# Grundlagen funktionaler Programmierung

http://www.drscheme.org/

## Erinnerung an den letzten Termin:

- Lambda-Ausdrücke (anonyme Funktionen)
  ((lambda (x) (* x x)) 5)

- Funktionsdef. mit Namen
  (define (quadr x) (* x x))

- Prädikate and, or, not
- Bedingungen if, when, unless, cond, case
- Sequenzen begin, let, let*

## Iteration und Schleifen

Oft ist es nötig, eine Reihe von Befehlen mehrmals zu wiederholen. Dazu verwendet man so genannte Schleifen. Die meistens Sprachen verwenden dafür Konstrukte wie for, while, do … until. In Scheme verwendet man eine Konstruktion mit dem **Do-Befehl**.

```
(do ((<variable1> <anfangswert1> <schritt1>)
       ...
     (<variablen> <anfangswertn> <schrittn>))

   (<abbruchbed> {<ausdr1> ... <ausdrp>})

   {<anweisung1>
      ...
    <anweisungm>})
```

Auswertung:

- Bindung der Variablen an ihre Anfangswerte
- Auswertung der Abbruchbedingung. Liefert diese den Wert „falsch“, so werden die Anweisungen des Körpers der Funktion ausgeführt.
- Anschliessend Auswertung der angegebenen Schrittausdrücke, Bindung der Ergebnisse an die Variablen, Auswertung der Abbruchbedingung, etc.
- Dieser Zyklus wiederholt sich, bis die Abbruchbedingung den Wert „wahr“ liefert. Erst jetzt Auswertung der nachstehenden Ausdrücke.

## Beispielprogramm zu Schleifen

```
(do ((counter 1 (+ counter 1)))
    ((> counter 10)(display "fertig")
                   (newline))
    (display counter)
    (display " ... " ))
```

liefert

```
1 ... 2 ... 3 ... 4 ... 5 ... 6 ... 7 ... 8 ...
9 ... 10 ... fertig
```

# Rekursion als Strategie zum Problemlösen

Wie in logischen Sprachen lässt sich auch in funktionalen Sprachen die Rekursion besonders elegant umsetzen.

- **Lineare Rekursion**: ein einmaliger rekursiver Aufruf pro Rekursionsschritt

- Bei **Baumrekursion** mindestens zwei rekursive Aufrufe pro Rekursionsschritt

Zunächst Beschränkung auf lineare Rekursion ...

Definitionen:

- Eine **echtrekursive Funktion** ist dadurch gekennzeichnet, dass die Berechnung des Funktionswertes erst nach Erreichen der Abbruchbedingung, also während des rekursiven Aufstiegs aus der Rekursionstiefe erfolgt.

- Eine **endrekursive Funktion** ist dadurch gekennzeichnet, dass die Berechnung des Funktionswertes bereits während des rekursiven Abstiegs erfolgt und bei Erreichen der Abbruchbedingung somit vorliegt.

Beispiel: Fakultät

n! = 1 für n=0,
= n* (n-1)! für n>=1

Direkte Umsetzung in Scheme - welche Art von Funktion liegt hier vor?

```
(define (fak_rek n)
    (cond ((zero? n) 1)
          (else (* n (fak_rek (- n 1))))))
```

Evaluierung dieser echtrekursiven Funktion

➔ siehe Skizze Tafel für 3!

- Rekursiver Abstieg bis zum Erreichen der Abbruchbedingung
- Schrittweise Berechnung während des rekursiven Aufstiegs

Aufgabe: Im debugger den Ablauf und die Bindungen der Variablen n nachvollziehen.

In einigen Fällen ist es relativ einfach möglich, auch eine endrekursive Funktion zur Lösung des gleichen Problems anzugeben. Dazu ist es notwendig, ein zusätzliches Argument (**akkumulierender Parameter**) hinzuzufügen. Aufruf der endrekursiven Funktion von einer Rahmenfunktion, die einen Anfangswert an den akkumulierenden Parameter bindet:

```
(define (fak_end n) (fak_h n 1))

(define (fak_h i j)
  (cond ((zero? i) j)
        (else (fak_h (- i 1) (* i j)))))
```

Evaluierung dieser endrekursiven Funktion

➔ siehe Skizze Tafel für 3!

**Anmerkung:**

- (echte) Rekursion ist für funktionale Sprachen die typische Form, Wiederholungen zu realisieren
- Beseitigung der echten Rekursion häufig schwierig
- Auch möglich: Nutzung von Schleifen (mit „do“, siehe oben)

## Geht auch: **Iterative Lösung zur Berechnung von n!**

```
(define (fak_it n)
  (do ((i 1 (+ i 1))
       (erg 1 (* erg i)))
    ((> i n) erg)))
```

- Leerer Körper der Funktion, keine Anweisungen
- Rückgabe von erg nach Erreichen der Endbedingung

Bitte mit Scheme-Debugger testen !!!!

```
(fak_it 5):

i   1 2 3 4  5  6
erg 1 1 2 6 24 120  → 120
```

## Baumrekursion

Mindestens zwei rekursive Aufrufe pro Rekursionsschritt sind nötig (genau 2 Aufrufe entsprechen einem binären Baum)

Beispiel: **Fibonacci-Zahlen** (bereits bekannt als Beispiel dafür, dass Rekursion schlechthin ineffizient ist).

Zur Erinerung:

f(0) = 0
f(1) = 1
f(n) = f(n-1) + f(n-2)  für n>1

In DrScheme:

```
(define (fib_rek n)
    (cond [(= 0 n) 0]
          [(= 1 n) 1]
          [else (+ (fib_rek (- n 1))
                   (fib_rek (- n 2)))] ))

(fib_rek 9)
(fib_rek 33)
(fib_rek 44)
```

Sehr ineffizient, da für grosse Werte von n viele Fibonacci-Zahlen mehrfach berechnet werden müssen, wie der Berechnungsbaum für `(fib_rek 5)` zeigt

➔ siehe Tafel

Die bekannte Lösung des Problems bei Prolog: Werte bereits berechneter f(n) speichern (mit assert !!) und bei Bedarf wieder verwenden, statt sie neu zu berechnen ...

Andere Möglichkeit: Endrekursiver (hier nicht) oder iterativer Ansatz:

Funktionsdefinition mittels do:

```
(define (fib_iter n)
  (do ((i 1 (+ i 1))
       (a 1 (+ a b))
       (b 0  a))
    ((= i n) a)))
```

Sehr effizient !!

Aufgabe: Füllen Sie die folgende Tabelle

| i | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|
| a | ? | ? | ? | ? | ? | ? | ? |
| b | ? | ? | ? | ? | ? | ? | ? |

| i | 1 | 2 | 3 | 4 | 5 | 6 | 7 | … |
|---|---|---|---|---|---|---|---|---|
| a | 1 | 1 | 2 | 3 | 5 | 8 | 13 | … |
| b | 0 | 1 | 1 | 2 | 3 | 5 | 8 | … |

## Einfache Probleme der Listenverarbeitung

Erinnerung: bereits bekannte Funktionen cons, car, cdr, quote und eval, symbolische Darstellung, …

Sehr viele vordefinierte Grundfunktionen, u.a.:

| | |
|---|---|
| `(atom? <objekt>)` | Objekt ein Atom ? |
| `(pair? <objekt>)` | Objekt ein Paar ? |
| `(list? <objekt>)` | Objekt eine Liste ? |
| `(null? <objekt>)` | Objekt eine leere Liste? |

```
(require (lib "compat.ss"))  ; für atom? benötigt
```

> (atom? (car '(1)))
> (atom? (cdr '(1)))
> (atom? (cdr '(1 2)))

ergibt :
#t, #t, #f

> (pair? '(1 . 2))
> (pair? '(1))
> (pair? '((2) 1))
> (pair? '())

ergibt :
#t, #t, #t, #f

> (list? '(1 2))
> (list? '(1 . (1)))
> (list? '(1 . 3))

ergibt :
#t, #t, #f

> (null? '())
> (null? (cdr '(1)))
> (null? '2)
> (null? '(2))

ergibt :
#t, #t, #f, #f

## Aufgaben:

1. Definieren Sie eine Scheme-Prozedur, die (rekursiv) die Anzahl der Elemente einer einfach verketteten Liste berechnet (Hinweis: (null? ...) prüft, ob eine Liste leer ist):

   (anzElemente '(1 3 4 7 2 5)) -> 6.

2. Definieren Sie eine Scheme-Prozedur, die die Summe aller Listenelemente berechnet:

   (sumElemente '(1 3 4 7 2 5)) -> 22

Eine Lösung:

```
(define (anzElemente liste)
  (cond
    [(null? liste) 0]
    [else (+ 1 (anzElemente (cdr liste)))] ))

(anzElemente '(1 3 4 7 2 5))
```

```
(define (sumElemente liste)
  (cond
    [(null? liste) 0]
    [else (+ (car liste)
             (sumElemente (cdr liste)))] ))

(sumElemente '(1 3 4 7 2 5))
```

3. Modifizieren Sie Ihr Programm aus Aufgabe 1 dahingehend, dass es die Atome eines Baumes zählt (Hinweis: (atom? ...) prüft, ob auf ein Atom gezeigt wird.):

   (anzBlaetter '(1 (3 4) (7 2 5)) ) -> 6.

```
; Prozedur, die die Anzahl der Atome eines Baumes
zurückgibt:

(define (anzBlaetter liste)
  (cond
    [(null? liste) 0]
    [(atom? liste) 1]
    [else (+ (anzBlaetter (car liste))
             (anzBlaetter (cdr liste)))] ))

(anzBlaetter '(1 (3 4) (7 2 5)) )
```

4. Definieren Sie eine Funktion zum Anfügen eines Elements an eine einfach verkettete Liste

**Also:** `(anfuegen 3 '(1 2))`
liefert `(1 2 3)`

```
(define (anfuegen el liste)
    (cond ((null? liste) (cons el ()))
          (else (cons (car liste)
             (anfuegen el (cdr liste))))))
```

5. Definieren Sie eine Funktion zur Umkehrung der Reihenfolge der Elemente einer einfach verketteten Liste

**Also:** `(umkehr '(1 2 3 4 5))`
liefert
`(5 4 3 2 1)`

```
(define (umkehr liste)
  (cond ((null? liste) liste)
        ((null? (cdr liste)) liste)
        (else (anfuegen (car liste)
                        (umkehr (cdr liste))))))
```

Hinweis: In DrSchemeDrRacket existiert die Grundfunktion reverse. Diese Funktion arbeitet auf einer Kopie der Ausgangsliste

## Arbeit mit Bäumen

Hier nur binäre Bäume (wichtig in der Informatik)

Binäre Bäume werden in Scheme als geschachtelte Listen dargestellt, wobei vor jeder Wurzel eines Teilbaumes eine Klammer geöffnet wird.

Beispiel:

```
(define biba '(8 (4 (2 (1) (3))
                    (6 (5) (7)))
                 (12 (10 (9) (11))
                     (14 (13) (15)))))
```

Skizze (siehe Tafel)

Es gibt 3 unterschiedliche Arten des Baumdurchlaufs, z.B. den **Praeorder-Durchlauf**: Wurzel – linker Teilbaum – rechter Teilbaum.

```
(define (praeorder baum)
  (cond ((null? (cdr baum))
           (display (car baum)) (display " "))
        (else (display (car baum)) (display " ")
              (praeorder (cadr baum))
              (praeorder (caddr baum)))))
```

Ausgabe:

```
(praeorder biba)
8 4 2 1 3 6 5 7 12 10 9 11 14 13 15
```

**Frage:** Wie muss man die Funktion ändern, um einen Inorder- und einen Postorder-Durchlauf zu erhalten

# Weiteres Interessantes

Nutzen Sie die Trace-Funktion, um die Abarbeitung der Beispiele besser nachvollziehen zu können.

- Laden Sie die Trace-Library mit (require (lib “trace.ss“))
- Aktivieren Sie die Trace-Funktion für den jeweiligen Test mit (trace <Fkt_Name>)
- Die Protokollfunktion kann mittels (untrace <Fkt_Name>)