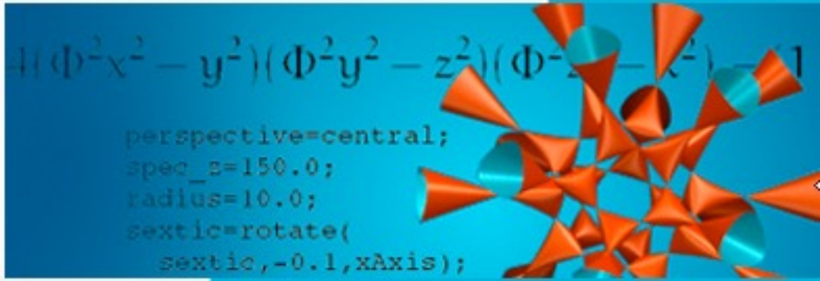
Philipps Universität Marburg

Declarative programming

Summer semester 2024

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

Imke Gürtler, Daniel Hinkelmann, Aaron Schafberg, Stefan Störmer



[Script 9.3 - 9.6]

Lists

- Previously: binary trees as a recursive data structure
- Binary tree: each node has two successors
- Trees can also have other numbers of successors
- Special case:
  - Each node has a successor
  - Tree "degenerates" into a list
- I.e.: Lists are also recursive data structures

Lists as a recursive data structure

```
(define-struct lst (first rest))
A List-of-Numbers is either:
; - (make-lst Number List-Of-Numbers)
; - false
; interp. the head and rest of a list, or the empty list

(make-lst 1 (make-lst 2 (make-lst 3 false)))
```

Functions via lists

- Functional design analogous to trees

```
List-Of-Numbers -> Number
; adds up all numbers in a list
(check-expect (sum (make-lst 1 (make-lst 2
                                (make-lst 3 false)))) 6)
(define (sum l) ...)
```

What does the template look like?

Functions via lists

- Functional design analogous to trees

```
List-Of-Numbers -> Number
; adds up all numbers in a list
(check-expect (sum (make-lst 1 (make-lst 2
                                        (                   )) 6)

(define (sum l)
   (cond [(lst? l) ... (lst-first l) ... (sum (lst-rest l)) ...]
         [else ...]))
```

Selector non-recursive partial data

Selector recursive partial data

Case: Recursion

Case: Recursion termination

Recursive call

Functions via lists

- Functional design analogous to trees

```
List-Of-Numbers -> Number
; adds up all numbers in a list
(check-expect (su                2
                            st 3 false)))) 6)

(define (sum l)
  (cond [(lst? l) ... (lst-first l) ... (sum (lst-rest l)) ...]
        [else ...]))
```

What happens with the result of the recursive calls?

What is the "standard result"?

Philipps Universität Marburg

Functions via lists

- Functional design analogous to trees

```
List-Of-Numbers -> Number
; adds up all numbers in a list
(check-expect (sum (make-lst 1 (make-lst 2
                                            (make-lst 3 false)))) 6)
(define (sum l)
   (cond [(lst? l) (+ (lst-first l) (sum (lst-rest l)))]
              [else 0]))
```

Lists built into BSL

- Constructor function: cons
- Constant for empty list: empty

N.B.: Definition of constants as "markers" (only meaning: marking of a certain intention)

Example:

```
(define-struct empty-lst ())
(define EMPTYLIST (make-empty-lst))
(make-lst 1 (make-lst 2 (make-lst 3 EMPTYLIST)))
```

Lists built into BSL

- The operation cons corresponds to our make-lst
  - Additional benefit: BSL checks that the second argument is a list

```
> (cons 1 2)
cons: second argument must be a list, but received 1 and 2
```

- We can also write such a review ourselves

```
(define (our-cons x l)
   (if (or (empty-lst? l) (lst? l))
      (make-lst x l)
      (error "second argument of our-cons must be a list")))
```

Lists built into BSL

Built-in list	Self-defined list	Meaning
empty or '()	EMPTYLIST	Value for empty list
empty?	empty-lst?	Test whether a value is the empty list
cons	our-cons	Constructor function
first	lst-first	function returns the first element of the list
rest	lst-rest	Function returns list without the first element
cons?	is?	Test whether a value is a list

Lists built into BSL

Hiding definitions is an important tool for information hiding. So far, we do not know of any way in BSL to hide definitions ourselves.

- BSL hides definition of structure for lists
  - Lists must be generated via cons function
  - Additional test that second argument is a list
  - Direct access would bypass test

Lists built into BSL

- Example of using built-in lists

```
A List-of-Numbers is one of:
; - (cons Number List-Of-Numbers)
; - empty
List-Of-Numbers -> Number
; adds up all numbers in a list
(check-expect (sum (cons 1 (cons 2 (cons 3 empty)))) 6)
(define (sum l)
  (cond [(cons? l) (+ (first l) (sum (rest l)))]
        [else 0]))
```

Syntactic sugar for lists

- Create lists
  - Nested calls of cons and empty
  - Elaborate/ complex
- Syntactic sugar: list function

(list exp-1 ... exp-n)

Transformation

(cons exp-1 (cons ... (cons exp-n empty)))

Philipps Universität Marburg

Syntactic sugar for lists

- Example
  - (list 1 2 3) corresponds:
  - (cons 1 (cons 2 (cons 3 empty)))

- Language levels in Racket:
  - Syntactic sugar "list" may be used in all language levels in the program
  - Language levels differ in the presentation of the program output:
    - "Beginners with list abbreviations"
      - List values are also displayed in abbreviated form
    - "Beginner"
      - List values are represented by nested cons calls

Philipps Universität Marburg

Lists for specific data types?

- In the example: Definition of the List-Of-Numbers data type

- Do we need a list structure for each data type?

- No:
  - The structure is always the same (Don't Repeat Yourself!)
  - Many functions are independent of the element data type
    - second, third, fourth, etc.
  - Reuse of these functions with a common list structure

Philipps Universität Marburg

Homogeneous lists

- Lists where all elements have the same type are called "homogeneous"
  - Type can be a sum type, for example, i.e. have alternatives

- Based on general list structure
  - Data definition for list with type restriction
  - Restriction through "type parameters"

```
; A (List-of X) is one of:
; - (cons X (List-of X))
; - empty
```

  - Any type can be used for X, e.g: (List-of String), (List-of Boolean), (List-of (List-of String)) or (List-of FamilyTree)

Lists with type variables

- "Type variables" for functions
  - Functions that can be used for several types
  - With dependencies between types of parameters and result
  - Declaration of type parameter: [X] at the beginning of the signature
  - X can now be used in the signature

```
X] (List of X) -> X
(define (second l) ...)
```

- Type variables can be replaced by any type, e.g:
  (List-of Number) → Number
  (List-of (List-of String)) → (List-of String)

Philipps Universität Marburg

Why lists as a recursive data type?

- Extension of the language to support lists is also conceivable
- Consequences
  - Representation closer to the hardware
  - Explicit language constructs for processing lists
  - Lists have special status
- Language extension not necessary in BSL
  - Definition and processing of lists with on-board tools (recursive data type)
  - No restriction on the use of lists
- For further argumentation, see script 9.3.5

Philipps Universität Marburg

Natural numbers as a recursive data structure

- Natural numbers can also be represented as a recursive data structure
  - (This applies to all "countable" value sets)
  - Intuition:
    - Lists are recursive data structures
    - The number n can be represented by a list with n elements
  - However, the element does not matter
    - We therefore do not save any values in the list
    - But string empty cells together

A Nat (Natural Number) is one of:

; - 0

; - (add1 Nat)

- Example: Representation of the number 3 as (add1 (add1 (add1 0)))

Philipps Universität Marburg

Natural numbers as a recursive data structure

List	Nat
empty	0
empty?	zero?
cons	add1
first	-/-
rest	sub1
cons?	positive?

Natural numbers as a recursive data structure

- Why do we want to see natural numbers as a recursive data type?
  - So we can write well-defined recursive functions over natural numbers
  - These can process or produce recursive data structures with the same structure.
  - Example: Creating a list with n entries

Philipps Universität Marburg

Natural numbers as a recursive data structure

X] Nat X -> (List of X)

; creates a list with n occurences of x

```
(check-expect (iterate-value 3 "abc") (list "abc" "abc" "abc"))
(define (iterate-value n x) ...)
```

What should the stencil look like?

Natural numbers as a recursive data structure

```
X] Nat X -> (List of X)
; creates a list with n occurences of x
(check-expect (iterate-value 3 "abc") (list "abc" "abc" "abc"))
(define (iterate-value n x)
   (cond [(zero? n) ...]
         [(positive? n) ... (iterate-value (sub1 n) ...)...]))
```

Case: Recursion termination

Case: Recursion

Recursive call for partial data

There is no non-recursive partial data. (Only the completely non-recursive case.)

Philipps Universität Marburg

Natural numbers as a recursive data structure

```
X] Nat X -> (List of X)
; creates a list with n occurences of x
(check-expect (iterate-value 3 "abc") (list "abc" "abc" "abc"))
(define (iterate-value n x)
  (cond [(zero? n) empty]
        [(positive? n) (cons x (iterate-value (sub1 n) x))]))
```


Philipps Universität Marburg

Functions over several recursive data types

- Example:
  - append
  - Merge two lists

```
; [X] (list-of X) (list-of X) -> (list-of X)
; concatenates l1 and l2
(check-expect (lst-append (list 1 2) (list 3 4)) (list 1 2 3 4))
(define (lst-append l1 l2) ... )
```

Which argument do we use for our design recipe?

Philipps Universität Marburg

Functions over several recursive data types

- Example:
  - append
  - Merge two lists

```
; [X] (list-of X) (list-of X) -> (list-of X)
; concatenates l1 and l2
(check-expect (lst-append (list 1 2) (list 3 4)) (list 1 2 3 4))
(define (lst-append l1 l2)
   (cond [(empty? l1) ...l2...]
         [(cons? l1) ... (first l1) ... (lst-append (rest l1) ...)]))
```

For example: Preferring the first argument.

Philipps Universität Marburg

Functions over several recursive data types

- Example:
  - append
  - Merge two lists

```
; [X] (list-of X) (list-of X) -> (list-of X)
; concatenates l1 and l2
(check-expect (lst-append (list 1 2) (list 3 4)) (list 1 2 3 4))
(define (lst-append l1 l2)
   (cond [(empty? l1) l2]
            [(cons? l1) (cons (first l1) (lst-append (rest l1) l2))]))
```

Leads to the goal in this case.

Philipps Universität Marburg

Functions over several recursive data types

- In general
  - Selection of the argument for the recursion "creative process"
  - Depending on how the information can be meaningfully decomposed

Design recipe with recursive data types

- 1. information representation
- Information of unlimited size must be represented via recursive data type
- Conditions:
  1. The data type is a sum type
  2. There must be at least two alternatives
  3. At least one of the alternatives is not recursive
- In any case: Specification of data examples
  - Used, among other things, to check the correct recursion termination

Design recipe with recursive data types

- 3. tests
- Data size is unlimited
- A test per alternative is therefore impossible
- Test cases for example for
  - Alternative with and without recursion
  - Further case distinctions

Design recipe with recursive data types

- 4. stencil
- Recursive data types are algebraic data types
- Essentially corresponds to template for algebraic data types
- Additions:
  - At least one alternative must be recursive
  - In this case: self-call instead of help function
  - As argument for recursive call: Selector for extracting the value from the data recursion

Design recipe with recursive data types

- 5. implement function body
- Start with the base cases (recursion termination)
- For recursion: We assume that the recursive function call calculates the result correctly

Design recipe with recursive data types

- 7. post-processing
- Are there data types that are not defined recursively, but can possibly be simplified by a recursive data type?
- E.G.:
  - Are there data types with the same structure?
  - Hierarchies can typically be modeled by recursion

Refactoring of recursive data types

- Recursive data types are algebraic data types
- "Calculation rules" also applicable here
- Helps
  - Recognize isomorphism
  - Simplify data types

Abbreviation for lists

- > (cons 1 (cons 2 (cons 3 empty)))
- (list 1 2 3)

- > (list 1 2 3)
- (list 1 2 3)

"Quote"

- > '(1 2 3)
- (list 1 2 3)

Philipps Universität Marburg

Quote

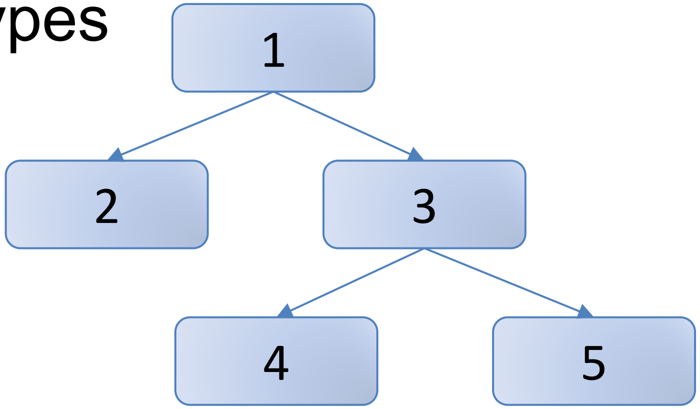
```
> '("a" "b" "c")
(list "a" "b" "c")

> '(5 "xx")
(list 5 "xx")
```

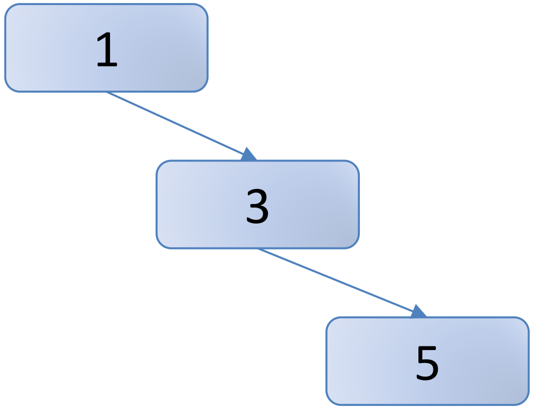

Philipps Universität Marburg

Trees as lists

- Previously: Trees as recursive data types
  - Each node
    - One element (value)
    - Several successors

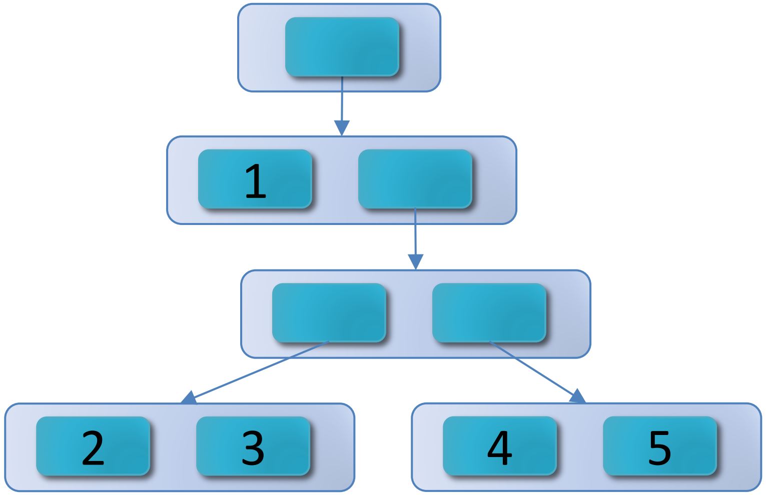


- Lists as recursive data types
  - Degenerated trees
  - Each node
    - One element (value)
    - A successor (possibly empty)



Nested lists

- What happens if the list element a list is used as a list element?
- Nested lists are Trees
    - E.g: Element either Value or list
        - Values in the sheets
- N.B.: Alternative
    - As an element pair of value and list
        - Values also in inner nodes
        - Empty list if there is no successor

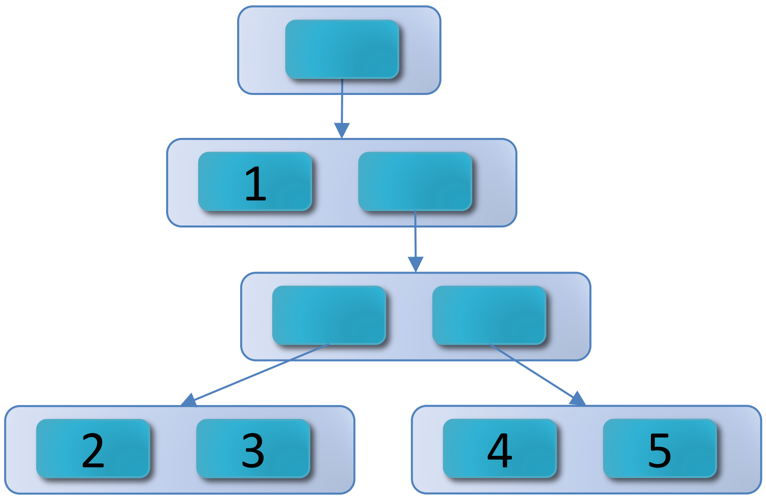


Nested lists

> '(1 ((2 3) (4 5)))

(list 1 (list (list 2 3) (list 4 5)))

Tree structure



> '(("a" 1) ("b" 2) ("c" 3))

(list (list "a" 1)
      (list "b" 2)
      (list "c" 3))

Table structure