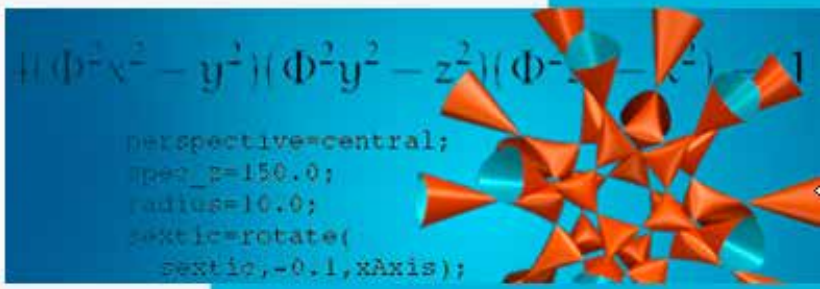
Philipps Universität Marburg

Deklarative Programmierung

Sommersemester 2018

Prof. Christoph Bockisch
(Programmiersprachen und –werkzeuge)
Steffen Dick, Alexander Bille, Johannes Frankenau,
Patrick Frömel, Niclas Schmidt, Jonas Stettin,
Robert Tran, Julian Velten



[Skript 15]

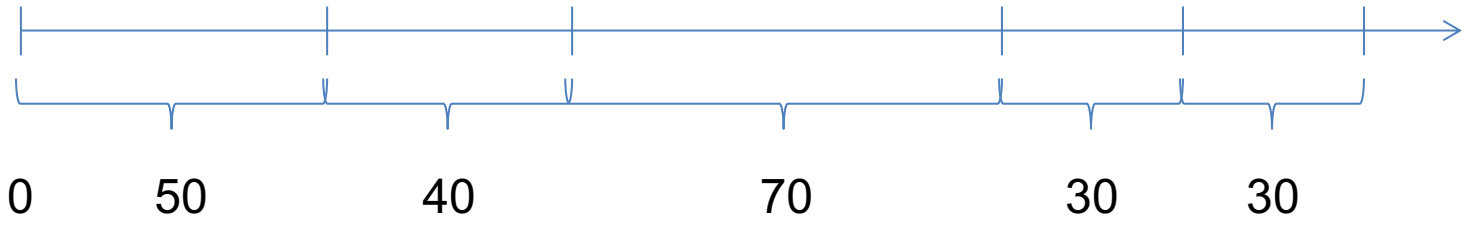
Funktionale Programmierung

- Funktionale Programmierung
  - Funktionen berechnen Wert ausschließlich basierend auf Funktionsparametern
  - Keine Seiteneffekte
  - Kontext-unabhängig

- Einige Probleme erfordern jedoch Bezug auf Kontext

Kontext-abhängige Funktionen – Beispiel 1

- Gegeben: Liste mit Distanzen zwischen Punkten



- Gesucht: Liste mit absoluten Distanzen

Kontext-abhängige Funktionen – Beispiel 1

```
; (list-of Number) -> (list-of Number)
; converts a list of relative distances to a list of absolute distances
(check-expect (relative-2-absolute (list 0 50 40 70 30 30))
    (list 0 50 90 160 190 220))
(define (relative-2-absolute alon)  ...)
```

Wie füllen wir die Schablone aus?

Kontext-abhängige Funktionen – Beispiel 1

```
; (list-of Number) -> (list-of Number)
; converts a list of relative distances to a list of absolute distances
(check-expect (relative-2-absolute (list 0 50 40 70 30 30))
    (list 0 50 90 160 190 220))
(define (relative-2-absolute a
    (match alon
        [empty  ...]
        [(cons x xs)  ... x  ... (relative-2-absolute xs)  ...]))
```

Wie können wir das Ergebnis aus (relative-2-absolute xs) berechnen?

Strukturelle Rekursion

Kontext-abhängige Funktionen – Beispiel 1

- Schwierigkeit:
  - Problem:
    - Pro Position berechneter Wert hängt ab von Listenanfang
  - Strukturelle Rekursion:
    - Berechnung basierend auf Teilergebnis für Listen-Rest
- Lösung: Schrittweise Aufbau des Ergebnisses

Kontext-abhängige Funktionen – Beispiel 1

```
; (list-of Number) -> (list-of Number)
; converts a list of relative distances to a list of absolute distances
(check-expect (relative-2-absolute (list 0 50 40 70 30 30))
    (list 0 50 90 160 190 220))
(define (relative-2-absolute alon)
    (match alon
        [empty empty ]
        [(cons x xs) (cons x
            (map
                (lambda (y) (+ y x))
                (relative-2-absolute xs)))]))
```

map erzeugt eine neue Liste aus den Ergebnissen des Lambda-Ausdrucks für jedes Element aus der Originalliste.

Kontext-abhängige Funktionen – Beispiel 1

```
(define (relative-2
   (match alon
      [empty empty ]
      [(cons x xs) (cor
         (map
            (lambda (y (+ y
            (relative-2-absolute xs)))]))
```

x: 0
xs: (list 50 40 70)

x: 50
xs: (list 40 70)

x: 40
xs: (list 70)

x: 70
xs: empty

```
> (relative-2-absolute (list 0 50 40 70))
```

Kontext-abhängige Funktionen – Beispiel 1

```
(define (relative-2
   (match alon
      [empty empty ]
      [(cons x xs) (cor
         (map
            (lambda (y (+ y
            (relative-2-absolute xs))
```

x: 0
xs: (list 50 40 70)

x: 50
xs: (list 40 70)

x: 40
xs: (list 70)

```
(cons 70 (map
 (lambda (y) (+ y 70))
 (relative-2-absolute empty)))
→ (list 70)
```

```
> (relative-2-absolute (list 0 50 40 70))
```

Kontext-abhängige Funktionen – Beispiel 1

```
(define (relative-2
   (match alon
      [empty empty ]
      [(cons x xs) (cor
         (map
            (lambda (y (+ y
            (relative-2-absolute xs)))]))
```

x: 0
xs: (list 50 40 70)

x: 50
xs: (list 40 70)

```
(cons 40 (map
 (lambda (y) (+ y 40))
 (relative-2-absolute (list 70))))
→ (list 40 110)
```

```
> (relative-2-absolute (list 0 50 40 70))
```

Kontext-abhängige Funktionen – Beispiel 1

```
(define (relative-2
    (match alon
        [empty empty ]
        [(cons x xs) (cor
            (map
                (lambda (y (+
                (relative-2-absolute xs)))]))
```

x: 0
xs: (list 50 40 70)

```
(cons 50 (map
 (lambda (y) (+ y 50))
 (relative-2-absolute
   (list 40 110))))
→ (list 50 90 160)
```

```
> (relative-2-absolute (list 0 50 40 70))
```

Kontext-abhängige Funktionen – Beispiel 1

```
(define (relative-2
   (match alon
      [empty empt
      [(cons x xs) (
         (map
            (lambda (y)    y x))
            (relative-2-absolute xs)))]))
```

```
(cons 0 (map
 (lambda (y) (+ y 0))
 (relative-2-absolute
   (list 50 90 160))))
→ (list 0 50 90 160)
```

```
> (relative-2-absolute (list 0 50 40 70))
(list 0 50 90 160)
```

Beispiel 1 Effizienz

- Pro rekursionsschritt “mappen“ der gesamten bisher berechneten Liste
- Daher: Anzahl Berechnungsschritte in Größenordnung von n^2 bei n Elementen in der Liste
- Wie viele Berechnungsschritte bei Berechnung von Hand?
  - n - 1 Additionen
  - Einmal von links nach rechts über List
  - Merken der bisherigen Summe

Achtung: Zustand! Den gibt es bei funktionaler Programmierung nicht! Können wir den simulieren?

Akkumulation

- Können wir die intuitive Berechnungsform auch mit funktionaler Programmierung durchführen?

- Gegeben zwei verschiedene Listen:
  (cons x1 xs) und (cons x2 xs)
  - Rekursiver Aufruf jeweils (f xs)
  - Rekursiver Aufruf kann nicht von Beginn der Liste (x1 bzw. x2) abhängen
  - Lösung: f benötigt zusätzlichen Parameter

Akkumulation

- Zusätzlicher Parameter
  - Akkumulator
  - Zustand der Berechnung
  - Enthält bisheriges Wissen
- Fragen zum Akkumulator
  - Wie wird das bisherige Wissen berechnet?
  - Wie wird aus dem Akkumulator der Folgezustand berechnet?
  - Welcher Wert ist der Ausgangspunkt?

Akkumulation – Beispiel 1

- Umrechnung: relative zu absoluten Entfernungen
- Berechnung des bisherigen Wissens
  - Summe aller bisherigen relativen Entfernungen
- Folgezustand
  - Summe von Akkumulator und aktueller relativer Entfernung
- Ausgangspunkt
  - Vor erster relativer Entfernung: 0

Akkumulation – Beispiel 1

```
; (list-of Number) Number -> (list-of Number)
(define (relative-2-absolute-with-acc alon accu-dist)
   (match alon
          [empty empty]
          [(cons x xs)
           (local [(define x-absolute (+ accu-dist x))]
              (cons x-absolute
                    (relative-2-absolute-with-acc xs
                                                  xabsolute)))]))
```

Akk - Beispiel 1

Veränderte Signatur

```
; (list-of Number) Number -> (list-of Number)
(define (relative-2-absolute-with-acc alon accu-dist)
  (match alon
    [empty em
    [(cons x xs)
     (local [(define x-absolute (+ accu-dist x))]
       (cons x-absolute
             (relative-2-absolute-with-acc xs
                                           xabsolute)))]))
```

Folgezustand

Akkumulierte absolute Entfernung

Verwendung für Berechnung des Ergebnisses

Verwendung bei rekursivem Aufruf

Akkumulation – Beispiel 1

```
; (list-of Number) -> (list-of Number)
; converts a list of relative distances to a list of absolute distances
(check-expect (relative-2-absolute-2 (list 0 50 40 70 30 30))
   (list 0 50 90 160 190 220))
(define (relative-2-absolute-2 alon)
   (local
      ; (list-of Number) Number -> (list-of Number)
      [(define (relative-2-absolute-with-acc alon accu-dist)
         (match alon
            [empty empty]
            [(cons x xs)
               (local [( define x-absolute (+ accu-dist x))]
               (cons x-absolute
                  (relative-2-absolute-with-acc xs xabsolute)))]))]
   (relative-2-absolute-with-acc alon 0)))
```

Akkumulatic

Ursprüngliche Signatur ohne Akkumulator

```
; (list-of Number) -> (list-of Number)
; converts a list of relative distances to a list of absolute distances
(check-expect (relative-2-absolute-2 (list 0 50 40 70 30 30))
    (list 0 50 90 160 190 220))
(define (relative-2-absolute-2 alon)
    (local
        ; (list-of Number) Number -> (list-of Number)
        [(define (relative-2-absolute-with-acc alon accu-dist)
            (match alon
                [empty empty]
                [(cons x xs)
                    (local [( define x-absolute (+
                    (cons x-absolute
                        (relative-2-absolute-with
                    (relative-2-absolute-with-acc alon 0)))
```

Funktion mit Akkumulator als lokale Definition

Aufruf der lokalen Funktion mit initialem Wert für Akkumulator

Beispiel 1 Effizienz

- Einmaliges Durchlaufen der Liste
- Pro Rekursionsschritt
  - Berechnung des Akkumulators
  - Eine Addition
- Daher: Anzahl Berechnungsschritte in Größenordnung von n bei n Elementen in der Liste

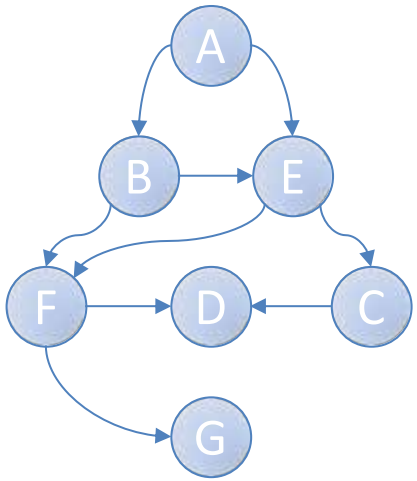
Beispiel 2

- Gegeben ein gerichteter Graph
  - Menge von Knoten
  - Menge von gerichteten Kanten zwischen jeweils zwei Knoten
- Suche in Graphen
- Aufgabe: Finden einer Route zwischen zwei Knoten

Repräsentation eines Graphen

- Liste von Paaren, die jeweils folgende Information enthalten:
  1. Knoten-Name (Symbol)
  2. Liste der durch eine Kante erreichbaren Knoten

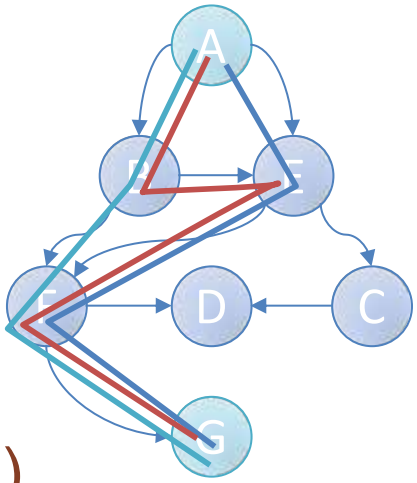
```
; A Node is a symbol
; A Node-with-neighbors is a (list Node (list-of Node))
; A Graph is a (list-of Node-with-neighbors)

(define graph1
   '((A (B E))
     (B (E F))
     (C (D))
     (D ())
     (E (C F))
     (F (D G))
     (G ())))
```



Suche in Graphen

```
; Node Node Graph -> (list-of Node) or false
; to create a path from origination to destination in G
; if there is no path, the function produces false
(check-member-of
   (find-route 'A 'G graph1)
      '(A E F G)
      '(A B E F G)
      '(A B F G))
(define (find-route origination destination G)  ...)
```

Ergebnis muss eine der drei gültigen Lösungen sein

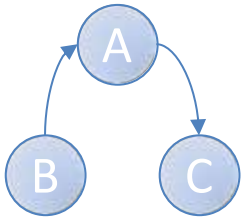


Suche in Graphen mittels struktureller Rekursion

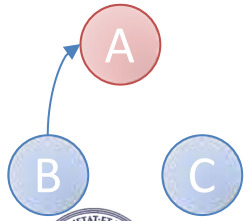
- Nicht möglich

- Beispiel

```
(define graph2
  '((A (C))
    (B (A))
    (C ())))
```



- Suche Route zwischen A und C mittels struktureller Rekursion unmöglich
  - Zweiter Rekursionsschritt würde in Teilgraph '((B (A)) (C ())) suchen, in dem relevante Kanten fehlen



Suche in Graphen mittels generativer Rekursion

- Entwurfsrezept

1. Triviales Problem: Route von einem Knoten zu sich selbst
2. Triviale Lösung: (list n)
3. Generierung neues Problem: Für jeden Nachfolger: gibt es eine Route vom Nachbarknoten zum Endknoten?
4. Berechnen der Lösung: Bei Nachbarn mit Lösung: Liste mit aktuellem Knoten zusammengefügt mit Ergebnis der Rekursion
5. Terminierung: wird später betrachtet

Suche in Graphen mittels generativer Rekursion

```
; Node Node Graph -> (list-of Node) or false
; to create a path from origination to destination in G
; if there is no path, the function produces false
(check-member-of (find-route 'A 'G graph1) '(A E F G)
   '(A B E F G) '(A B F G))
(define (find-route origination destination G)
   (cond
      [(symbol=? origination destination) (list destination)]
      [else ( local (( define possible-route
            (find-route/list (neighbors origination G) destination G)))
         (cond
            [(boolean? possible-route) false]
            [else (cons origination possible-route)]))]))
```

Suche in Graphen mittels generativer Rekursion

```
; Node Node Graph -> (list-of Node) or false
; to create a path from origination to destination in G
; if there is no path, the function produces false
(check-member-of (find-route 'A 'G graph1) '(A E F G)
   '(A B E F G) '(A B E G))
(define (find-ro                    tinatio
   (cond
      [(symbol=? origination destination) (list destination)]
      [else ( local (( define possible-route
                (find-route/list (neighbors origination G) destination G)))
                                    an? possible-route)
                [else (cons origination possible-route)]))]))
```

Triviales Problem

Triviale Lösung

Berechnung Ergebnis

Generierung neues Problem. Rekursiver Aufruf in Hilfsfunktion

Suche in Graphen mittels generativer Rekursion

```
; Node Graph -> (list-of Node)
; computes the set of neighbors of node n in graph g
(check-expect (neighbors 'B graph1) '(E F))
(define (neighbors n g)
  (match g
    [(cons (list m m-neighbors) rest)
     (if (symbol=? m n)
         m-neighbors
         (neighbors n rest))]
    [empty (error "node not found")]))
```

Strukturelle Rekursion

Knoten mit Nachbarn

m ist gesuchter Knoten

Rekursionsabbruch

Rekursiver Fall

Basisfall

Suche in Graphen mittels generativer Rekursion

```
; (list-of Node) Node Graph -> (list-of Node) or false
; to create a path from some node on lo-Os to D
; if there is no path, the function produces false
(check-member-of (find-route/list '(E F) 'G graph1)
    '(F G) '(E F G))
(define (find-route/list lon D G)
    (match lon
        [empty false]
        [(cons n ns)
         (local ((define possible-route (find-route n D G)))
           (cond
              [(boolean? possible-route) (find-route/list ns D G)]
              [else possible-route]))]))
```

Strukturelle Rekursion

Basisfall

Strukturrekursion

Transitiv rekursiver Aufruf der Hauptfunktion

Rekursiver Fall

Rekursionsabbruch

Backtracking Algorithmus

```
(define (find-route origination destination G)
  (cond
    [(symbol=? or
    [else ( local ((
          (find-route/list (neighbors origination G) destination G)))
        (cond
          [(boolean? possible-route) false]
          [else (cons origination possible-route)]))]))
(define (find-route/list lon D G)
  (match lon
    [empty false]
    [(cons n ns)
     (local ((define possible-route (find-route n D G)))
       (cond
         [(boolean? possible-route) (find-route/list ns D G)]
         [else possible-route]))]))
```

Backtracking: Systematisches Ausprobieren von Alternativen

Probiere für aktuellen Nachbarn Route zu finden

Ansonsten probiere nächsten Nachbarn

Bei Erfolg: Ergebnis

Terminierung

- Kommt der Algorithmus mit jedem Schritt dem Ziel näher?
- Ziel:
  - Zielknoten erreicht
  - Keine verbleibenden Alternativen
- Finden einer Abbildung der Funktionsargumente auf (maximale) Anzahl der verbleibenden Rekursionsschritte

Terminierung

- Alternativen in find-route/list
  - Alle Nachbarn des aktuellen Knotens
  - Alle Nachbarn der Nachbarn
- Unendliche Anzahl an Alternativen bei zyklischen Graphen

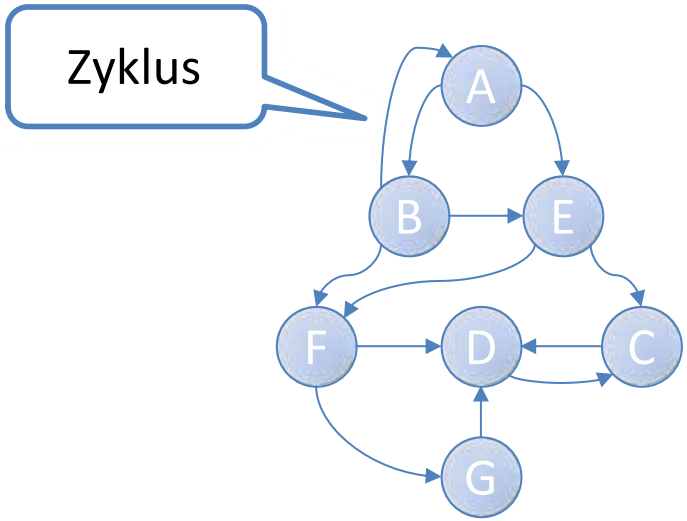
Terminierung

```
(define graph3
    '((A (B E))
      (B (A E F))
      (C (D))
      (D (C))
      (E (C F))
      (F (D G))
      (G (D))))
>(find-route 'A 'G graph3)
```

Führt zu (find-route/list '(B E) 'G graph3)

Führt zu (find-route 'A 'G graph3)

Etc.



Terminierung

- Zyklische Graphen enthalten Routen von **unendlicher** Länge

- Daher terminiert der Algorithmus evtl. nicht
  - Falls Rekursion entlang einer unendlichen Route stattfindet

Lösungsansatz

- Zyklische Graphen enthalten auch Routen **endlicher** Länge
  - find-route sucht nur ob **irgendeine** Route existiert
  - Es reicht, nach dem kürzesten Pfad zu suchen
  - Der kürzeste Pfad enthält jeden Knoten nur ein Mal
- Mitprotokollieren der bereits besuchten Knoten in Akkumulator visited
- find-route und find-route/list sind “wechselseitig rekursiv“
  - Daher muss der Akkumulationsparameter zu beiden Funktionen hinzugefügt werden

Akkumulator visited

```
(define (find-route origination destination G visited)
  (cond
    [(symbol=? origination destination) (list destination)]
    [else ( local (( define possible-route
                     (find-route/list (neighbors origination G) destination G
                       (cons origination visited)    )))
                  (cond
                    [(boolean? possible-route) false]
                    [else (cons origination possible-route)]))]))
  (match lon
    [empty false]
    [(cons n ns)
     (local ((define
       (cond
         [(boolean
         [else possible-route]))]))
```

Folgezustand

Durchreichen des Akkumulators

Durchreichen des Akkumulators

Lösungsansatz

- Wie kann der Akkumulator für die Berechnung genutzt werden?

- Ausschließen von Knoten die bereits besucht wurden

Akkumulator visited

```
(define (find-route origination destination G visited)
   (cond
      [(symbol=? origination destination) (list destination)]
      [else ( local (( define possible-route
            (find-route/list
               (remove-all visited (neighbors origination G))
               destination
               G
               (cons origination visited)    )))
         (cond
            [(boolean? possible-route) false]
            [else (cons origination possible-route)]))]))
```

Terminierung

1. Größe der Eingabe: Abbildung der Argumente auf natürliche Zahl
   - Gegeben ein Graph mit n Knoten und die Liste visited mit m Knoten
   - Größe der Eingabe: n - m (Anzahl der Knoten die noch nicht besucht wurden

- n - m ist stets positiv
  - Verwendung nur der Nachbarn, die noch nicht besucht wurden
    (remove-all visited (neighbors origination G))
  - Daher: nur hinzufügen von Knoten zu visited die noch nicht besucht wurden
    (cons origination visited)
  - Visited ist eine Teil**menge** der Knoten von G

Terminierung

- Wechselseitige Rekursion von find-route und find-route/list
  - find-route/list ist strukturell rekursiv (terminiert daher immer)
  - find-route/list übergibt G und visited immer unverändert an find-route

2. Daher: durch Hinzufügen eines Knoten zu visited wird n - m immer strikt kleiner

- Rekursionstiefe ist maximal die Anzahl der Knoten im Graphen

Entwurf von Funktionen mit Akkumulator

- Anwendung von Akkumulatoren nur, wenn
  - Bisherige Entwurfsrezepte scheitern
  - Alternativlösung ist zu kompliziert oder langsam

- Schlüsselaktivitäten
  1. Erkennen, dass ein Akkumulator nötig/sinnvoll ist
  2. Verstehen, was der Akkumulator repräsentiert

Wann braucht man einen Akkumulator

- Gegeben eine strukturell rekursive Funktion, die eine rekursive Hilfsfunktion aufruft
  - Oft kann ein Akkumulator verschachtelte Rekursion ersetzen
  - Dadurch meist lineare statt quadratischer Laufzeit
- Beispiel ohne Akkumulator

```
; invert : (listof X) -> (listof X)
; to construct the reverse of alox
; structural recursion
(define (invert alox)
  (cond
    [(empty? alox) empty]
    [else (make-last-item
     (first alox) (invert (rest alox)))]))
```

Rekursive Hilfsfunktion

```
; make-last-item : X (listof X) -> (listof X)
; to add an-x to the end of alox
; structural recursion
(define (make-last-item an-x alox)
  (cond
    [(empty? alox) (list an-x)]
    [else (cons (first alox)
      (make-last-item an-x (rest alox)))]))
```

Wann braucht man einen Akkumulator

- Benötigte Information
  - Ist nicht lokal verfügbar
  - Aber kann im Laufe der rekursiven Aufrufe gesammelt werden
- Beispiel:
  - Mitprotokollieren der besuchten Knoten
  - Notwendig für die Terminierung der Suche in zyklischen Graphen

Template für Funktionen mit Akkumulator

```
; invert : (listof X) -> (listof X)
; to construct the reverse of alox
(define (invert alox0)
   (local (; accumulator ...
      [define (rev alox accumulator)
         (cond
            [(empty? alox)  ...]
            [else
               ... (rev (rest alox)  ... (first alox)  ... accumulator)
               ...])])
      (rev alox0  ...)))
```

Template für Funktionen mit Akkumulator

```
; invert : (listof X) -> (listof X)
; to construct the reverse of alox
(define (invert alox0)
  (local (; accumulator ...
          [define (rev alox accum
            (cond
              [(empty? alox)  ...]
              [else
                ... (rev (rest alox)  ... (first alox)  ... accumulator)
                ...])])
    (rev alox0  ...)))
```

Lokale Definition der Funktion mit Akkumulator

Rekursion mit neuem Problem

Berechnung neuer Akkumulator aus aktuellem Problem und ursprünglichem Akkumulator

Aufruf der lokalen Funktion mit initialem Akkumulator

Akkumulator Invariante

- Was repräsentiert der Akkumulator in jedem Rekursionsschritt?
- Welche Daten werden akkumuliert?

- Beispiel: Funktion invert
  - Akkumulation der bisher besuchten Listenelemente
  - In umgekehrter Reihenfolge

Dokumentation der Akkumulator-Invarianten im Code

```
; invert : (listof X) -> (listof X)
; to construct the reverse of alox
(define (invert alox0)
   (local (; accumulator is the reversed list of all those items
           ; on alox0 that precede alox
           (define (rev alox accumulator)
              (cond
                 [(empty? alox)  ...]
                 [else
                    ... (rev (rest alox)  ... ( first alox)  ... accumulator)
                     ...])))
    (rev alox0  ...)))
```

Anwenden der Akkumulator-Invarianten

- Akkumulator-Invariante muss eingehalten werden
  1. Was ist der initiale Wert?
  2. Wie bleibt die Invariante beim rekursiven Aufruf erhalten?

- Beispiel
  1. Vor dem 1. Aufruf wurden noch keine Listenelemente besucht. Der initiale Akkumulator ist also die leere Liste.
  2. Beim Rekursionsschritt muss das aktuelle Element vorne an den Akkumulator angehängt werden
     (cons (first alox) accumulator)

Nutzung des Akkumulators

- Implementierung der Akkumulator-Invarianten an sich reicht noch nicht aus:
  - Wie kann die Implementierung der Funktionalität den Akkumulator nutzen?

- Beispiel:
  - Ist das Ende der Liste erreicht, enthält der Akkumulator das Ergebnis

Wann braucht man einen Akkumulator

```
; invert : (listof X) -> (listof X)
; to construct the reverse of alox
(define (invert alox0)
  (local (; accumulator is the reversed list of all those items
          ; on alox0 that precede alox
          (define (rev alox accumulator)
            (cond
              [(empty? alox) accumulator]
              [else
                (rev (rest alox) (cons (first alox) accumulator))])))
    (rev alox0 empty)))
```