Declarative programming

Summer semester 2024

Prof. Christoph Bockisch, Steffen Dick
(Programming languages and tools)

Imke Gürtler, Daniel Hinkelmann, Aaron Schafberg, Stefan Störmer

[Script 15]

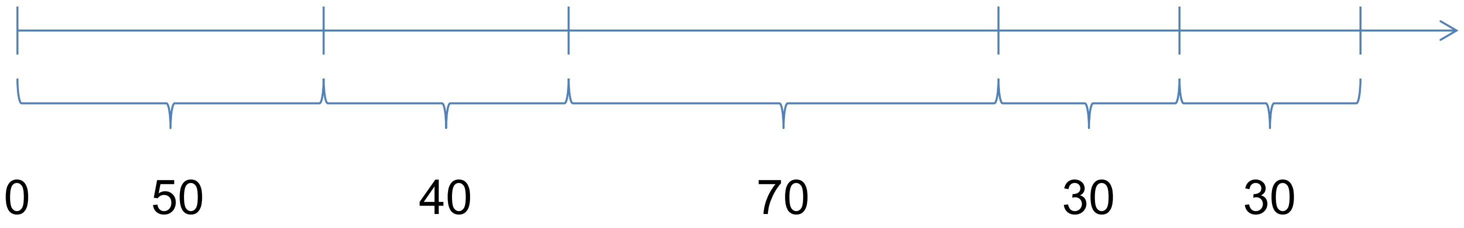
Functional programming

- Functional programming
  - Functions calculate value based solely on function parameters
  - No side effects
  - Context-independent

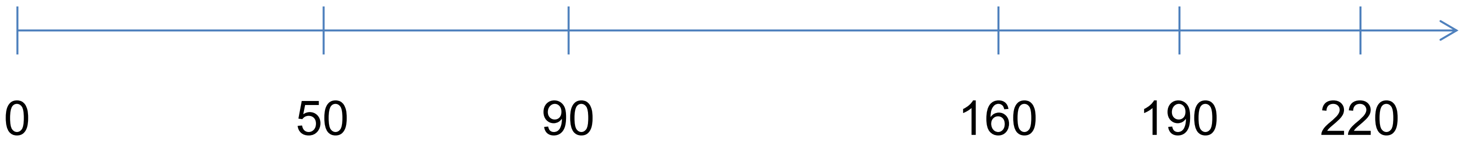
- However, some problems require reference to context

Context-dependent functions - Example 1

- Given: List of distances between points



- Wanted: List with absolute distances



Context-dependent functions - Example 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) ...)
```

How do we fill in the template?

Context-dependent functions - Example 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
  (cond
    [(empty? alon) ...]
    [else ... (first alon) ... (relative-2-absolute (rest alon)) ...]))
```

How can we calculate the result from (relative-2-absolute (rest alon)) calculate?

Structural recursion

Context-dependent functions - Example 1

- Difficulty:
  - Problem:
    - Value calculated per position depends on start of list
  - Structural recursion:
    - Calculation based on partial result for list remainder

- Solution: Step-by-step structure of the result

Context-dependent functions - Example 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)
  (cond
    [(empty? alon) empty]
    [else (cons (first alon)
      (map
        (lambda (y) (+ y (first alon)))
        (relative-2-absolute (rest alon))))]))
```

map creates a new list from the results of the lambda expression for each element from the original list.

Context-dependent functions - Example 1

```
(define (relative-2-
 (cond
   [(empty? alon) e
   [else (cons (first          ))
     (map
       (lambda (y) (+ y   irst al    )))
       (relative-2-absolute (rest alon))
```

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))
```

Context-dependent functions - Example 1

```
(define (relative-2-
 (cond
    [(empty? alon) e
    [else (cons (first            ))
       (map
          (lambda (y) (+ y  irst a
          (relative-2-absolute (res

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

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)
```

Context-dependent functions - Example 1

```
(define (relative-2-
 (cond
     [(empty? alon) e
     [else (cons (first        l)
         (map
             (lambda (y  (+ y   rst
             (relative-2-absolute (rest alon))))]))
```

x: 0
xs: (list 50 40 70)

x: 50
xs: (list 40 70)

```
(cons 40 (map
 (lambda (y) (+ y 40))
 (list 70)))
 → (list 40 110)
```

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

Context-dependent functions - Example 1

```
(define (relative-2-
 (cond
     [(empty? alon) e
     [else (cons (first          )
         (map
             (lambda (y  (+  )   (
             (relative-2-absolute (rest alon)))]))
```

x: 0
xs: (list 50 40 70)

```
(cons 50 (map
 (lambda (y) (+ y 50))
  (list 40 110)))
→ (list 50 90 160)
```

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

Context-dependent functions - Example 1

```
(define (relative-2
 (cond
     [(empty? alo
     [else (cons (f
         (map
            (lambda (y)       y (first alon)))
            (relative-2-absolute (rest alon))))]))
```

```
(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)
```

Example 1 Efficiency

- Per recursion step "mapping" of the entire list calculated so far
- Therefore: Number of calculation steps in the order of n^2 for n elements in the list
- How many calculation steps for manual calculation?
  - n - 1 Additions
  - Once from left to right via list
  - Remembering the previous total

Attention: State! This does not exist in functional programming! Can we simulate it?

Accumulation

- Can we also perform the intuitive calculation form with functional programming?

- Given two different lists:
  (cons x1 xs) and (cons x2 xs)
  - Recursive call in each case (f xs)
  - Recursive call cannot depend on the start of the list (x1 or x2)
  - Solution: f requires additional parameter

Accumulation

- Additional parameter
  - Accumulator
  - State of the calculation
  - Contains previous knowledge

- Questions about the accumulator
  - How is previous knowledge calculated?
  - How is the subsequent state calculated from the accumulator?
  - Which value is the starting point?

Accumulation - Example 1

- Conversion: relative to absolute distances
- Calculation of previous knowledge
  - Sum of all previous relative distances
- Subsequent state
  - Sum of accumulator and current relative distance
- Starting point
  - Before first relative distance: 0

Accumulation - Example 1

```
; (list-of Number) Number -> (list-of Number)
(define (relative-2-absolute-with-acc alon accu-dist)
   (cond
            [(empty? alon) empty]
            [else
             (local [(define x-absolute (+ accu-dist (first alon)))]
               (cons x-absolute
                    (relative-2-absolute-with-acc (rest alon)
                                                   x-absolute)))))
```

Acc Example 1

Changed signature

```
; (list-of Number) Number -> (list-of Number)
(define (relative-2-absolute-with-acc alon accu-dist)
   (cond
         [(empty? a
         [else
          (local [(define x-absolute (+ accu-dist (first alon)))]
            (cons x-absolute
                  (relative-2-absolute-with-acc (rest alon)
                                                x-absolute)))))
```

Subsequent state

Accumulated absolute distance

Use for calculation of the result

Use with recursive call

Accumulation - Example 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)
      (cond
        [(empty? alon) empty]
        [else
         (local [(define x-absolute (+ accu-dist (first alon)))]
           (cons x-absolute
                 (relative-2-absolute-with-acc (rest alon)
                                               x-absolute)))])))]
    (relative-2-absolute-with-acc alon 0)))
```


Philipps Universität Marburg

Accumulatio

Original signature without accumulator

```
(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)
         (cond
            [(empty? alon) empty]
            [else
             (local [(define x-absolute (+ ac
                (cons x-absolute
                      (relative-2-absolute-with-         alon)
                                              x-absolute))))))]
      (relative-2-absolute-with-acc alon 0)))
```

Function with accumulator as local definition

Calling the local function with initial value for accumulator

Example 1 Efficiency

- Run through the list once
- Per recursion step
  - Calculation of the accumulator
  - An addition
- Therefore: Number of calculation steps in the order of n for n elements in the list

Example 2

- Given a directed graph
  - Set of nodes
  - Set of directed edges between two nodes

- Search in graphs

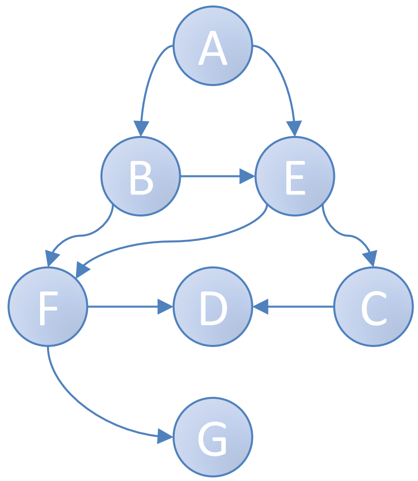
- Task: Finding a route between two nodes

Representation of a graph

- List of pairs, each containing the following information:
  1. Node name (symbol)
  2. List of nodes that can be reached by an edge

```
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 ())))
```



Search in graphs

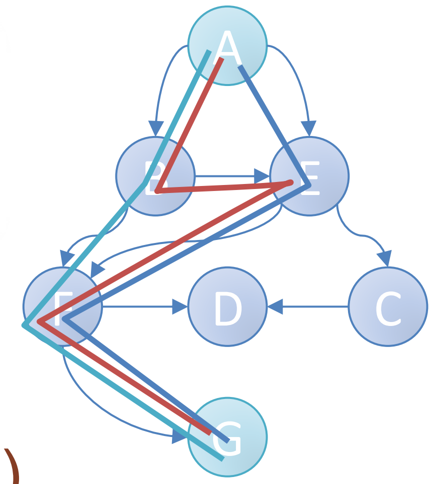
Node Node Graph -> (list-of Node) or false

; 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) ...)
```

The result must be one of the three valid solutions

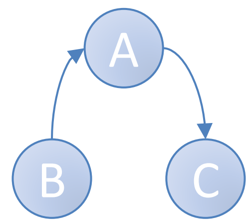


Search in graphs using structural recursion

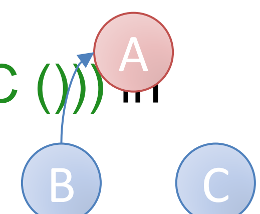
- Not possible

- Example

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



- Search route between A and C using structural recursion impossible
  - Second recursion step would search in subgraph '((B (A)) (C ())) in which relevant edges are missing



Search in graphs using generative recursion

- Design recipe

1. Trivial problem: Route from a node to itself
2. Trivial solution: (list n)
3. Generation of new problem: For each successor: is there a route from the neighboring node to the end node?
4. Calculating the solution: For neighbors with solution: List merged with current node with result of recursion
5. Scheduling: will be considered later

Search in graphs using generative recursion

```
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)]))])))
```

Search in graphs using generative recursion

```
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)]))))))
```

Trivial problem

Trivial solution

Calculation result

Generation of new problem.
Recursive call in auxiliary function

Search in graphs using generative recursion

```
; 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)
  (cond
    [(cons? g)
     (if (symbol=? (first (first g)) n)
        (second (first g))
        (neighbors n (rest g)))]
    [empty (error "node not found")]))
```

Structural recursion

Knots with neighbors

m is searched node

Recursion termination

Recursive case

Base case

Search in graphs using generative recursion

Structural recursion

```
(list-of Node) Node Graph -> (list-of Node) or false
; to create a path in G from some node in lon 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)
  (cond
    [(empty? lon) false]
    [else
     (local [(define possible-route (find-route (first lon) D G))]
       (cond
         [(boolean? possible-route) (find-route/list (rest lon) D G)]
         [else possible-route]))]))
```

Search in graphs using generative recursion

```
(list-of Node) Node Graph -> (list-of Node) or false
; to create a path in G from some node in lon 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)
  (cond
    [(empty? lon) false]
    [else
     (local [(define possible-route (find-route (first lon) D (
       (cond
         [(boolean? possible-route) (find-route/list (rest lon) D G)]
         [else possible-route])))]))
```

Structural recursion

Base case

Structural recursion

Transitive recursive call of the main function

Recursive case

Recursion termination

Backtracking algorithm

```
(define (find-route origination destination G)
  (cond
    [(symbol=? o
    [else ( local (( define possible-route
          (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)
  (cond
    [(empty? lon) false]
    [else
     (local [(define possible-route (find-route (first lon) D G
        (cond
          [(boolean? possible-route) (find-route/list (rest lon) D G)]
          [else possible-route]))))])
```

Backtracking: Systematic testing of alternatives

Try to find route for current neighbors

Otherwise try your nearest neighbor

If successful: Result

Scheduling

- Is the algorithm getting closer to its goal with every step?

- Target:
  - Target node reached
  - No remaining alternatives

- Find a mapping of the function arguments to the (maximum) number of remaining recursion steps

Scheduling

- Alternatives in find-route/list
  - All neighbors of the current node
  - All neighbors of the neighbors

- Infinite number of alternatives for cyclic graphs

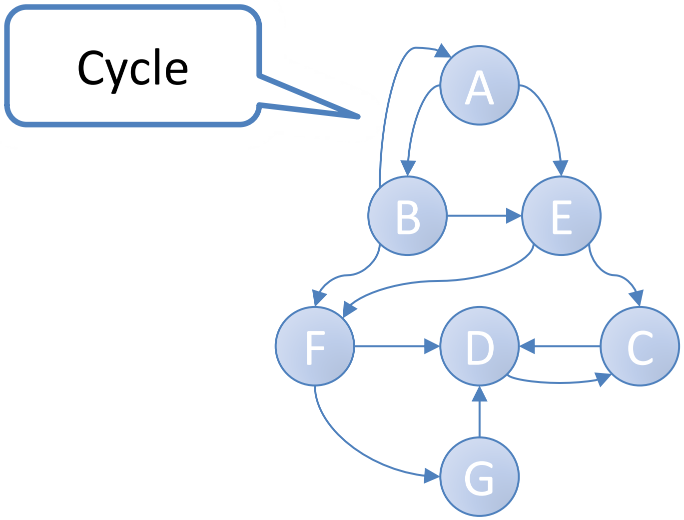
Scheduling

```
(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)
```

Leads to (find-route/list '(B E) 'G graph3)

Leads to (find-route 'A 'G graph3)

Etc.



Scheduling

- Cyclic graphs contain routes of **infinite** length
- Therefore, the algorithm may not terminate
  - If recursion takes place along an infinite route

Solution approach

- Cyclic graphs also contain routes **of finite** length
  - find-route only searches whether **any** route exists
  - It is enough to look for the shortest path
  - The shortest path contains each node only once
- Logging the nodes already visited in the accumulator
- find-route and find-route/list are "mutually recursive"
  - The accumulation parameter must therefore be added to both functions

Accumulator 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) )))
            an? possible-route) false]
            ons origination possible-route)])))))
(define (find-route/list lon D G visited)
  (cond
    [(empty? lon) false]
    [else
     (local ((define possib                              G visite
       (cond
         [(boolean? poss                          st lon) D G visited)]
         [else possible-route])))))
```

Subsequent state

Passing through the accumulator

Passing through the accumulator

Solution approach

- How can the accumulator be used for the calculation?
- Exclude nodes that have already been visited

Accumulator 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)]))))))
```

Scheduling

1. Size of the input: Mapping of the arguments to natural numbers
   - Given a graph with n nodes and the list visited with m nodes
   - Input size: n - m (number of nodes that have not yet been visited)

- n - m is always positive
  - Use only the neighbors that have not yet been visited (remove-all visited (neighbors origination G))
  - Therefore: only add nodes to visited that have not yet been visited (cons origination visited)
  - Visited is a subset of the nodes of G

Scheduling

- Mutual recursion of find-route and find-route/list
  - find-route/list is structurally recursive (therefore always terminates)
  - find-route/list always passes G and visited unchanged to find-route

2. Therefore: by adding a node to visited
   n - m is always strictly smaller

- Recursion depth is the maximum number of nodes in the graph

Design of functions with accumulator

- Use of accumulators only if
  - Previous design recipes fail, or if
  - Alternative solution is too complicated or too slow

- Key activities
  1. Recognize that an accumulator is necessary/useful
  2. Understanding what the accumulator represents

When do you need an accumulator

- Given a structurally recursive function that calls a recursive auxiliary function
  - An accumulator can often replace nested recursion
  - Therefore mostly linear instead of quadratic runtime
- Example without accumulator

```
; 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)))]))
```

Recursive auxiliary function

```
; 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)))]))
```

When do you need an accumulator

- Required information
  - Is not available locally
  - But can be collected in the course of recursive calls

- Example:
  - Logging the nodes visited
  - Necessary for scheduling the search in cyclic graphs

Template for functions with accumulator

```
; 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 ...)))
```


Philipps Universität Marburg

Template for functions with accumulator

```
; 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 ...)))
```

Local definition of the function with accumulator

Calculation of new accumulator from current problem and original accumulator

Recursion with new problem

Calling the local function with initial accumulator

Accumulator invariant

- What does the accumulator represent in each recursion step?
- What data is accumulated?

- Example: Invert function
  - Accumulation of previously visited list elements
  - In reverse order

Documentation of the accumulator variants in the 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 ...)))
```

Applying the accumulator variants

- Accumulator variant must be adhered to
  1. What is the initial value?
  2. How is the invariant retained in the recursive call?

- Example
  1. No list elements were visited before the 1st call. The initial accumulator is therefore the empty list.
  2. In the recursion step, the current element must be appended to the front of the accumulator
     `(cons (first alox) accumulator)`

Use of the accumulator

- Implementation of the accumulator variants is not enough in itself:
  - How can the implementation of the functionality use the accumulator?

- Example:
  - When the end of the list is reached, the accumulator contains the result

When do you need an accumulator

```
; 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)))
```