Grundlegendes zu Listen

- Listen können in Racket verschiedene Elemente unterschiedlichen Typs enthalten

```
(define list 1(list 2 5 0 -4 3 5)) // akzeptiert beliebig viele Parameter
und liefert Liste aus den Parametern

(define list2 (cons 7 list 1))
// → (list 7 2 5 0 -4 3 6)

(define list3 (cons 2 empty)) → empty ist leere liste
```

- `cons` -> Funktion, die als Parameter (element, liste) hat. Gibt eine neue Liste zurück mit dem element an erster stelle und den Elementen der ursprünglichen Liste
- `(define list3 (cons 2 empty))` -> empty ist leere liste,
  in **Java** ähnlich null in ListItems
- `(first list1)` -> erstes Element der Nichtleeren Liste,
  in **Java** wie Key bei ListItems
- `(rest list2)`-> nichtleere Liste ohne Element->
  in **Java** wie next

Rekursive Funktionen auf Listen

- Listen werden rekursiv bearbeitet
- Beschreibung deutscher Sprache ist oft leicht in Racket umzusetzen

Beispiel: Summe einer Liste

```
(define (sum lst))
(if (empty? lst)
0
(+(first lst) (sum(rest lst))))
```

- Rekursive Möglichkeit, die Summe einer Liste zu addieren

- rekursionsanker, falls die lst 0 ist wird 0 zurückgegeben
- andernfalls wird der **Key** bzw das erste Element addiert und sum(lst rest) bzw **next** aufgerufen.

Gefilterte Liste

- Leer, falls lst empty ist
- Restliste wird gefiltert, falls das erste Element filter nicht passiert
- Sonst erst erstes Element gefolgt von der Filterung der Restliste

```
(define (less-than-only lst x)
(if (empty? lst)
empty

(if(<(first lst) x)
(cons(first lst)(less-than-only (rest lst) x))
(less-than-only (rest lst) x))))
```

- Falls liste leer ist-> null
- Filter: `(if(<(first lst) x)` -> wenn erstes Element kleiner als x ist
- -> wenn ja wird : `(cons (first lst) (less-than-only (rest lst) x))` aufgerufen->erstes Element wird der neuen liste hinzugefügt, danach rekursiver Aufruf mit rest
- -> wenn nein, dann wird nur die Rekursion ausgerufen mit dem Rest der Liste

Rackets Objektmodell

- eine Liste ist eine Folge von Werten, nicht von Objekten, da es keine Objekte gibt
- Gefilterte Liste = Kopien von Werten

Cond

- vereinfacht ineinandergeschachtelte if- Anweisungen

```
(define (less-than-only lst x)
(cond
[(empty?lst) empty]
[(< (first lst) x)
(cons (first lst) (less-than-only (rest lst) x))]))
```

```
//Restfall
[ else (less-than-only (rest lst))x]
```

- Wenn Restfall nicht vorhanden ist und das ende von cond erreicht wird--> Fehler

---

Structs

```
(define-struct student
(last-name firstname enrollment-number)
(define alf-tanner(make-student 'make-student  'Tanner  'Als 123))
```

- Struct wird definiert-> in dem Fall student
- Namen der Attribute werden aufgezählt-> last name first name...
- `make-"Name des Structs` -> sowas wie der "Konstruktor"
- Parameter werden in der gleichen Reihenfolge der Attribute gesetzt

```
(define (last-name studi)  //gibt das Attribut von last-name zurück
(student-last name stude)  //<Name des Structs>—<Name des Attributs>
(student? x))              // prüfen, ob x vom Typ Students ist
```

- Also sowas wie selbst erstellte "Typen"

Akkumulatoren

- Im grunde genommen variablen, um Ergebnisse schrittweise zwischenzuspeichern

```
(define (sum-list lst acc)
(if (empty? lst)
acc
(sum-list (cdr lst)
(+ acc(car lst))))
)
```

```
bsp;
(sum-list '(1 2 3 4 5) 0) → Gibt 15 zurück (1 + 2 + 3 + 4 + 5 = 15)
```