Principles of Programming Languages

II.1 Functional Programming Languages

Dr. Herbert Prähofer
Institute for System Software
Johannes Kepler University Linz

Functional Programming Paradigm

Functional programming (FP) programming with **mathematical functions**

```
f: X → Y,        y = f(x)
```

- with property

```
x1 = x2   ⇒    f(x1) = f(x2)
```

Result only depends on values of arguments → No side effects!!

- Functional programs exclusively consist of
  - **Function definitions** including
    - recursive functions
    - higher-order functions
  - **Function application** and **function composition**
  - **Value domains**

```
fact(1) = 1
fact(n) = n * fact(n – 1)
```

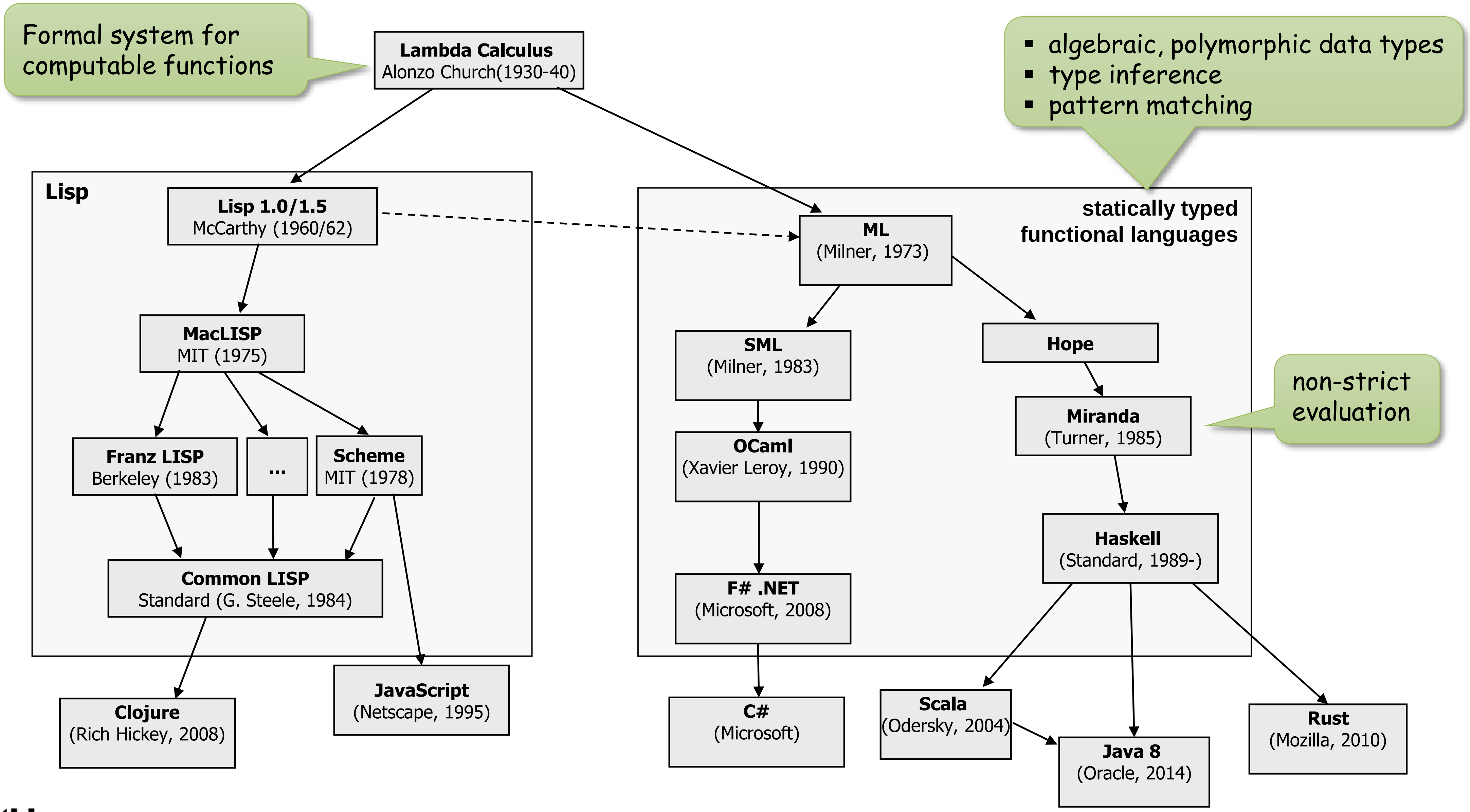
```
map(f, []) = []
map(f, l)  = f(first(l)) :: map(f, rest(l))
```

```
(g ∘ f)(x) = g ( f (x))
```

- No side effects:
  - **NO mutable** variables, **NO mutable** data structures
  - **NO assignments**
  - **NO pointers** and **references, pure value semantics**
  - **NO statements** but only **function application expressions**

Pure functional programming, e.g., Haskell

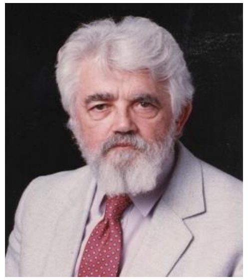
History of functional languages

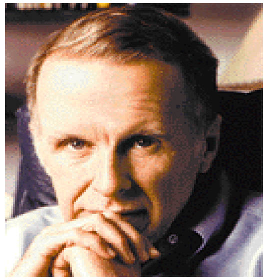


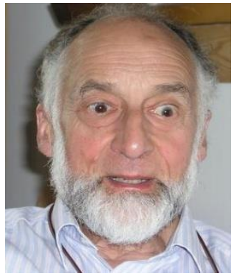
Miles stones in the development of functional languages

1930-40:	Alonzo Church	Lambda Calculus	Mathematical foundation
1960/62	John McCarthy	Lisp	First functional language (not pure), list processing
197x	John Backus	FP	Higher-order functions
197x	Robin Milner	ML	Polymorphic types, type inference
198x	David Turner	Miranda	Lazy evaluation
1987	S. Peyton-Jones,	Haskell	Lazy evaluation, type classes
	P. Wadler, P. Hudak, J. Hughes, et al.		Glasgow Haskell Compiler
2010	Haskell Committee	Haskell 2010	New standard released


Alonzo Church

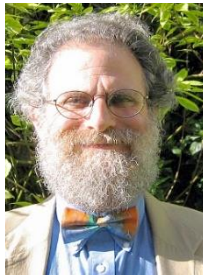

John McCarthy


John Backus


Robin Milner


David Turner

from the Haskell Committee:


Phil Wadler


S. Peyton-Jones


Paul Hudak


John Hughes

... *and many more*

Functional Programming Model

From Lambda Calculus

- Function objects
- Function application
- Higher-order functions
- Strict and non-strict execution semantics

Functional data structures

- Lists
- Algebraic data types
- Polymorphic data types (Generics)

II.1.a Lisp

Lisp

Lisp = List Processing

- Developed in the **late 1950s** / **early 1960s** by **John McCarthy**
- Language for **symbol computation**
- Implementation of **lambda calculus**
  - with **strict evaluation semantics**
- **Dynamically typed**
  - variables, parameters and memory cells have no type
  - values carry type
- Functions as **first class objects**
  - function literals (**lambdas**)
  - **higher-order functions**
- Pairs and **lists**
  - complex data structures built up from lists
  - expressions (= programs) represented as lists

Scheme

- Lisp language with clear and simple semantics
- Developed at **MIT** as an **educational language**
- Characteristics
  - **Lexical scoping**
  - **Tail recursion** optimization
  - **Closures** (first complete implementation)

From Lambda Calculus to Lisp

Lambda Calculus

- Lambda abstraction

```
λ Arguments  .  Lambda-expr
```

```
λ x y . + x y
```

- Function application

```
Lambda-expr  Lambda-expr …
```

```
+ 2 3
```

```
+ (* x 2) (* y  3)
```

```
(λ x  y   .   +  x  y)  2  3
```

Lisp

See rounded brackets!

```
(lambda (<argument-list>) <expr>)
```

```
(lambda (x y) (+ x y))
```

Prefix notation in brackets

```
(fn expr … )
```

```
(+ 2 3)
```

```
(+ (* x 2) (* y 3))
```

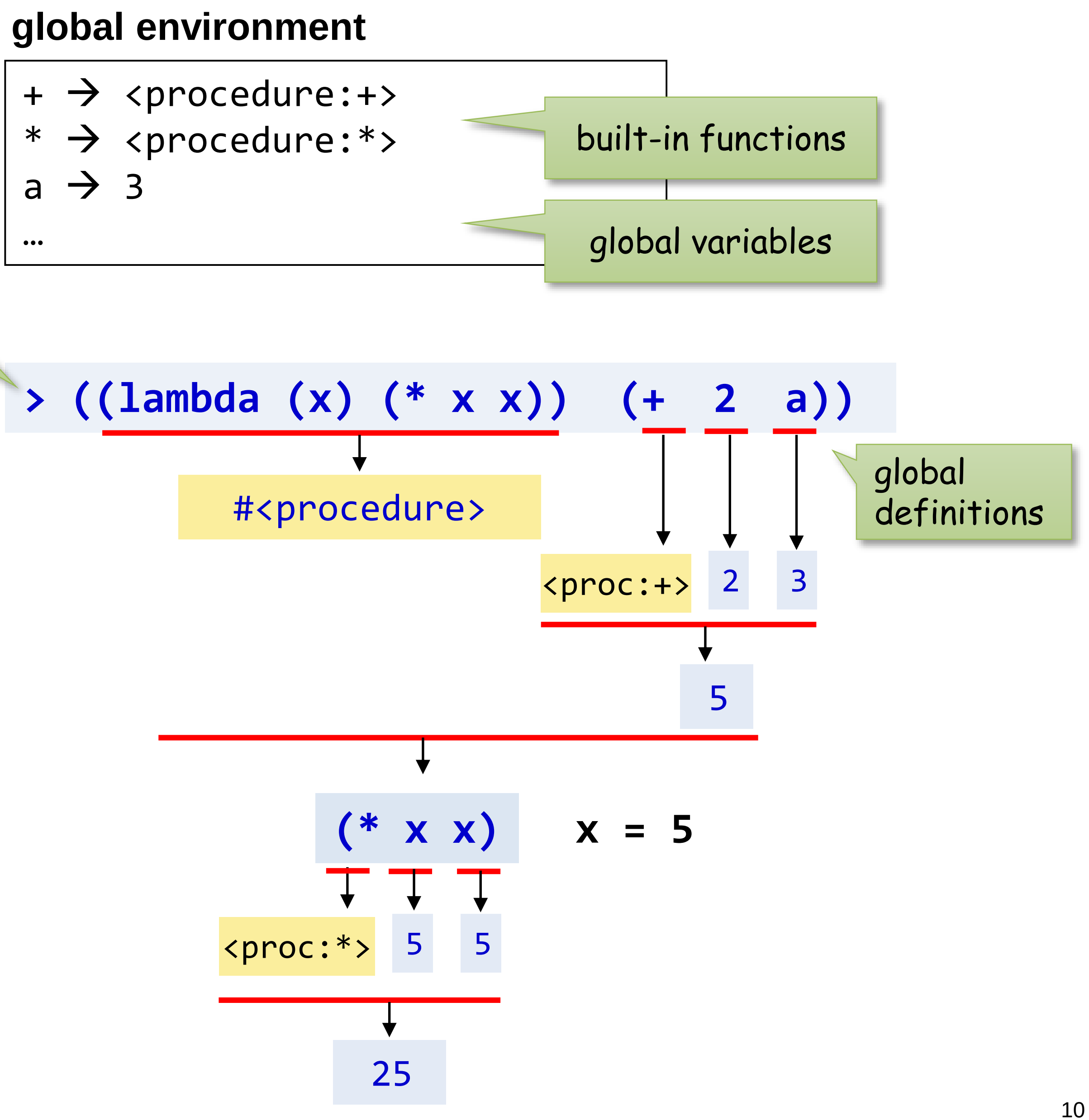
```
((lambda (x y) (+ x y)) 2 3)
```

Evaluation of Function Applications

Strict evaluation

- given definitions in global environment
- evaluate expression by interpreter
- first element in expression is evaluated to function
- rest is evaluated to argument values
- function call with parameter passing: argument values bound to function arguments
- evaluation of body

REPL: interact with interpreter



Special forms

- Special forms are NOT evaluated as function calls
- But each special form has its own rule of evaluation (implemented in interpreter)

- Important special forms are:
  - **`define`** - global definitions
  - **`if`**, **`cond`** - conditional expressions
  - **`lambda`** - creates a function object
  - **`quote`** - protection from evaluation
  - **`set!`** - assignment
  - **`let`** - block with local definitions
  - … a few more …

assignment ➔ Lisp is not purely functional!

Define Special Form

```
(define <symbol> <expression>)
```

- Establishes a binding of names to data objects in global environment
  - evaluate ***<expression>***
  - bind result value to ***<symbol>*** in global environment

Example:

```
> (define pi 3.14159)
> pi
3.14159
```

global environment

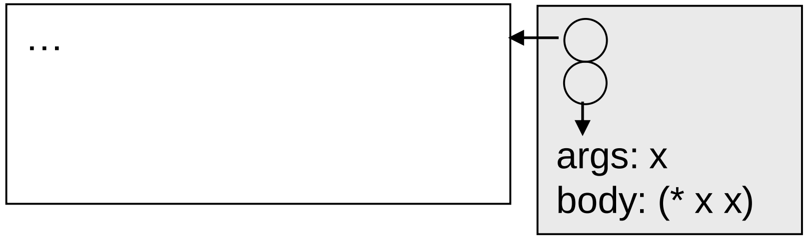
```
…
pi → 3.14159
```

Function Objects and Function Definitions

- **Lambda expressions** create **function objects** (named *procedures*)
  - Function objects are ***first class***, i.e., are like other data objects

```
> (lambda (x) (* x x))
#<procedure>
```

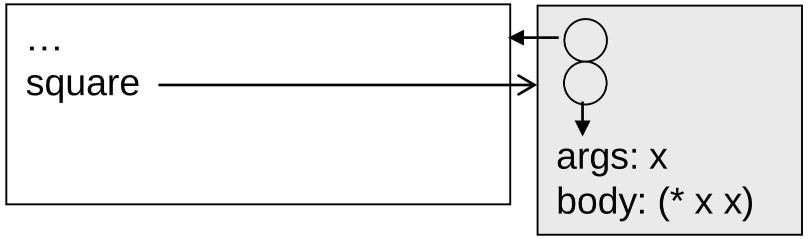
global environment



- **Function definitions** by binding symbol to function object using **define**

```
> (define square (lambda (x) (* x x)))
> (square 5)
25
```

global environment



- Alternative notation (usually used, creates ***named*** function):

```
> (define (square x) (* x x)))
> square
#<procedure:square>
> (square 5)
25
```

If Special Form

```
(if <predicate-expr>
      <then-expr>
      <else-expr>)
```

- Conditional expression with then and else branch
  - evaluate **<*predicate-expr*>** (has to evaluate to a Boolean value)
  - if **true** then evaluate **<*then-expr*>** and return result of <*then-expr*>
  - *else evaluate* **<*else-expr*>** and return result of <*else-expr*>

(!) If is expression and returns a value

Example: Recursive function fac

```
(define (fac n)
      (if (= n 1)
          1
          (* n  (fac (- n 1)))))
```

Equality operators:
= for comparing numbers
eq? for comparing all other values

Cond Special Form

```
(cond (<test-1> <expr-1>)
      (<test-2> <expr-2>)
      …
     [(else <else-expr>)] )
```

- Conditional expression with multiple tests

  From top to bottom:

  - Evaluate tests ***<test-i>*** (has to evaluate to a Boolean value)
  - if ***<test-i>*** evaluates to **true** then evaluate ***<expr-i>*** and return result
  - if all tests fail *evaluate* ***<else-expr>*** and return result

<u>Example: Sign of a number:</u>

```
(define (sign x)
  (cond ((positive? x) 1)
        ((negative? x) -1)
        (else          0)))
```

Data Types

- Built-in data types: numbers, characters, strings, Booleans, ...

true, false

```
... -1, 0, 1 ...
```

```
3.141592653589793
```

```
"Hallo"
```

```
#T, #F
```

- Symbols
  - like identifiers
  - but are data elements
  - have identity, can be compared for equality

```
+, pi, x, ...
```

```
(define mySyb 'x)
```

```
(eq? 'x 'y)
```

  - can be used as variables
  - evaluate to their data binding

Symbol pi

```
> (define pi 3.14159)
```

```
> pi
3.14159
```

global environment

pi → 3.14159 …

- Pairs and lists
  - see next

Data Structure Pair [1/2]

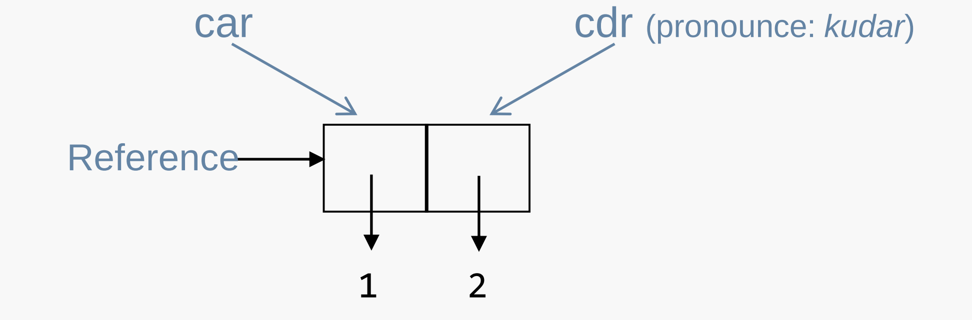
Pair (or also named **cons cell**) is a memory cell with 2 parts

- **cons**: creation of pair

```
> (cons 1 2)
(1 . 2)
```

  - ☐ Creates a cons-cell on heap
  - ☐ returns a reference

internal representation



external representation

```
(1 . 2)
```

- **car**: access first element

car = head

```
> (car (cons 1 2))
1
```

- **cdr**: access second element

cdr = tail

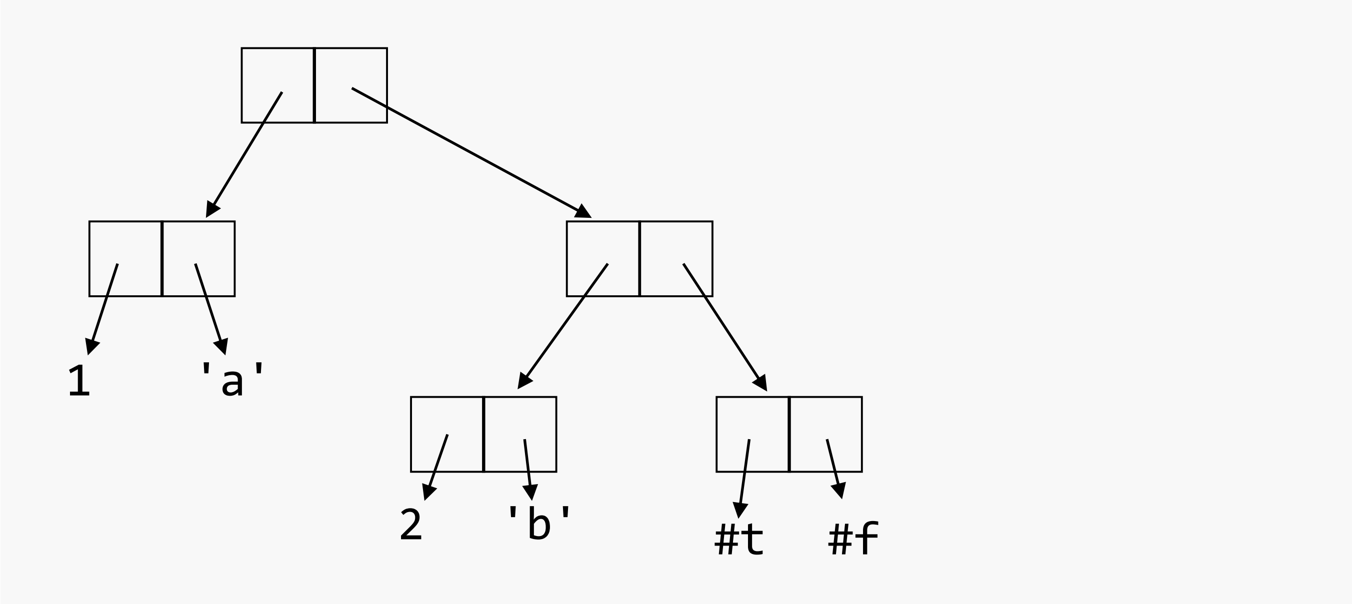
```
> (cdr (cons 1 2))
2
```

Data Structure Pair [2/2]

- Primary memory abstraction for creating more complex data structures
- Car and cdr can point to arbitrary data elements
  ➔ **not statically typed**
- Arbitrary recursive tree structures possible

```
> (cons (cons 1 'a') (cons (cons 2 'b')) (cons #t #f)))
```

creation



internal representation

```
((1 . a) . ((2 . b) . (#t . #f)))
```

external representation

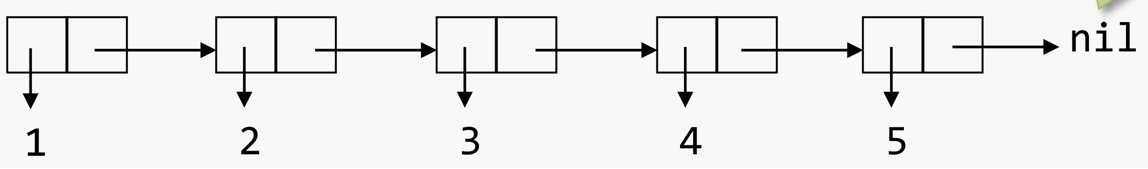
JKU

Lists

- Lists are
  - either empty list **nil**

    also written as ()
  - or recursive structures of pairs where
    - **car** points to **current element**
    - **cdr** points to **rest** of list
    - and last element is empty list **nil**

empty list at end



internal representation

```
(1 2 3 4 5)
```

```
(1 . (2 . (3 . (4 . (5 . nil)))))
```

external representation

- ***list representation***
- ***dot notation***

- Constructing lists
  - using **cons**

```
> (cons 1 (cons 2 (cons 3 (cons 4 (cons 5 nil)))))
```

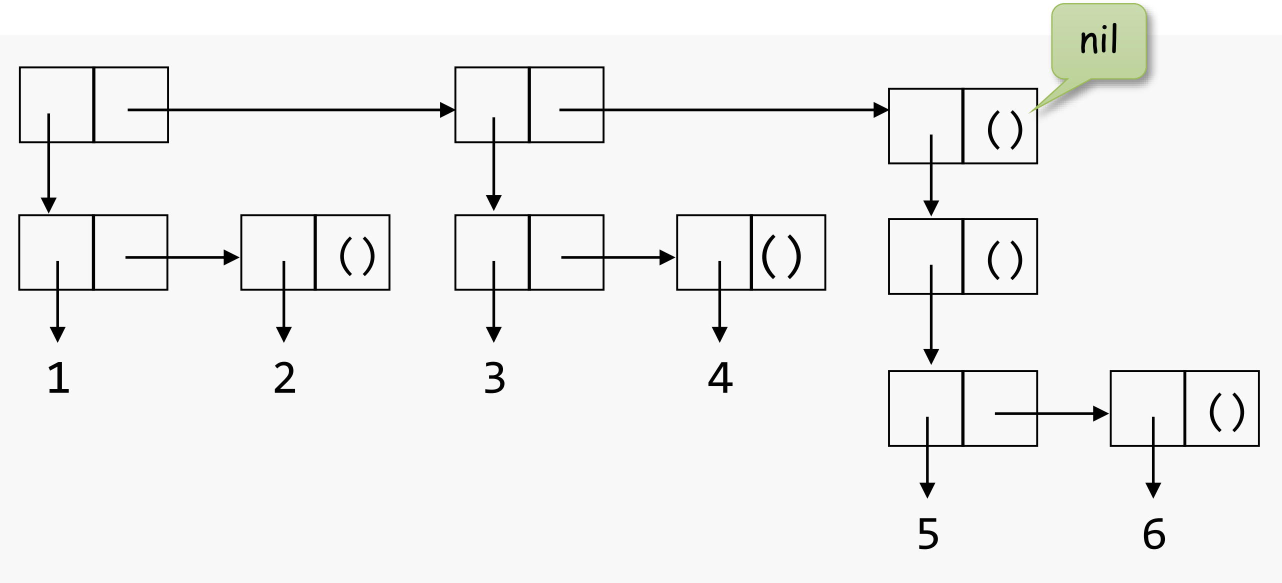
  - using **list**

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

Lists with sub-lists

- Lists with sub-lists allow building complex structures

```
> (list (list 1 2) (list 3 4) (list (list 5 6)))
((1 2) (3 4) ((5 6)))
```



internal representation

```
((1      2)      (3       4 )   ((5        6)))
```

external representation

Higher-order Functions for Lists

- Example: map

```
(define (map fn l)
  (if (eq? l nil)
      nil
      (cons (fn (car l)) (map fn (cdr l))))
  )
```

function application

```
> (map (lambda (x) (* x x)) '(1 2 3))
(1 4 9)
```

- Example: filter

function application

```
(define (filter pred l)
  (if (eq? l nil)
      nil
      (if (pred (car l))
          (cons (car l) (filter pred (cdr l)))
          (filter pred (cdr l))))
)
```

```
> (filter (lambda (x) (> x 0)) '(-1 2 0 1))
(2 1)
```

EXPRESSIONS

- Expressions are represented lists

```
> (+ 1 2)
3
```

List (+ 1 2) evaluated as expression

```
> (sqrt
    (+
      (* 3 3)
      (* 4 4)))
5
```

List with sublists

➔ programs are lists and can be manipulated in the same way as other data elements!

Expressions vs. Data Elements

- Problem with evaluation:
  - Lists are evaluated as function applications or as a special form
  - Symbols are evaluated as variables with data bindings
- Question:
  - How can lists and symbols be used as data values in expressions?

Cannot evaluate list (1 2 3) as function application

```
> (car (1 2 3))
```

application: not a procedure;

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

a: undefined;

Cannot evaluate a as variable!

Quote Special Form

- Quote: Protection from evaluation:
  - lists and symbols are not evaluated
  - but used as data elements

```
(quote <expression>)
```

or short form

```
'<expression>
```

```
> (car (quote (1 2 3)))
1
```

```
> (car '(1 2 3)))
1
```

```
> (list (quote a) (quote b) (quote c))
(a b c)
```

```
> (list 'a 'b ‘c)
(a b c)
```

JKU

Symbolic Computation

Construct expression and evaluate it with built-in function eval

```
> (eval (list '+ 1 2))
```

Interpreter

```
(eval '(+ 1 2))
```

```
3
```

```
> (+ 1 2)
3
```

Construct function and apply it

- Construct lambda expression

```
> (eval (list 'lambda (list 'x 'y) (list '+ 'x 'y)))
```

```
> (eval '(lambda (x y) (+ x y)
```

```
#<procedure>
```

```
> (lambda (x y) (+ x y))
#procedure
```

- Create function object and define function

```
> (define f (eval (list 'lambda (list 'x 'y) (list '+ 'x 'y))))
#<procedure>
```

```
> (define f (lambda (x y) (+ x y)))
```

- Create and evaluate function application

```
> (eval (list 'f 1 2))
3
```

```
> (f 1 2)
3
```

Example: Symbolic Differentiation [1/7]

Problem formulation

- representation of expression as lists

```
(* (* x 2) (* x 3))
```

- function to symbolically compute the derivative of expression by a given variable

```
> (deriv '(* (* x 2) (* x 3)) 'x)
(+ (* (* x 2) 3) (* 2 (* x 3)))
```

- implementation of the following rules

$$\boxed{\frac{dc}{dx} = 0} \quad \boxed{\frac{dx}{dx} = 1} \quad \boxed{\frac{d(u+v)}{dx} = \frac{du}{dx} + \frac{dv}{dx}} \quad \boxed{\frac{d(u*v)}{dx} = u*\frac{dv}{dx} + v*\frac{du}{dx}}$$

Example: Symbolic Differentiation [2/7]

Approach

- Data representation
  - expression as lists, symbols and literals

```
(* (* x 2) (* x 3))
```

- Data abstraction
  - function for creating expressions

```
(define sum-expr (make-sum a1 a2))
```

  - functions for accessing data elements

```
(addend sum-expr)
```

  - functions for testing elements

```
(sum? sum-expr)
```

- Function for the solution

```
(deriv sum-expr 'x)
```

EXAMPLE: SYMBOLIC DIFFERENTIATION [3/7]

Data abstraction for expressions

- constants as numbers

```
3
```

```
(define (constant? x)
  (number? x))
```

check if constant expression

- variables as symbols

```
x
y
```

```
(define (variable? x)
   (symbol? x))

(define (same-variable? v1 v2)
   (and (variable? v1) (eq? v1 v2)))
```

check if variable

check if same variables v1 and v2

EXAMPLE: SYMBOLIC DIFFERENTIATION [4/7]

- sum as list of 3 elements
  - symbol + at first 1st position
  - addend and augend at 2nd and 3rd position

```
(+ x 3)
```

```
(define (make-sum a1 a2)
  (list '+ a1 a2))

(define (sum? expr)
  (if (list? expr) (eq? (car expr) '+) #f))

(define (addend expr)
  (car (cdr expr)))

(define (augend expr)
  (car (cdr (cdr expr))))
```

constructor for sum expressions

check if sum expression

accessing addend

accessing augend

Example: Symbolic Differentiation [5/7]

- product as list of 3 elements
  - ☐ symbol * at first 1st position
  - ☐ multiplicand and multiplier at 2nd and 3rd position

```
(* x 2)
```

```
(define (make-product m1 m2)
  (list '* m1 m2))
```

constructor for product expressions

```
(define (product? expr)
  (if (list? expr) (eq? (car expr) '*) #f))
```

check if product expression

```
(define (multiplicand expr)
  (car (cdr expr)))
```

accessing multiplicand

```
(define (multiplier expr)
  (car (cdr (cdr expr))))
```

accessing multiplier

Example: Symbolic Differentiation [6/7]

- Symbolic differentiation as implementation of rules

```
(define (deriv expr dx)
  (cond
    ((constant? expr)
         0)

    ((variable? expr)
         (if (same-variable? expr dx) 1 0))

    ((sum? expr)
         (make-sum
           (deriv (addend expr) dx)
           (deriv (augend expr) dx)))

    ((product? expr)
         (make-sum
           (make-product
             (multiplicand expr)
             (deriv (multiplier expr) dx))
           (make-product
             (deriv (multiplicand expr) dx)
             (multiplier expr))))))
```

$$\frac{dc}{dx} = 0$$

$$\frac{dx}{dx} = 1$$

$$\frac{d(u+v)}{dx} = \frac{du}{dx} + \frac{dv}{dx}$$

$$\frac{d(u*v)}{dx} = u*\frac{dv}{dx} + v*\frac{du}{dx}$$

```
>(deriv '(* (* x y) (+ x 3)) 'x)
(+ (* (* x y) (+ 1 0)) (* (+ (* x 0) (* 1 y)) (+ x 3)))
```

Example: Symbolic Differentiation [7/7]

- make-sum and make-product with simplification of resulting expressions

```
(define (make-sum a1 a2)
  (cond ((and (number? a1) (number? a2)) (+ a1 a2))
        ((number? a1) (if (= a1 0) a2 (list '+ a1 a2)))
        ((number? a2) (if (= a2 0) a1 (list '+ a1 a2)))
        (else (list '+ a1 a2))))
```

```
(define (make-product m1 m2)
  (cond ((and (number? m1) (number? m2)) (* m1 m2))
        ((number? m1)
           (cond ((= m1 0) 0)
                 ((= m1 1) m2)
                 (else (list '* m1 m2))))
        ((number? m2)
           (cond ((= m2 0) 0)
                 ((= m2 1) m1)
                 (else (list '* m1 m2))))
        (else (list '* m1 m2))))
```

```
> (deriv '(* (* x y) (+ x 3)) 'x)
'(+ (* x y) (* y (+ x 3)))
```

<u>Beispiele</u>:

<u>a1</u>	<u>a2</u>	
1	2	→ 1+2
0	y	→ y
x	0	→ x
x	y	→ (+ x y)

<u>m1</u>	<u>m2</u>	
1	2	→ 1*2
0	y	→ 0
1	y	→ y
3	y	→ (* 3 y)
x	0	→ 0
x	1	→ x
x	3	→ (* x 3)
x	y	→ (* x y)

SUMMARY

- Lisp is an **implementation of the lambda calculus**
  - with **strict evaluation**
  - with some **special forms**
- **Recursive lists** are the main data structures
  - built up from **pairs** with **first (car) is first element** and **rest (cdr) is rest of list**
  - **nil** as **empty list**
- **Lisp expressions** are represented as **lists**
  - **Lisp expressions** as **data objects**
  - **Data objects** as **Lisp expressions**

II.1.в Haskell

Haskell

name by: **Haskell** B. Curry (1900–1982), US-american mathematician working on the Lambda Calculus

- Developed by a consortium (1987-)
  - standard for education and research
  - now also applied in industry-size projects
    - see http://www.haskell.org/haskellwiki/Haskell_in_industry

- Resources: www.haskell.org
  - Language specification
  - Tutorials
  - Literature
  - Implementations
  - Tools
  - Libraries
  - Example
  - …

Haskell Characteristics

- **Pure functional language**
  - no side effects
  - only immutable data

- **Statically typed**
  - based on ***typed lambda calculus***

- Data types
  - **Algebraic data types**
  - **Parametric polymorphism**

- **Non-strict call-by-need evaluation** semantics (***lazy evaluation***)

Typed Lambda Calculus

- The Lambda Calculus can be extended so that expressions carry types!
  ➔ Expressions are valid if they are type correct

- Types
  - Base types

    B = Int | Bool | ...

  - Types are base types plus function types

    T = B | t_1 -> t_2   where $t_1 , t_2 \in T$

    Function from type t_1 to t_2!

Typed Lambda Calculus

with $\mathbf{t}, \mathbf{t_1} \ldots \in \mathbf{T}$

- Typed lambda expressions
  - Typed variables

    v : t

  - Typed lambda abstraction

    $(\lambda\, v : t_1 \;.\; E) : t_1 \text{->} t_2$

    where $E : t_2$

  - Typed function application

    $(F\;\; E) : t_2$

    where $F : t_1 \rightarrow t_2$ and $E : t_2$

Typed Lambda Calculus

If untyped: **IF = (λ c a b . c a b)**

- Function IF

IF_{Int} = (λ c : Bool a : Int b : Int . c a b) : Bool -> Int -> Int -> Int

IF_{Bool} = (λ c : Bool a : Bool b : Bool . c a b) : Bool -> Bool -> Bool -> Bool

Separate if-functions for all data types needed!!

- Type variables universally quantified
  - e.g., function IF with generic type variable

∀t ∈ T => IF = (λ c : Bool a : t b : t . c a b) : Bool -> t -> t -> t

Generic function definition with type parameter t

Haskell: Expressions

- Operators
  - Infix notation
  - Prefix notation possible
    - operators in rounded brackets
    - `(<op>)` is function name for operator `<op>`

```
> 1 + 2
3
> (2 + 3) * 5
25
```

Function (+)

```
> (+) 1 2
3
> (*) ((+) 2 3) 5
25
```

- Function applications
  - Prefix notation without brackets
    - juxtaposition of function and arguments

```
> mod 8 3
2
```

Would be in Java:
mod(8, 3) ?!

  - Infix notation
    - for functions with 2 arguments
    - function name in back-quotes

```
> 8 `mod` 3
2
```

  - Precedence and associativity
    - Use rounded brackets to define structure of compound expressions

```
> (+) (*) 2 3 4
ERROR
```

(+) is left-associative
➔ ((+) (*) 2) 3 4

```
> (+) ((*) 2 3) 4
10
```

correct usage of brackets

Lambda Abstractions

- **Lambda abstractions** are **function literals** which create **function objects**

Example:

```
\ x -> x*2 + 1
```

lambda symbol | formal parameter | mapping symbol | defining expression

In lambda calculus:
`λx . + (* x 2) 1`

Function literals with multiple arguments

```
\ x y -> x + y
```

internally always represented in **Curry-form**

```
\ x -> (\ y -> x + y )
```

Functions are **first-class objects**

- can be stored in variables and data structures (e.g., lists)
- can be passed as parameter
- can be created and returned from functions

Function Types

- **Functions** have **data type**

```
a -> b
```

represents **type of functions**
which map values of some **type a** to values of some **type b**

a and b are type variables

- Function objects with specific types for type variables a and b

is type

```
not      :: Bool -> Bool
isDigit  :: Char -> Bool
xOr      :: Bool -> (Bool -> Bool)
(+)      :: Num a => a -> (a -> a)
```

polymorphic for
different number types

Function Definitions

Function definitions as assignment of lambdas to variables

```
xOr  = \x ->  (\y ->  (x || y) && not (x && y))
```

- With short form

```
xOr x y  = (x || y) && not (x && y)
```

Function name

Formal parameters

Defining expression

- With explicit type declaration (**optional** but **recommended** !)

Function name

Types of parameters

Type of result

```
xOr :: Bool -> (Bool -> Bool)
xOr x y  = (x || y) && not (x && y)
```

Curry-form and right associative

Expressions: If and Recursion

If-then-else conditional expression

```
sign  :: Int -> Int
sign n  = if n < 0  then -1
          else if n == 0  then 0
               else 1
```

If is expression and returns value of expression in then or else branch

Recursion

```
fac :: Int -> Int
fac n  = if n == 1 then 1
         else n * fac (n-1)
```

Note: Recursion is without side effects !

JKU

Conditional Expressions: Guards

Guarded definitions

- Conditional expression with multiple branches
- Each branch guarded by a condition
- First branch whose guard gives true provides value

```
sign n
    | n < 0          = -1
    | n == 0         = 0
    | otherwise      = 1
```

analogous to mathematical notation:

```
sign(n) =
     -1     if n < 0
     0      if n = 0
     1      otherwise
```

Local Definitions with let

let for defining local variables

```
sumSquares :: Int -> Int -> Int
sumSquares n m =
    let
        sqN = n*n
        sqM = m*m
    in
        sqN + sqM
```

Local Definitions only valid in **surrounding expression**

= „**Lexical scoping**“ with scope is expression

Local Definitions with Where

where for local definitions

- mostly used for local function definitions

local scope

```
f x =
   ... local ...

   where local = ...
```

local definition

analogous to mathematical notation:

```
    f(x) =
      …   local …
   where local = …
```

<u>Example:</u>

tail recursive

```
fac :: Int -> Int
fac n =
  facAcc 1 1
  where facAcc acc i
          | i > n  = acc
          | True   = facAcc (i * acc) (i + 1)
```

Layout Blocks and Indentations

- In Haskell one primarily works with **single compound expressions**
  - here precedence rules and rounded brackets define the structure

```
xOr x y  = (x || y) && not (x && y)
```

```
sign  n =  if n < 0  then -1
           else if n == 0  then 0
                else 1
```

- Keywords **let**, **where**, **do**, and **of** introduce so-called **layout blocks** containing multiple elements
  - block enclosed in **braces** with elements separated by **semicolons**
  - alternatively, use proper **indentation**
    ➔ **same indentation** means **same layout block**

```
sumSquares :: Int -> Int -> Int
sumSquares n m =
   let { sqN = n*n ; sqM = m*m }
   in sqN + sqM
```

```
sumSquares :: Int -> Int -> Int
sumSquares n m =
   let
      sqN = n*n
      sqM = m*m
   in sqN + sqM
```

same layout block

Algebraic Data Types (ADTs)

Grammar-like definitions of data structures

- with **products**, **variants** and **generic type parameters**

Products (similar to records)

- type definition with constructor function

constructor

```
data Person    =    MkPers String Int
```

type

data tag =
constructor function

contructor parameters
= object fields

- creation of data objects

```
frank = MkPers "Frank" 25
```

```
ann = MkPers "Ann" 23
```

call of constructor
creates Person object

Data objects:
- Data tag
- plus field values

MkPers	"Frank"	25

MkPers	"Ann"	23

ALGEBRAIC DATA TYPES (ADTS)

Grammar-like definitions of data structures

- with **products**, **variants** and **generic type parameters**

Variants: alternative records

- type definition

or

```
data IntList  =  Nil    |    Cons Int IntList
```

type

1st variant

2nd variant (recursive)

- creation of data elements

```
list = Cons 2 (Cons 1 Nil)
```

Data objects:

- two variants
- with two different tags

Nil

Cons	1	Nil

Cons	2	Cons 1 Nil

ALGEBRAIC DATA TYPES (ADTS)

Grammar-like definitions of data structures

- with **products**, **variants** and **generic type parameters**

Polymorphic types: use type parameters for generic types

- polymorphic type definition

```
data List a = Nil  |   Cons a (List a)
```

Polymorphic type **List a** with type parameter **a**

using generic type parameter **a**

- creation of data elements

```
charList = Cons 'c' (Cons 'b' (Cons 'a' Nil))       :: List Char
boolList = Cons True (Cons False (Cons True Nil))   :: List Bool
numList = Cons 3 (Cons 2 (Cons 1 Nil))              :: Num a => List a
```

polymorphic !

Enumeration Types

ADT with variants with data tags only

```
data DayOfWeek =   Monday
                 | Tuesday
                 | Wednesday
                 | Thursday
                 | Friday
                 | Saturday
                 | Sunday
```

only data tags

Pattern Matching with ADTs

ADTs allow matching values to patterns

- Patterns correspond to data constructors
- Field values are bound to pattern variables

```
data Person = MkPers String Int
```

construction:

```
frank = MkPers "Frank" 25
```

pattern match:

```
(MkPers name age) = frank
```

data tag

variables for fields

Variants are distinguished in case expressions

```
data List a = Nil  |  Cons a (List a)
```

```
list = Cons 1 ...
```

```
case list of
   Nil         -> ... handle empty list ...
   Cons v tail -> ... handle non-empty list ...
```

list can be either **Nil**

or a Cons with first element **v** and a restlist **tail**

Pattern Matching with ADTs

Example: Function listLength on List

```
listLength :: List a -> Int
listLength list =
   case list of
      Nil          ->  0
      Cons v tail  ->  1 + (listLength tail)
```

translated to

Multiple definitions for functions

- functions can be defined in multiple cases
- with different patterns

```
listLength :: List -> Int
listLength Nil              = 0
listLength (Cons v tail)    = 1 + (listLength tail)
```

PATTERN MATCHING WITH ADTS: MORE FEATURES

- Pattern matching with values and variables

```
fac :: Int -> Int
fac 1   = 1
fac n   = n * fac (n - 1)
```

- Wildcard "_": matches any value, no binding

```
and :: Bool -> Bool -> Bool
and True   expr      = expr
and False  _         = False
```

matches any value

```
or :: Bool -> Bool -> Bool
or True  _            = True
or False expr          = expr
```

- Guards: Patterns with additional condition

```
max :: Int -> Int -> Int
max x y | x >= y  = x
max _ y           = y
```

additional test of bound variables

Pattern Matching with ADTs: More Features

- Recursive patterns

```
containsPerson :: List[Person] -> String -> Bool
containsPerson (Cons   (MkPers n _)   _     ) name | n == name   = True
containsPerson (Cons   _              rest)   name               = findPerson rest name
containsPerson Nil                            _                  = False
```

- @ : Binding matching values to variables

type Maybe see below

```
findPerson :: List[Person] -> String -> Maybe Person
findPerson (Cons  pers@(MkPers n _)  _    ) name | n == name   = Just pers
findPerson (Cons  _                rest)    name               = findPerson rest name
findPerson Nil                              _                  = Nothing
```

matching **Person** value
bound to variable **pers**

Library ADT List

[a] polymorphic recursive list type

same as List a from before but with different built-in syntax

- with value constructors for empty list and cons operator **(:)**

```
data [a] =     []  |  a : [a]
```

list type with type parameter a

empty list

Cons operator (in infix notation)

```
> 1 : 2 : 3 : []
[1, 2, 3]
```

right associative ➔

```
> 1 : (2 : (3 : []))
```

List literals:

```
[1,2,3]
```

abbreviation for

```
1 : 2 : 3 : []
```

value	type
`[False,True,False]`	`:: [Bool]`
`['a','b','c','d']`	`:: [Char]`
`[1, 4, 9, 16]`	`:: Num a => [a]`
`[['a'],['b','c']]`	`:: [[Char]]`
`[]`	`:: [a]`

empty list

generic

Pattern Matching with List

List patterns

`[]` matches empty list
`x:xs` matches any non-empty list
where `x` is bound to first element and `xs` to rest of list

```
len :: [a] -> Int
len []       = 0
len (_:xs)   = 1 + (len xs)
```

Example: equalLists

requires equality operator for a ➔ see Part III

```
equalLists :: Eq a => [a] -> [a] -> Bool
equalLists []     []                 = True
equalLists _      []                 = False
equalLists []     _                  = False
equalLists (x:xs) (y:ys) | x == y = equalLists xs ys
equalLists _      _                  = False
```

Higher-order Functions for Lists

- Example: library function **map**

Note: function type

```
map :: (a -> b) -> [a] -> [b]
map fn []       = []
map fn (x:xs)   = (fn x) : (map fn xs)
```

function application

```
> map (\x -> x * x) [1,2,3]
[1 4 9]
```

- Example: library function **filter**

```
filter :: (a -> Bool) -> [a] -> [a]
filter pred []                = []
filter pred (x:xs) | pred x   = x : (filter pred xs)
filter pred (x:xs)            = filter pred xs
```

```
> filter (\ x -> x > 0) [-1, 2, 0, 1]
[2, 1]
```

Tuple Types

Several values of different types for tuples (pairs, triples, …)

```
(a, b)
(a, b, c)
```

... up to 15 elements ...

ADT definitions:

```
data () a b = (a, b)
data () a b c = (a, b, c)
```

Examples:

```
(1, 'a')            :: (Num c => c, Char)
(False, 'a', 1)     :: (Bool, Char, Num a => a)
```

Working with Tuples

Functions (for pairs only)

- Accessing first element

```
fst :: (a, b) -> a
fst (x, _)  = x
```

- Accessing second element

```
snd :: (a, b) -> b
snd (_, y)  = y
```

Application:

```
> fst (1,'a')
1
```

```
> snd (1,'a')
'a'
```

Tuples in pattern assignments

```
(f, s) = (1, 'a')
```

Bindings: f = 1, s = 'a'

```
(f, s, t) = (1, 'a', True)
```

Bindings: f = 1, s = 'a', t = True

Tuples in case expressions (for distinguishing multiple values)

```
case (boolVal1, boolVal2) of
    (True, False) -> True
    (False, True) -> True
    (_, _)        -> False
```

Working with Characters and Strings

- Datatype Char for Unicode characters
  - Char type and literals (as in Java)

```
Char
```

```
'a'
```

- String type is just are list of characters

```
type String = [Char]
```

  - Strings can be handled as lists

`head "abc"` ➔ 'a' `tail "abc"` ➔ "bc" `null ""` ➔ True

    - Pattern matching

```
case str of
    ""            -> …
    "a"           -> …
    "ab"          -> …
    ('a':rest)    -> …
```

matches
- empty string
- string with single character 'a'
- string with two character 'a' followed by 'b'
- string with first character 'a'

Data Type Maybe

Container of a value which might be empty

Just a : has a value

Nothing : no value

```
data Maybe a =  Just a  |  Nothing
```

compare Java's **Optional** class

- Constructing Maybe values

```
positive :: Int -> Maybe Int
positive x = if x > 0 then Just x else Nothing
```

- Pattern matching Maybe values s

```
case (positive x) of
  Just p  -> "Positive value is " ++ (show p)
  Nothing -> "Not a positive value"
```

Example: Symbolic Differentiation

```haskell
data Expr =
      Lit Int
    | Var String
    | Plus Expr Expr
    | Times Expr Expr
```

Algebraic data type for expressions with
- Literals with Int value
- Variables with name
- Plus expression with left and right operand
- Times expression with left and right operand

```haskell
deriv :: Expr -> Expr -> Expr

deriv (Lit _)      dx                   = Lit 0

deriv (Var n)      (Var x) | n == x    = Lit 1

deriv (Var _)      dx                   = Lit 0

deriv (Plus u v)   dx                   = Plus (deriv u dx) (deriv v dx)

deriv (Times u v)  dx                   = Plus (Times u (deriv v dx)) (Times v (deriv u dx))
```

$$\frac{dc}{dx} = 0$$

$$\frac{dx}{dx} = 1$$

$$\frac{dy}{dx} = 0$$

$$\frac{d(u+v)}{dx} = \frac{du}{dx} + \frac{dv}{dx}$$

$$\frac{d(u*v)}{dx} = u*\frac{dv}{dx} + v*\frac{du}{dx}$$

Summary and Outlook

- Haskell is implementation of Typed Lambda Calculus, plus
  - algebraic data types for defining new types
    ➔ **see more in lecture on Algebraic data types**
  - case expressions for pattern matching
    ➔ **see more in lecture on pattern matching**
  - some special syntactic constructs

- Haskell's expressive power comes from higher-order functions
  ➔ **see lecture on higher-order functions**

- Haskell's execution engine (G-machine) implements a lazy *Call-by-Need* evaluation scheme
  ➔ **see lecture on non-strict execution semantics**

Quick Reference: Basic data types

Predefined basic data types:

`Bool`	Boolean values True und False
`Char`	Characters (e.g. 'a')
`Int`	Integers with limited precision
`Integer`	Integers with arbitrary precision
`Float`	Floating point numbers (32 bit)
`Double`	Floating point numbers (64 bit)
`Ratio`	Rational numbers
`Complex`	Complex numbers

…

Quick Reference: Built-in operators

- Associativity and precedence order of built-in operators

higher value has precedence

operators	description	associativity	precedence order
`!!`	nth element in list	left	9
`.`	dereference element in module	right	9
`**, ^`	power	right	8
`*, /, `div`,`mod``	arithmetic	left	7
`+, -,`	arithmetic	left	6
`:`	cons operator	right	5
`++`	concatenation operator	right	5
`==, <, >, <=, >=`	relations operators	N/A	4
`&&, ||`	logical operators	right	3

Associativity

- Left associative

```
x + y + z = (x + y) + z
```

- Right associative

```
x : y : z = x : (y : z)
```

Quick Reference: Arithmetic Functions

all numbers

- addition
- subtraction
- multiplication
- negation
- absolute value
- sign

```
x + y
x - y
x * y
negate y
abs x
signum x
```

must use brackets

or `(-y)`

integer division !

integer numbers (called *integral* numbers)

- integer division
- remainder

```
x `div` y
x `mod` y
```

or alternatively

```
x `quot` y
x `rem` y
```

where the two differ when dividing negative numbers

```
div (-5) 2 = (-3)
mod (-5) 2 = 1
```

```
quot (-5) 2 = (-2)
rem  (-5) 2 = (-1)
```

- even
- odd
- …

```
even x
odd x
…
```

Quick Reference: Arithmetic Functions

floating point numbers (*fractional* numbers)

- floating division
- reciprocal value
- e to the power of x
- logarithm base e
- logarithm base b
- sqrt

```
x / y
recip x
exp x
log x
logBase b x
sqrt x
```

floating point division !

- different power operators
  - to positive integer
  - to positive or negative integer
  - to any base and any power

```
x ^ i
x ^^ -i
x ** y
```

e.g.

```
0.2^2
0.2^^(-2)
0.3**0.5
```

Type conversions

- from integer to double

```
fromIntegral x
```

e.g.

```
sqrtOfInt :: Int -> Double
sqrtOfInt x = sqrt (fromIntegral x)
```

- from double to integer

```
round x
truncate x
ceiling x
floor x
```

Note: Haskell is very strict on the types of the arguments of arithmetic operators
→ no implicit casts
→ all type conversions must be done explicitly

- integer and fraction part of float number

```
(i, f) = properFraction 1.2
```

1

0.2

Quick Reference: List Functions

Test for empty list

```
null :: [a] -> Bool
```

create new list with element added in front

```
(:) :: a -> [a] -> [a]
```

Accessing first element

```
head :: [a] -> a
```

Accessing rest list

```
tail :: [a] -> [a]
```

n-th element

```
(!!) :: [a] -> Int -> a
```

Concatenating two lists

```
(++) :: [a] -> [a] -> [a]
```

Taken first n elements

```
take :: Int -> [a] -> [a]
```

Dropping first n elements

```
drop :: Int -> [a] -> [a]
```

Length of list

```
length :: [a] -> Int
```

Examples:

```
> null []
True
```

```
> null [1,2]
False
```

```
> 0 : [1,2,3]
[0,1,2,3]
```

```
> head [1,2,3]
1
```

```
> tail [1,2,3]
[2, 3]
```

```
> [1,2,3] !! 0
1
```

```
> [1,2,3] !! 2
3
```

```
> [1,2] ++ [3,4] ++ []
[1,2,3,4]
```

```
> take 2 [1,2,3]
[1,2]
> drop 2 [1,2,3]
[3]
```

```
> length [1,2,3]
3
```

Quick Reference: List functions

import Data.List required

```
reverse :: [a] -> [a]
```

```
concat :: [[a]] -> [a]
```

```
> concat [[1,2],[3,4],[4,5]]
[1,2,3,4,4,5]
```

```
splitAt :: Int -> [a] -> ([a], [a])
```

```
isPrefixOf :: [a] -> [a] -> Bool
```

```
isSuffixOf :: [a] -> [a] -> Bool
```

```
isInfixOf :: [a] -> [a] -> Bool
```

```
elem :: Eq a => a -> [a] -> Bool
```

```
elemIndex :: Eq a => a -> [a] -> Maybe Int
```

see next slide

```
elemIndices :: Eq a => a -> [a] -> [Int]
```

```
zip :: [a] -> [b] -> [(a, b)]
```

```
permutations :: [a] -> [[a]]
```

```
subsequences :: [a] -> [[a]]
```

```
> subsequences [1,2]
[[],[1],[2],[1,2]]
```

Quick Reference: Special Lists

```haskell
import Data.List
```

Strings

```haskell
lines :: String -> [String]
```

```haskell
words :: String -> [String]
```

Sets

```haskell
union :: Eq a => [a] -> [a] -> [a]
```

```haskell
intersect :: Eq a => [a] -> [a] -> [a]
```

```haskell
(\\) :: Eq a => [a] -> [a] -> [a]
```

Boolean lists

```haskell
or :: [Bool] -> Bool
```

```haskell
and :: [Bool] -> Bool
```

Lists where element types define an order relation (`Ord a`)

```haskell
maximum :: Ord a => [a] -> a
```

```haskell
minimum :: Ord a => [a] -> a
```

```haskell
sort :: Ord a => [a] -> [a]
```

```haskell
insert :: Ord a => a -> [a] -> [a]
```

Quick Reference: HOFs for lists

Higher-order list functions available from Prelude (without import)

```
map :: (a -> b) -> [a] -> [b]
```

```
filter :: (a -> Bool) -> [a] -> [a]
```

```
takeWhile :: (a -> Bool) -> [a] -> [a]
```

```
dropWhile :: (a -> Bool) -> [a] -> [a]
```

```
foldr :: (a -> b -> b) -> b -> [a] -> b
```

```
foldl :: (a -> b -> a) -> a -> [b] -> a
```

```
foldr1 :: (a -> a -> a) -> [a] -> [b]
```

```
foldl1:: (a -> a -> a) -> [a] -> [a]
```

```
zipWith :: (a -> b -> c) -> [a] -> [b] -> [c]
```

```
any :: (a -> Bool) -> [a] -> Bool
```

```
all :: (a -> Bool) -> [a] -> Bool
```

Quick Reference: HOFs for lists

Higher-order list functions available with **`import Data.List`**

```
find :: (a -> Bool) -> [a] -> Maybe a
```

```
partition :: (a -> Bool) -> [a] -> ([a], [a])
```

```
findIndex :: (a -> Bool) -> [a] -> Maybe Int
```

```
findIndices :: (a -> Bool) -> [a] -> [Int]
```

```
insertBy :: (a -> a -> Ordering) -> a -> [a] -> [a]
```

```
sortBy :: (a -> a -> Ordering) -> [a] -> [a]
```

```
maximumBy :: (a -> a -> Ordering) -> [a] -> a
```

```
minimumBy :: (a -> a -> Ordering) -> [a] -> a
```

`... and many more ...`

```
import Data.List
```

where **`Ordering`** is an enumeration type

```
compare :: Ord a => a -> a -> Ordering
```

having three values **`LT`**, **`EQ`**, and **`GT`**
and for example function **`compare`**

```
data Ordering = LT | EQ | GT
```

returns an **`Ordering`** can be used to compare **`Ord`** types