REPL – read evaluate print loop)
      - Wo interaktiv programmiert und ausgeführt werden kann.
  - Hilfebereich sehr nützlich!
    - Tutorials
    - Die Macht der Abstraktion – Eingebaute Prozeduren und Signaturen
    - Zusätzliche Pakete
    - Suchfunktion nach allen verfügbaren Prozeduren, Signaturen etc.



## Anwendungen von Funktionen in Präfix-Notation

| <u>Mathematik</u> | <u>Scheme</u> |
|-------------------|---------------|
| 44-2              | ( - 44 2 )    |
| $f(x,y)$          | ( f x y )     |
| $\sqrt{81}$       | ( sqrt 81 )   |
| $9^3$             | ( expt 9 3 )  |
| $3!$              | ( ! 3 )       |

**Allgemein:**  
 (<Funktionsbezeichner>  
 <argument1> <argument2> ...)

- Allgemein:
  - Ausdrücke werden in Klammern gesetzt
  - Die Eingabe eines abgeschlossenen Ausdrucks (alle Klammern korrekt geschlossen) im Interaktionsfenster führt zur **Auswertung** (auch: **Reduktion**) des Ausdrucks (also zu einem **Berechnungsprozess**).
- Interaktionsfenster funktioniert nach dem „REPL“ Prinzip:
  - Read -> Evaluate -> Print > Read -> Eval ...
  - READ – EVALUATE – PRINT Loop
- Einfache Ausdrücke werden zu sich selbst evaluiert (bzw. reduziert).



# Einfachste Ausdrücke: Literale bzw. Konstanten

---

## Literal

## Signatur („Sorte“)

### Beispiel:

### Typ:

### Signatur-Schlüsselwort:

#t #f

(true, false)

boolean

“xyz“, “0815“, “”

(Zeichenketten)

string

0 1704 815 -42

(Ganze Zahlen)

integer

.42 0.42 3.1415

(Fließkommazahlen)

real

1/3 4/2.5 -1/10

(Rationale Zahlen)

rational



(Bild)

image



## Komplexere Ausdrücke

- Auswertung von komplexeren, zusammengesetzten Ausdrücken von „innen“ nach „außen“ bis alles reduziert wurde...

– ( + ( + 15 15 ) ( + 6 6 ) )

➤ ( + 30 ( + 6 6 ) )

➤ ( + 30 12 )

➤ 42

(Notiz: Parallele Ausführung der beiden inneren Summen ist möglich.)



- Achtung (nicht verwirren lassen):  
Scheme rundet reelle Fließkommazahlen bei arithmetischen Berechnungen (die interne Darstellung ist binär).

## Beispiel:

- $0.7 + \frac{1}{2} / 0.25 - 0.6 / 0.3$
- 2 Möglichkeiten der Umsetzung in Scheme:
- 1.:
  - $(+ 0.7 (- (/ 1/2 0.25) (/ 0.6 0.3)))$
  - $(+ 0.7 (- 2 2))$
  - $(+ 0.7 0)$
  - 0.7
- 2.:
  - $(- (+ 0.7 (/ 1/2 0.25)) (/ 0.6 0.3))$
  - $(- (+ 0.7 2) 2)$
  - $(- 2.7 2)$
  - 0.70000000000000002

## Rationale Zahlen werden „rational“ verarbeitet

- $(- (+ 7/10 (/ 1/2 1/4)) (/ 6/10 3/10))$
- $(- (+ 7/10 2) 2)$
- $(- 22/10 2)$
- 0.7

## Zahlen können quasi beliebig groß werden

- $(* 378647831241746321786237843$   
4326421378467186327862137846487673)
- 16381900719945266624596404890928671939  
67938781392166145609339



**Schlüsselwort `define` weist Namen `<id>` einem Ausdruck `<e>` zu: `(define <id> <e>)`**


**`;` -> kommentiert den Rest der Zeile aus – also ein „Zeilenkommentar“.**

**`# | <TEXT> | #` -> kommentiert den `<TEXT>` aus (auch über mehrere Zeilen) – also ein „Blockkommentar“**

### Beispiele:

```
(define absoluter-nullpunkt -273.15)           ; 0° Kelvin
(define pi 3.141592653)                         ;  $\pi$ 
(define Gründungsjahr-SC-Freiburg 1904)         ; nicht: Schalke 04
(define top-level-domain-germany "de")          ; Internet DNS
```



```
(define sierpinski  )           ; Bild einfügen
(define minutes-in-a-day (* 24 60))              ; 1440
(define vorwahl-tübingen (sqrt 1/2))             ; 0.7071067811865476
```



## Namen / Bezeichner in Scheme

- ... können fast beliebig gewählt werden.
- Regeln:
  - Die folgenden Zeichen dürfen nicht vorkommen:  
`()[]{}", ' ` # | \ ;`
  - Der Bezeichner darf nicht wie eine Zahl aussehen.
  - Im Bezeichner dürfen keine Leer-Charakter vorkommen (keine Leerzeichen, Tabulatoren, oder Zeilenumbrüche)
  - Groß und Kleinschreibung wird dabei ignoriert (`name = NAME`)

Beispiel für gültige Namen bzw. Bezeichner:

|                         |                      |                             |                             |
|-------------------------|----------------------|-----------------------------|-----------------------------|
| <code>+</code>          | <code>!</code>       | <code>euro€-&gt;us\$</code> | <code>meine-variable</code> |
| <code>abenteurer</code> | <code>x-&gt;y</code> | <code>is_alive?</code>      | <code>is-number?</code>     |





# Programmierung von Prozeduren durch Lambda-Abstraktionen

`(lambda (<p1> <p2> ... <pn>) <e>)`

- Wobei:
  - <p1> bis <pn> Bezeichner für Literale einer bestimmten Signatur sind, die im Rumpf der lambda-Abstraktion vorkommen dürfen.
  - <e> sich auf den Rumpf der Prozedur bezieht (der typischerweise relativ komplex ist).
- Aus solch einer Definition entsteht dann eine Prozedur, die erwartet, dass ihr n Werte übergeben werden.

`#<procedure>`

- Mit `define` kann eine solche Prozedur an einen Namen gebunden werden:

`(define <name> (lambda (<p1> ... <pn>) <e>))`



# Signaturen

- Signaturen geben einem Wert bzw. einem mittels `define` spezifizierten Namen, dem ein Wert zugeordnet ist, eine bestimmte Sorte (oder auch Typ).
- Signaturverletzungen werden protokolliert (führen zu Fehlermeldungen).
- Mit folgendem Befehl weist man einem Bezeichner `<id>` eine bestimmte Signatur `<signatur>` zu:

`( : <id> <signatur> )`

- Es gibt einige, bereits eingebaute Signaturen in Scheme:

|                      |                      |                       |
|----------------------|----------------------|-----------------------|
| <code>natural</code> | <code>integer</code> | <code>rational</code> |
| <code>real</code>    | <code>number</code>  | <code>boolean</code>  |
| <code>string</code>  | <code>image</code>   |                       |



## Prozedur-Signaturen

- Einer (mittels) `lambda` programmierten Prozedur, kann eine Signatur zugeordnet werden.
- Diese Prozedur-Signatur spezifiziert, welche Art bzw. Arten von Werten (wiederum Signaturen) die programmierte Signatur verarbeitet und in welchen neuen Wert (mit entsprechender Signatur) die Eingabewerte überführt werden.

- Wird eine Prozedur wie folgt einem Namen zugewiesen:

```
(define <name> (lambda (<p1> ... <pn>) <e>))
```

- Dann kann dieser Prozedur eine Prozedur-Signatur wie folgt zugewiesen werden:

```
(: <name> (<signatur-p1> ... <signatur-pn> -> <signatur-e>))
```



# Testfälle

- Mittels Testfällen kann schon bevor eine Prozedur überhaupt programmiert wurde, festgelegt werden, was die Prozedur für bestimmte Eingabewerte zurückliefern sollte.
- Dabei gibt es zwei grund-test Schlüsselworte:
  - `check-expect` kontrolliert ob zwei Werte gleich sind.
  - `check-within` kontrolliert ob der erste Wert ähnlich des zweiten Wertes ist.

```
(check-expect <e1> <e2>)  
  ; prüft ob die Werte der Ausdrücke e1 und e2 gleich sind.  
(check-within <e1> <e2> <r1>)  
  ; prüft ob die Werte der Ausdrücke e1 und e2  
  ; mit Genauigkeit r1 gleich sind.
```



## Weitere wichtige Signaturen sind:

- `(one-of <ausdruck1> <ausdruck2> ... <ausdruckn>)`
  - Erlaubt es eine Menge von Werten, die in den Ausdrücken `<ausdrucki>` spezifiziert sind, als einen Signatur zu definieren.
  - Beispiel:
    - `(one-of 1 2)`
    - `(one-of "Fehler")`; auch nur ein Ausdruck ist erlaubt
- `(mixed <signatur1> <signatur2> ... <signaturn>)`
  - Erlaubt es verschiedene Signaturen als Werte zuzulassen.
  - Beispiel:
    - `(mixed natural string)`
    - `(mixed number (one-of Fehler: "keine Zahl"))`



# Konstruktionsanleitung für Prozeduren

Als Hilfestellung und Systematisierung der Programmierung in Scheme ist es sehr sinnvoll, der folgenden Konstruktionsanleitung zu folgen:

1. Schreibe eine Kurzbeschreibung der zu programmierenden Prozedur `<name>` mittels eines Kommentars:  
`; Kurzbeschreibung...`
2. Schließe einen Vertrag für die Prozedur `<name>`, durch das spezifizieren einer Prozedur-Signatur:  
`( : <name> ( ... ) )`
3. Bestimme Testfälle, um das gewünscht Verhalten der Prozedur zu bestimmen:  
`(check-expect (<name> ...) ...)`
4. Schreibe das Prozedurgerüst:  
`(define <name> (lambda (...) <rumpf>))`
5. Programmiere den Rumpf der Prozedur



# Top-down Entwurf eines Programms

- Wie gehe ich nun vor, wenn ich ein Programm in Scheme schreiben möchte?
- Top-down stelle ich mir eine Aufgabe vor und abstrahiere Teilaufgaben

## Beispiel: Ziffernblatt zeichnen gegeben eine bestimmte Uhrzeit....:

- Benötige Uhrzeit in Stunden und Minuten (z.B. 16:15)
  - Ergo: Prozedur erwartet zwei Eingabewerte.
- Muss die Orientierung des Stunden und des Minutenzeigers berechnen. Ergo:
  - Schreibe Prozedur, die die Orientierung des Stundenzeigers berechnet: .
    - Diese Prozedur benötigt zwei Eingabewerte (Stunde h und Minute m).
    - Diese Prozedur muss wissen, um wie viel Grad der Stundenzeiger pro Stunden weiterrückt ( $360/12$ ).
    - Im Resultat sollte die Prozedur dann  $h * (360/12) + m * (360/(12*60))$  zurückliefern.
    - Da dabei ( $360/12$ ) zwei mal berechnet werden muss, lohnt es sich in jedem Fall diesen Werte im Voraus zu berechnen... z.B: durch `(define degrees-per-hour 360/12)`.
  - Schreibe Prozedur, die die Orientierung des Minutenzeigers berechnet.
    - Diese Prozedur benötigt einen Eingabewert (Minute m).
    - Diese Prozedur muss wissen, um wie viel Grad der Minutenzeiger pro Minute weiterrückt ( $360/60$ ).
    - Im Resultat sollte die Prozedur somit  $m * (360/60)$  zurückliefern.
    - Da ( $360/60$ ) bei jedem Aufruf neu berechnet wird, lohnt es sich diesen Wert im Voraus zu berechnen... z.B: durch `(define degrees-per-minute 360/60)`  
(klar,  $360/60 = 6$  – man könnte also auch einfach 6 in die Prozedur reinschreiben).
- Am Ende muss ich diese beiden Werte dann benutzen, um ein Bild zu generieren...
  - Ergo: Muss Zeiger und Kreis generieren können und zusammensetzen.



## Reduktionsregeln in Scheme

- Sehr wichtig ist es zu verstehen, wie genau nun ein Ausdruck jeder möglichen Art ausgewertet wird.
- Dabei reicht eigentlich das einfache Regelwerk:  
Wiederhole bis keine Reduktion mehr möglich:
  - Ein Literal bleibt ein Literal (ein Wert bleibt so wie er ist).
  - Ein Bezeichner wird zu dem, wofür er steht, reduziert:  
(z.B.: zu einem Lambda Ausdruck oder zu einem Literal)
  - Ein lambda-Ausdruck selbst bleibt ein lambda-Ausdruck so lange bis alle notwendigen Werte für die Prozedur verfügbar sind.
  - Applikation einer Prozedur  $f$  auf  $e_1 \ e_2 \ \dots$ 
    - Reduziere zunächst alle involvierten Ausdrücke  $f, e_1, e_2 \dots$  zu  $f', e_1', e_2' \dots$
    - Wende dann  $f'$  auf  $e_1', e_2', \dots$  an und zwar:
      - Wenn  $f'$  eine primitive Operation ist (wie z.B. +), dann wende diese an und ersetze das ganze durch das Resultat.
      - Wenn  $f'$  eine lambda-abstraktion ist, dann setze  $e_1', e_2' \dots$  in den Rumpf von  $f'$  ein und liefere den resultierenden Rumpf zurück.





## Namensbindung

- Namen, die in Teachpacks oder in der gewählten Sprache selbst definiert wurden, existieren global.
- Namen, die im Definitionsbereich definiert wurden, existieren nach Ausführung des Definitionsbereichs (Start-Knopf drücken oder Strg-R).
- Namen dürfen nicht doppelt (durch define) definiert werden.
- Namen können aber innerhalb eines lambda-Ausdrucks überschrieben werden.
  - Tip: Durch den Button Syntaxprüfung kann in DrRacket geprüft werden, welcher Name wie bzw. auch wo definiert wurde und in welchem Bereich gültig ist.



# Prädikate = Tests = Prozeduren, die Boolesche (boolean) Werte zurückliefern.

- Prädikate werden sehr häufig in Scheme angewandt.
- Sie können auch genutzt werden, um neue Signaturen zu generieren.
- Beispiel von bekannten Prädikatsignaturen sind:

```
(: = (number number -> boolean))  
(: < (real real -> boolean))  
(: not (boolean -> boolean))  
(: string=? (string string -> boolean))  
(: boolean=? (boolean boolean -> boolean))  
(: zero? (number -> boolean))  
(: even? (integer -> boolean))
```

...



# Fallunterscheidungen in Scheme I

Um verschiedene Fälle zu unterscheiden, gibt es zwei Fallunterscheidungen:

- **(if <t1> <e1> <e2>):**
  - Erwartet dass <t1> ein `boolean` liefert (also entweder ein Boolesches Literal oder ein Prädikat ist) und reduziert daraufhin entweder <e1> (wenn <t1> = #t) oder <e2> (wenn <t1> = #f).
    - Notiz: `if` reduziert immer nur eine der beiden Möglichkeiten (entweder <t1> oder <t2>), niemals beide!
- **(cond (<t1> <e1>) (<t2> <e2>) ... (<tn> <en>) (else <ee>)) :**
  - Wertet nacheinander die Bedingungen <ti> aus
    - Die wieder Boolesche Werte liefern müssen,
  - bis das erste <ti> = #t ist...
  - Ist <ti> = #t und alle <tj> = #f mit 1<=j<i, dann wird der `cond` Ausdruck zu <ei> reduziert.
    - (Weder die <tk> Ausdrücke mit k>i noch die anderen <ej>, <ek> oder <ee> Ausdrücke werden reduziert.)
  - Sind alle <ti> = #f, dann wird der `cond` Ausdruck zu <ee> reduziert.
  - Sind alle <ti> = #f und es gibt keinen (else <ee>) Ausdruck, dann liefert der `cond` Ausdruck einen Laufzeitfehler.



## Fallunterscheidungen in Scheme II

- Wenn Boolesche Rückgabewerte erwartet werden, können aber auch die Booleschen Operatoren `or` und `and` als Fallunterscheidung genutzt werden:
- $(\text{or } \langle t-1 \rangle \langle t-2 \rangle \dots \langle t-n \rangle)$  ist äquivalent zu  $(\text{if } \langle t-1 \rangle \#t (\text{or } \langle t-2 \rangle \dots \langle t-n \rangle))$
- Zusätzlich gilt:  $(\text{or}) == \#f$ 
  - Das bedeutet, dass immer nur so lange die  $\langle t-i \rangle$  Ausdrücke reduziert werden, bis der erste wahre gefunden wurde  $\langle t-i \rangle = \#t$ .
  - Daraufhin wird der Ausdruck zu  $\#t$  reduziert und der Rest wird verworfen.
  - Sind alle Ausdrücke  $\langle t-i \rangle$  false, so werden auch alle zu  $\#f$  reduziert und am Ende wird  $\#f$  zurückgeliefert.
- $(\text{and } \langle t-1 \rangle \langle t-2 \rangle \dots \langle t-n \rangle)$  ist äquivalent zu
- $(\text{if } \langle t-1 \rangle (\text{and } \langle t-2 \rangle \dots \langle t-n \rangle) \#f)$
- Zusätzlich gilt:  $(\text{and}) == \#t$ 
  - Das bedeutet, dass immer nur so lange die  $\langle t-i \rangle$  Ausdrücke reduziert werden, bis der erste Ausdruck sich zu false evaluiert  $\langle t-i \rangle = \#f$ .
  - Daraufhin wird der Ausdruck zu  $\#f$  reduziert und der Rest wird verworfen.
  - Sind alle Ausdrücke  $\langle t-i \rangle$  true, so werden auch alle zu  $\#t$  reduziert und am Ende wird  $\#t$  zurückgeliefert.



---

Records und Zusammengesetzte Daten

# VL 05: RECORDS



## Zusammengesetzte Daten = Records in Scheme

- Records in Scheme legen zusammengesetzte Datenstrukturen fest.
- In einem Record kann also nicht nur ein Datenwert gespeichert werden, sondern eine bestimmte Anzahl von möglicherweise verschiedenem Typ.
- Auch geschachtelte Datenstrukturen sind möglich.
- Durch die Definition eines neuen Records, entsteht auch gleichzeitig eine neue Signatur – nämlich die Signatur des definierten Records.
- Die typische Definition eines Records mit Namen `<t>` sieht wie folgt aus:

```
(define-record-procedures <t>      ; Signaturname
  make-<t>                          ; KONSTRUKTOR
  <t>?                              ; Prädikat
  (<t>-<comp1>
   :
   <t>-<compn>)                    ; Liste von n SELEKTOREN
)
```



## Aus der Record-Definition ergeben sich die Verträge für den Konstruktor des Rekords und die Selektoren des Records:

```
(: make-<t> (<t1> ... <tn> -> <t>))
(: <t>-<comp1> (<t> -> <t1>))
      ⋮
(: <t>-<compn> (<t> -> <tn>))
```

- ... wodurch direkt die Sorten der Komponenten  $\langle t1 \rangle$  bis  $\langle tn \rangle$  des Records festgelegt sind.
- Das Prädikat  $\langle t \rangle?$  überprüft, ob eine Datenstruktur vom Typ  $\langle t \rangle$  ist – also insbesondere, ob diese Datenstruktur von  $\text{make-}\langle t \rangle$  erzeugt wurde.



# Überprüfung von Algebraischen Eigenschaften

- Um zu überprüfen, dass eine Prozedur für beliebige Eingabewerte etwas bestimmtes macht, kann man eine Überprüfung mittels `check-property` vornehmen:

```
(check-property
  (for-all ((<id-1> <signatur-1>)
            :
            (<id-n> <signatur-n>))
    <ausdruck>))
```

- Wobei in dem Ausdruck theoretisch alle möglichen Werte `<id-i>` mit den Signaturen `<signatur-i>` vorkommen dürfen.
- Scheme macht diesen check selektiv mit einigen zufälligen Werten für die Parameter `<id-i>`.
- Mit solche einem Test können insbesondere algebraische Eigenschaften getestet werden.
- Beispiel:  $(\forall x_1, x_2 \in \mathbb{N} : x_1 + x_2 \geq \max \{ x_1, x_2 \})$  entspricht:

```
(check-property
  (for-all ((x1 natural)
            (x2 natural))
    (>= (+ x1 x2) (max x1 x2))))
```





## Prozeduren, die Records nutzen...

- Prozeduren, die Records mit Namen `<id>` kreieren, nutzen typischerweise den Konstruktor des Records, also:

```
(define ...
  (lambda (...)
    ... (make-<id> ...) ...))
```

- Prozeduren, die einen Record Datensatz übergeben bekommen und damit arbeiten, sind typischerweise wie folgt strukturiert:

```
(: <prozedur> (<id> ... -> ...))
(define prozedur
  (lambda (c)
    ... (<id>-<comp1> c) ... ; nutze einen Wert des Records
    ...))
```



GEOCODER

# VL 06: RECORD-BEISPIEL



# Signaturdefinitionen durch Prädikate

- Eine Signatur prüft grundsätzlich, ob ein Wert von einer bestimmten Art ist.
- Gegeben ein beliebiges Prädikat  $\langle p \rangle$ , das einen Wert  $\langle t \rangle$  als Eingabe akzeptiert und einen Booleschen Wert zurückliefert:

$( : \langle p \rangle (\langle t \rangle \rightarrow \text{boolean}) )$

- ... dann kann solch ein Prädikat explizit als Prädikat spezifiziert werden und dann als neue Signatur definiert werden:

$(\text{define } \langle \text{new-signature} \rangle (\text{signature } (\text{predicate } \langle p \rangle)))$

➤ Notiz: Diese neue Signatur ist unabdingbar eine Restriktion von der Signatur  $\langle t \rangle$ !



## Lokale Namen innerhalb einer Prozedur

- Um lokale Namen einem Wert, bzw. einer Teilrechnung, innerhalb einer Prozedur zuzuordnen, kann das Schlüsselwort `let` benutzt werden.

```
(let ((<id-1> <e-1>)
      (<id-n> <e-n>))
  <e>)
```

- Dabei können die Ausdrücke `<e-i>` parallel ausgewertet werden (dürfen also nicht voneinander abhängen) und die Resultate an den respektiven Bezeichner `<id-i>` gebunden.
- Außerdem existieren die Bezeichner dann nur innerhalb des Rumpfs `<e>` des `let`-Ausdrucks.



## Geocoder-Zusammenfassung

- Das Geocoder-Beispiel, wie auch das Abenteuerer Beispiel, sollte insbesondere zeigen, wie komplex und divers Datenstrukturen sein können und wie differenziert es möglich ist, diese Datenstrukturen auf Wertebereiche einzuschränken und geschachtelt zu kombinieren.
- Was in Records nicht möglich ist, ist eine Datenstruktur dynamisch wachsen zu lassen.
- Dafür wird das dynamische ineinander schachteln von Daten ermöglicht.
- Damit beschäftigt sich der Rest des Scheme-Teils der Vorlesung...
  - ... nämlich damit, wie mit einer Liste als beispielhafte dynamische Datenstruktur rekursiv gearbeitet werden kann.



Polymorphe Signaturen

Polymorphe Paare, geschachtelte Paare und Listen

Rekursion, Endrekursion & Letrec

**VL 07/08/09: LISTEN**



## Polymorphe Prozeduren und Paare

- Prozeduren, die unabhängig von den Signaturen ihrer Argumente arbeiten, sollten eine Polymorphe Signatur erhalten.
- Eine polymorphe Signaturvariable wird mittels `%a`, `%b` etc. spezifiziert.

```
(: <prozedur> (%a ... -> %a))
```

- Durch diesen Polymorphismus wird es nun möglich, polymorphe Paare zu definieren.
- Ein Polymorphes Paar besteht aus einer ersten Komponente und dem Rest:

```
(define-record-procedures-PARAMETRIC pair pair-of
  make-pair
  pair?
  (first
   rest))
```

- Wobei `pair-of` eine Signatur mit zwei Signaturparametern ist (`pair-of <t1> <t2>`).



# Listen als Polymorphe Paare

- Eine Liste von Werten der Signatur  $\langle t \rangle$  ist entweder
  - leer
    - (Signatur `empty-list`) spezifiziert durch den Bezeichner `empty`

```
(: empty empty-list)
(: empty? (%b -> boolean))
```

oder

- ein Paar, das aus einem Wert der Signatur  $\langle t \rangle$  und einer Liste (dem Rest der Liste) besteht.
  - (Signatur `(pair-of <t> (list-of t))` )
- Beispiel:
  - `(make-pair 1 (make-pair 2 empty))`
  - `#<record:pair 1 #<record:pair 2 #<empty-list>>>`







# Die wichtigsten Operationen auf Listen

- **Konstruktoren:**

```
(: empty empty-list)          ; konstruiert die leere Liste  
(: make-pair (%a (list-of %a) -> (list-of %a)))  
    ; konstruiert Liste aus Kopf %a und  
    ;      Restliste vom Typ (list-of %a)
```

- **Prädikate:**

```
(: empty? (%b -> boolean)) ; prüft ob die Liste %b leer ist  
(: pair?  (%b -> boolean))  
    ; prüft ob die Liste %b ein Paar, also nicht leer ist
```

- **Selektoren:**

```
(: first ((list-of %a) -> %a))  
    ; Kopfelement der Liste vom Typ (list-of %a)  
(: rest  ((list-of %a) -> (list-of %a)))  
    ; Restliste der Liste vom Typ (list-of %a)
```



# Sprachebene „Macht der Abstraktion“

- Dort ist der Vertrag (list-of %a) eingebaut.
- Dort existiert ein short-cut (syntaktischer Zucker) für die Listenkonstruktion:  

```
(list <e1> <e2> ... <en>)
```

entspricht  

```
(make-pair <e1>
  (make-pair <e2>
    (make-pair ...
      (make-pair <en> empty)...)))
```
- Mit den Ausgabeformen:
  - Für nicht-leere Listen:  

```
#<list x1 x2 ... xn>
```
  - Für eine leere Liste:  

```
#<empty-list>
```
- Weitere nützliche eingebaute Prozeduren:
 

```
(: append ((list-of %a) (list-of %a) -> (list-of %a)))
(: cons (%a (list-of %a) -> (list-of %a)) ; hängt %a vorne an
      ; die Liste an

(: length ((list-of %a) -> natural))
(: list-ref ((list-of %a) natural -> %a))
```



## Listen sind Strukturgerüste.

- Heißt: eine Struktur die Daten miteinander verknüpft.
- Daten können einfache Werte sein (wie Zahlen) oder auch komplexere Strukturen (wie durch einen Record definiert, oder selbst Listen).

## Listen können rekursiv verarbeitet werden.

- Eine grundsätzliche Konstruktionsanleitung charakterisiert diese Verarbeitung:

Einfach Rekursion:

```
(: <f> ((list-of <t1>) -> <t2>))
(define <f>
  (lambda (xs)
    (cond ((empty? xs) ...)
          (else
           ... (first xs) ...
           ... (<f> (rest xs)) ...))))
```

Bemerkungen:

- |                   |                             |
|-------------------|-----------------------------|
| - (first xs)      | hat Signatur <t1>           |
| - (rest xs)       | hat Signatur (list-of <t1>) |
| - (<f> (rest xs)) | hat Signatur <t2>           |

Rekursion mit Berücksichtigung des letzten Wertes in der Liste:

```
(: <f> ((list-of <t1>) -> <t2>))
(define <f>
  (lambda (xs)
    (cond ((empty? xs) ...)
          ((empty? (rest xs)) ...)
          (else
           ... (first xs) ...
           ... (<f> (rest xs)) ...))))
```



# Rekursion über Natürlichen Zahlen

- Rekursion über Listen ist vom Schema her ähnlich wie das über natürliche Zahlen...
- Denn die natürlichen Zahlen, bestehen ja wie eine Liste
  - aus einem ersten Element – das typischerweise als 0 bezeichnet wird – und
  - einer Liste von folgenden Zahlen –  
nämlich  $0+1$ ,  $0+1+1$ , ...  
(also 0 1 2 ...)
- So kann man sich, zum Beispiel, die Fakultätsfunktion, als Rekursion über die natürlichen Zahlen vorstellen, während der die einzelnen Zahlen miteinander multipliziert werden.
  - Vorsicht – in dem Fall beginnen wir mit 1 denn  $0! = 1$ .

Rekursionsgerüst über natürliche Zahlen:

```
(: <f> (natural -> number))  
(define <f>  
  (lambda (n)  
    (cond ((= n 0) ...)   
          (else  
            ... n ...  
            ... (<f> (- n 1)) ...))))
```



# Rekursion im Allgemeinen

- Rekursive Prozeduren sind Prozeduren, die sich selbst in ihrem Rumpf aufrufen.
- Wichtige allgemeine Konstruktionshinweise:
  - Eine rekursive Prozedur sollte immer
    - (irgendwo) VOR dem rekursiven Aufruf nach einem Abbruchkriterium fragen
      - Zum Beispiel: Ist die leere Liste erreicht? Oder: Ist die Zahl 0 erreicht?
    - Und wenn gegeben, entsprechend die Rekursion mit der Rückgabe eines Basisresultats beenden.
      - Zum Beispiel: Dann gebe die leere Liste zurück. Oder: Dann gebe die Zahl 1 zurück.
  - Beim rekursiven Aufruf einer Prozedur sollte die noch zu bearbeitende Datenstruktur kleiner werden.
    - Zum Beispiel: Ein Element weniger in der Liste. Oder: Die nächste kleinere natürliche Zahl.
  - Deswegen wird bei einer rekursiven Verarbeitung einer Liste bei dem rekursiven Aufruf typischerweise die Restliste übergeben.
  - Deswegen wird bei einer rekursiven Verarbeitung von einer natürlichen Zahl, bei dem rekursiven Aufruf typischerweise eine (meist um 1) verkleinerte Zahl übergeben.



# Endrekursion

- Die einfache Rekursion kann leicht eine sehr lange Kette von noch abzuarbeitenden Befehlen auf dem Stack generieren.
  - Beispiel: `(factorial 30000)` – muss beim Erreichen der 0 noch 30000 Multiplikationen durchführen -> das kostet temporär viel Speicherplatz!
- Deswegen sollten aufwendige Rekursionen ENDREKURSIV programmiert werden.
  - Das bedeutet, dass das Resultat des rekursiven Aufrufs gleichzeitig das Endresultat ist.
- Nicht alle Rekursionen können direkt Endrekursiv umgeschrieben werden.
- Sind die noch auszuführenden Operationen allerdings assoziativ, dann ist das Umschreiben in eine endrekursive Prozedur relativ einfach.
- Generiere eine „Worker-Prozedur“, die das aktuelle Teilresultat im rekursiven Aufruf mit übergibt.



## Beispiel: Liste Umdrehen „backwards“

- **Einfach Implementierung von backwards** ist
  - NICHT endrekursiv
  - SEHR langsam
- Wieso so langsam?
  - Die Prozedur append geht durch die gesamte Liste, um das erste Element hinten anzuhängen.
  - Append wird N-mal ( $N = \text{Länge der Liste}$ ) aufgerufen.
  - Dadurch entsteht ein **quadratischer Berechnungsaufwand**  $O(N^2)$
- **Lösung:**
  - Programmiere die Lösung **endrekursiv!**
  - Nutze die Tatsache, dass es sehr viel einfacher ist, ein Element **vorne** an eine (einfache) Liste **anzuhängen!**
  - Benutze **letrec**, um die eigentliche Rekursion (Worker) direkt in der Hauptprozedur implementieren zu können.

```
(: backwards ((list-of %a) -> (list-of %a)))
(define backwards
  (lambda (xs)
    (cond ((empty? xs) empty)
          (else
           (append (backwards (rest xs))
                    (list (first xs)))))))
```

```
(: backwards-letrec ((list-of %a) -> (list-of %a)))
(define backwards-letrec
  (lambda (xs)
    (letrec
      ((backwards-worker
        (lambda (xs result)
          (cond ((empty? xs) result)
                ((pair? xs)
                 (backwards-worker
                  (rest xs)
                  (make-pair (first xs)
                             result))))))
      (backwards-worker xs empty))))
```



Prozeduren verarbeiten und/oder liefern Prozeduren

Listenfaltung

Big-Bang Umgebung

# **VL 10/11/12: HIGHER ORDER PROZEDUREN**





# Prozeduren Höherer Ordnung

- HIGHER ORDER PROZEDUREN sind Prozeduren, die andere Prozeduren als Argumente übergeben bekommen, oder die eine Prozedur zurückliefern.
- HOPs sind besonders gut dafür geeignet, eine Datenstruktur systematisch zu verarbeiten.
- Dabei können Datenstrukturen ganz verschiedenartig sein, wie zum Beispiel:
  - Listen
  - Bilder
  - Graphen
  - Datenbanken
  - Etc.



# Listenfaltung

- Higher Order Prozeduren auf Listen angewandt, abstrahieren über die Operationen, die auf die Listenelemente angewandt werden.
- Wenn die gleiche Prozedur, auf jedes Listenelement angewandt werden soll, so spricht man auch von einer Listenfaltung.
- Bei der Listenfaltung müssen insbesondere zwei Dinge bestimmt werden:
  1. Der Rückgabewert, wenn das Ende der Liste – also die leere Liste – erreicht ist. (z.B. die leere Liste selbst – also `empty` – oder auch die Zahl 0).
  2. Die Prozedur, die das aktuelle, erste Listenelement mit dem Resultat, das der rekursive Aufruf gegeben die Restliste zurückliefert, verrechnet – und wiederum dann das Resultat gegeben die aktuelle Liste zurückliefert.



# Die Prozedur Fold-Right

- Konsumiert zwei Attribute und die Liste selbst:
  1. Resultat Wert bei der leeren Liste vom Typ %b.
  2. Prozedur, die den ersten Wert der Liste mit dem Resultat der Restliste verarbeitet (%a %b -> %b).
  3. Die zu verarbeitende Liste: (list-of %a)
- Ist in Scheme „Die Macht der Abstraktion“ unter dem Namen fold verfügbar:  
 (: fold (%b (%a %b -> %b) (list-of %a) -> %b) )

**; Falte Liste xs bzgl. des Abbruchwertes z und der Prozedur p**


```
(: foldr (%b (%a %b -> %b) (list-of %a) -> %b))
(define foldr
  (lambda (z p xs)
    (cond ((empty? xs) z)
          (else
           (p (first xs)
              (foldr z p (rest xs)))))))
```



## Teachpack universe.ss

- Teachpack "universe" nutzt H.O.P., um Animationen (= Sequenzen von Szenen/Bildern) zu definieren:

```
(big-bang
  <init>
  (on-tick <tock> <r>)
  (on-draw <render> <w> <h>))
```

  
 optional

- (: <init> %a)
  - Startzustand des „Universums“.
- (: <tock> (%a -> %a))
  - Funktion, die neuen Zustand aus altem Zustand berechnet.
  - Diese Funktion wird 1/<r> Mal pro Sekunde aufgerufen
- (: <render> (%a -> image))
  - Funktion, die aus aktuellem Zustand eine Szene berechnet
    - (wird in Fenster mit Dimensionen <w> x <h> angezeigt)
- Beim Schließen der Animation wird der letzte Zustand zurückgegeben.



## Prozedurfabriken 2ter Ordnung

- Komposition von Prozeduren  $((\text{compose } f \ g) \ x)$  entspricht  $(f \ (g \ x))$  produziert neue Prozedur:

- `(: g (%a -> %b))`
- `(: f (%b -> %c))`
- `(: compose ((%b -> %c) (%a -> %b) -> (%a -> %c)))`  
`(: compose ((%b -> %c) (%a -> %b) -> (%a -> %c)))`  
`(define compose`  
`(lambda (f g)`  
`(lambda (x)`  
`(f (g x))))`

- Wiederholter (n-facher) Aufruf einer Prozedur produziert neue Prozedur .

- `(repeat (natural (%a -> %a) -> (%a -> %a)))`  
`(: repeat (natural (%a -> %a) -> (%a -> %a)))`  
`(define repeat`  
`(lambda (n f)`  
`(cond ((= n 0) (lambda (x) x))`  
`(> n 0) (compose f (repeat (- n 1) f))))`



# Schönfinkel-Isomorphismus

- Manche Prozeduren, benötigen direkt zwei Elemente für die Verarbeitung... manchmal sind aber nicht beide direkt verfügbar.
- Deswegen ist es möglich, solche Prozeduren zu „curryfizieren“ – also so umzuschreiben, dass zunächst ein Element verarbeitet wird und eine neue Prozedur erzeugt wird, die dann (später) das zweite Element zusammen mit dem ersten verarbeitet.
  - Entdeckt von den Mathematikern Moses Schönfinkel und Haskell Curry
- Die Prozedur „curry“ macht genau das.
- Die Prozedur „uncurry“ hingegen macht das Umgekehrte.
- Die Hintereinanderreihung von „curry“ und „uncurry“ produziert eine Prozedur, die isomorph zur ursprünglich gegebenen Prozedur ist.

```
(: curry ((%a %b -> %c) -> (%a -> (%b -> %c))))
```

```
(define curry
```

```
  (lambda (f)
```

```
    (lambda (x)
```

```
      (lambda (y)
```

```
        (f x y))))
```

```
(: uncurry ((%a -> (%b -> %c)) -> (%a %b  
-> %c)))
```

```
(define uncurry
```

```
  (lambda (f)
```

```
    (lambda (x y)
```

```
      ((f x) y))))
```



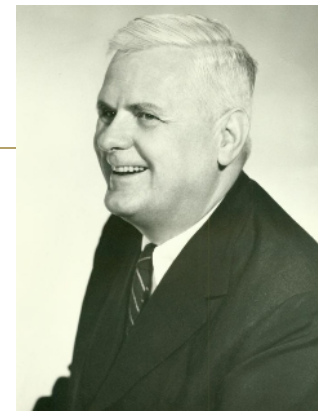
# Mengen als Prozeduren

- Teilmengen (zum Beispiel, der Ganzen Zahlen) können als Prozeduren definiert werden, die  $\#t$  (wahr) zurückliefern, wenn eine ganze Zahl Teil der Menge ist.
  - Neue Signatur einer solchen Mengenschreibweise:
    - Für ganze Zahlen:
 

```
(define set-of-integer (signature (integer -> boolean)))
```
    - Allgemein:
 

```
(define set-of (lambda (t) (signature (t -> boolean))))
```
- Die Leere Liste ist durch diese Sichtweise auf Mengen nichts anders, als die Prozedur die immer  $\#f$  (falsch) zurückliefert:
 

```
(define empty-set (lambda (x) #f))
```
- Kombinationen von Mengen, können durch diese Sichtweise auf Mengen mit logischen Operatoren realisiert werden!
  - Beispiel: Vereinigung, Schnitt, Komplement, Zufügen eines Elements...



# Das Lambda-Kalkül

- Eingeführt von Alonzo Church 1936.
- Ist die allgemeine Grundlage jeder funktionalen Programmiersprache.
- Ist definiert durch die allgemeine Basis eines Ausdrucks, der
  1. Eine Variable  $var$ ;
  2. Eine Anwendung  $app$  eines Ausdrucks  $e1$  auf einen Ausdruck  $e2$ ;
  3. Ein lambda-Ausdruck  $lam$ , der über einen Wert in einem Ausdruck  $e1$  mittels einer (noch freien) Variablen  $v$  abstrahiert ( $\lambda v.e1$ ), sein kann.
- Das Lambda-Kalkül ist ein Grundkonzept in der theoretischen Informatik. (siehe Informatik III)





# Logische Ausdrücke mit dem Lambda-Kalkül

- Logische Werte  $\#t$  und  $\#f$  können als Basis-Lambdaausdrücke verstanden werden:
  - $\text{TRUE} \equiv (\lambda x.(\lambda y.x))$  ; ignoriere zweites Argument, liefere erstes
  - $\text{FALSE} \equiv (\lambda x.(\lambda y.y))$  ; ignoriere erstes Argument, liefere zweites
  - $\text{IF-THEN-ELSE} \equiv (\lambda x.x)$
- Auf diese Art ist TRUE bzw. FALSE als Basis-Lambda-Ausdruck definiert, der in der Verzweigung if-then-else entsprechend zur Anwendung kommt.
- Logische Operatoren können nun durch Reduktionen basierend auf den zwei Basisstrukturen realisiert werden.
  - Zum Beispiel:
    - $\text{AND} = (\lambda a.(\lambda b.((b\ a)\ b)))$ 
      - Konsumiert zwei Boolesche Werte... ist der erste TRUE so ist das Resultat der Wahrheitswert des anderen, ist der erste FALSE, so ist das Resultat dieser Wert (also FALSE).
    - $\text{OR} = (\lambda a.(\lambda b.((b\ b)\ a)))$ 
      - Konsumiert zwei Boolesche Werte... ist der erste FALSE so ist das Resultat der Wahrheitswert des anderen, ist der erste TRUE, so ist das Resultat dieser Wert (also TRUE).

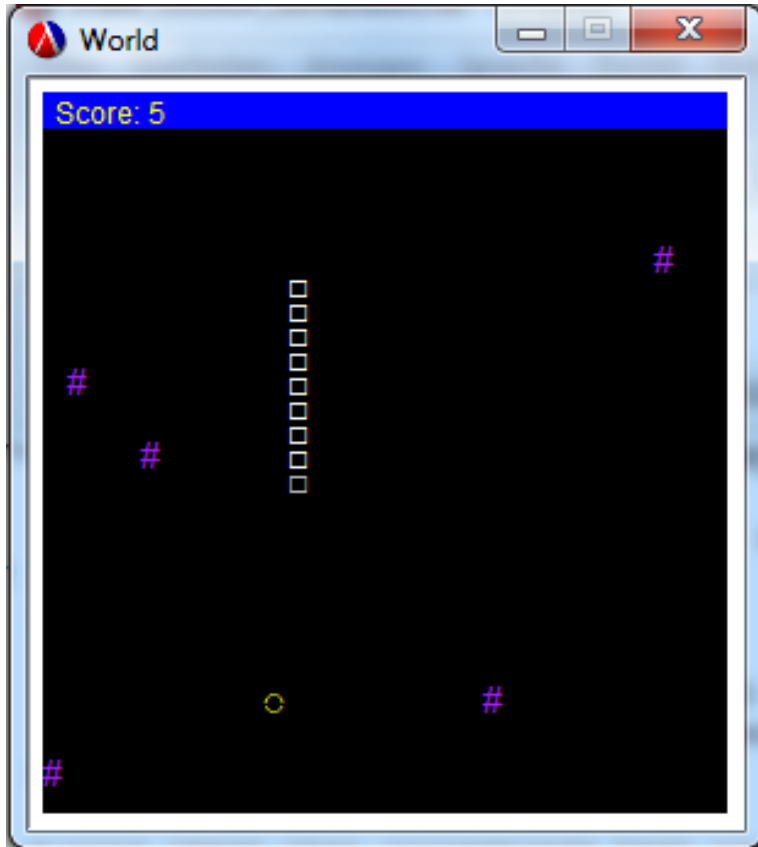


# Das Lambda-Kalkül in der Linguistik

- Das Lambda-Kalkül kommt auch in der Linguistik zum Einsatz.
  - Insbesondere zur Bestimmung der Korrektheit einer grammatikalischen Struktur.
  - Zum Beispiel:
    - Prüfe ob die Syntax eines Satzes korrekt ist.
    - Generiere eine Prozedur, die wahr zurück gibt, wenn ein Satz korrekt strukturiert ist.
    - Resultat: Menge aller möglichen Sätze.
- Lambda Ausdrücke sind dadurch Prozeduren, die Worte verarbeiten und Satzteile oder auch ganze Sätze generieren.
- Sätze werden auf Konsistenz getestet, in dem weitere Eigenschaften für bestimmte Worte mit abgefragt werden.
- Beispielhafte Konstruktion:
  - Verb: Eine Prozedur, die ein Subjekt erwartet ODER ein Subjekt und ein Objekt.
  - Nomen: Variablen, die mit Verb-Prozeduren zu Sätzen kombiniert werden können.
  - Relativpronomen: steht für Subjekt oder Objekt im Nebensatz und bezieht sich auf Subjekt oder Objekt im Hauptsatz.



# Spielbeispiel zum Abschluss: Snake



- **Records:**

- Der Spielzustand ist ein Record, der alle relevanten Aspekte kodiert.
- Das Spielfeld ist ein Grid aus Segmenten.
- Der Spiel-Record (`game`) selbst besteht aus der Schlange (`game-snake`), dessen Richtung (`game-dir`), der Position des Futters (`game-candy`), dem Spielfeld (`game-field`) und dem aktuellen Spielstand (`game-score`).

- **Listen:**

- Der Schlangenkörper wird als Liste dargestellt.

- **Rekursionen:**

- Schlangenkörper wird auf Kollision untersucht (Kopf trifft auf Körper?).

- **HOPs mit Fold:**

- Schlangenkörper wird rekursiv auf das Bild gefaltet (gemalt).



# Spielbeispiel zum Abschluss: Death-Star-Fighter

## • Records:

- Der Spielzustand ist ein Record, der alle relevanten Aspekte kodiert.
- Das Spielfeld ist ein (nahezu) kontinuierlicher Raum.
- Der Spiel-Record (*game*) besteht aus der aktuellen Zeit (*time*), der Position des X-Wings (*pos*), dessen rechts-links Flugrichtung (*lrdir*), die Geschwindigkeit des Hintergrundes (*scroll-speed*), einer Liste von Schüssen (*bullets*) und einer Liste von gegnerischen Flugzeugen (*tie-fighters*).

## • Listen:

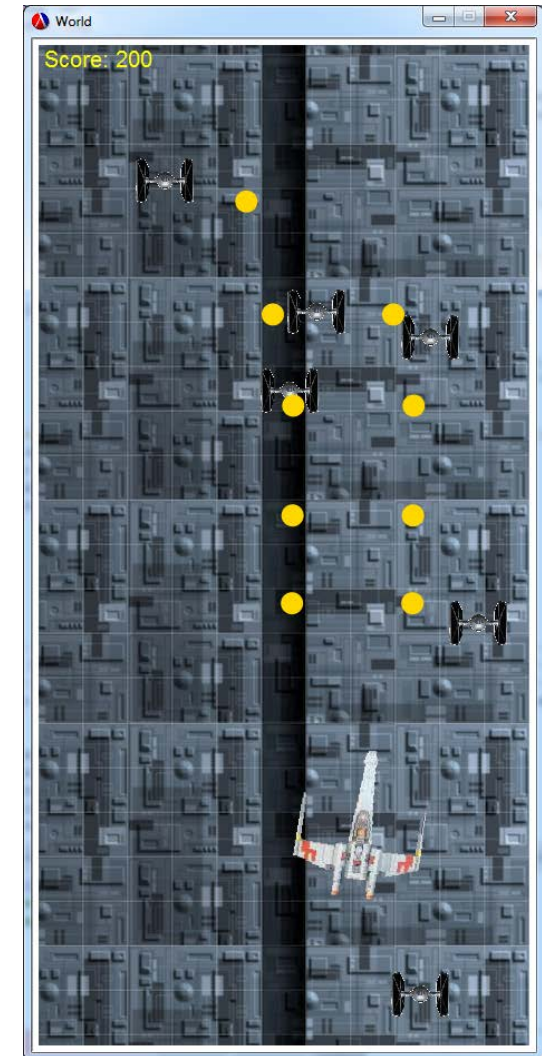
- Die Geschosse und aktuellen gegnerischen Flugzeuge werden als Listen dargestellt.

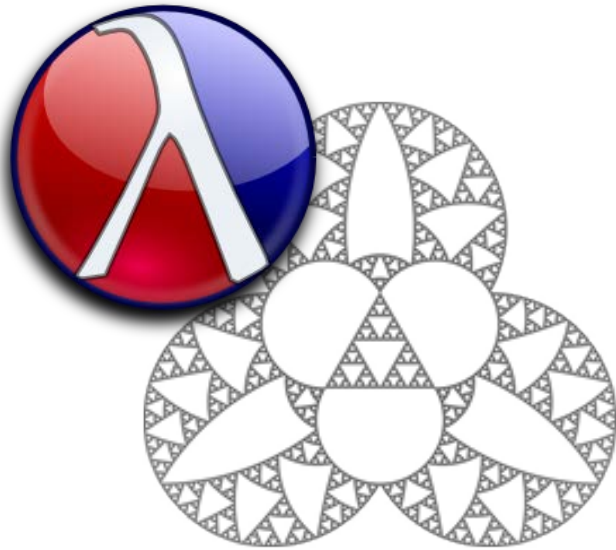
## • Rekursionen:

- Geschosse werden auf Zusammenstoß mit Flugzeugen rekursiv untersucht.
- Flugzeuge werden auf Kollisionen mit eigenem X-Wing geprüft.

## • HOPs mit Fold:

- Geschosse und gegnerischen Flugzeuge werden auf das Spiel-Bild gefaltet (gemalt).
- Bewegungen der Flugzeuge und Geschosse werden durch Faltung realisiert:
  - Verlassen sie das Spielfeld, so werden sie aus der neuen Liste gelöscht – sonst mit neuer Position wieder angefügt.
- Kollisionen werden durch Faltung überprüft und gefiltert.





# VIELEN DANK FÜR DIE AUFMERKSAMKEIT