Functional Programming

- No side effects
- Recursion instead of iteration
- Polymorphism is supported => one function handles multiple types

---

Haskell

- Function with different cases:

```
functionName x1 ... xn
  | guard1 = expr1
    :
  | guardm = exprm
  |otherwise =e (optional)
```

- Define constant: myConstant = 5
- Operators: +/-/^/-
  - Default written in infix: 5 + 7
  - Prefix notation: (+) 5 7
- Integrated functions: div / mod / abs
  - Default in prefix: mod 5 7
  - Infix notation: 5 `mod` 7
- Types returning True/False: > / >= / == / /= / <= / <
- Tuples:
  - Can be nested (3,("hi",True)) :: (Int, (String, Bool))
  - Can be taken as argument

```
addPair :: (Int, Int) -> Int
addPair (x, y) = x + y
? addPair (3, 4)
7
```

- let:
  - Before function definition

```
gcdInt1 :: Int -> Int -> Int
gcdInt1 x y =
        let
            x' = if x>0 then x else -x
            y' = if y>0 then y else -y
        in gcdiv x' y'
```

- Where:
  - After function definition

```
f p1 p2 ... pm
  | g1 = e1
  | g2 = e2
    :
  | gk = ek
  where
    v1 a1 ... an = r1
    v2 = r2
    :
```

- If (condition)
  then <true value>
  else <false value>

Natural Deduction for proposition formulae

- Natural Deduction:
  - Natürliches Schliessen
  - Mittels Ableitungsbaum zeigen, dass Formel eine Tautologie ist
  - Formeln können Propositional-Logic oder First-order-Logic sein
- Propositional Logic:
  - Aussagenlogik
  - Conjunction: $\wedge$
  - Disjunction: $\vee$
  - Conditional: $\rightarrow$
  - Evaluation formula: α
  - $\models$: Tautology
  - $\vdash$: Ableitungsschritt

$$\frac{\Gamma \vdash A \qquad \Gamma \vdash B}{\Gamma \vdash A \wedge B}\ \wedge\text{-}I \qquad \frac{\Gamma \vdash A \wedge B}{\Gamma \vdash A}\ \wedge\text{-}EL \qquad \frac{\Gamma \vdash A \wedge B}{\Gamma \vdash B}\ \wedge\text{-}ER$$

$$\frac{\Gamma, A \vdash B}{\Gamma \vdash A \rightarrow B}\ \rightarrow\text{-}I \qquad \frac{\Gamma \vdash A \rightarrow B \qquad \Gamma \vdash A}{\Gamma \vdash B}\ \rightarrow\text{-}E$$

$$\frac{\Gamma \vdash A}{\Gamma \vdash A \vee B}\ \vee\text{-}IL \qquad \frac{\Gamma \vdash B}{\Gamma \vdash A \vee B}\ \vee\text{-}IR$$

$$\frac{\Gamma \vdash A \vee B \qquad \Gamma, A \vdash C \qquad \Gamma, B \vdash C}{\Gamma \vdash C}\ \vee\text{-}E$$

$$\frac{\Gamma \vdash \bot}{\Gamma \vdash A}\ \bot\text{-}E \qquad \frac{\Gamma \vdash \neg A \qquad \Gamma \vdash A}{\Gamma \vdash B}\ \neg\text{-}E \qquad \frac{\Gamma, \neg A \vdash \bot}{\Gamma \vdash A}\ RAA$$

Derivation rules for propositional logic

Natural deduction for first-order-logic

- First-order-logic: · introduces $\forall$ / $\exists$
  · Has functions with extra truth-tables
- Each occurance of each variable in a formula is bound or free
  · Bounded variable by binding-operator $\forall x$ / $\exists x$
  · Free: not bound

$$(q(x) \vee \exists x.\, \forall y.\, p(f(x), z) \wedge q(a)) \vee \forall x.\, r(x, z, g(x))$$

- α-conversion: · We can rename only the bound variables

$\forall x.\, \exists y.\, p(x, y)$	$\forall y.\, \exists x.\, p(y, x)$	yes
$\exists z.\, \forall y.\, p(z, f(y))$	$\exists y.\, \forall y.\, p(y, f(y))$	no
$(\forall x.\, p(x)) \vee (\exists x.\, q(x))$	$(\forall z.\, p(z)) \vee (\exists y.\, q(y))$	yes
$p(x) \to \forall x.\, p(x)$	$p(y) \to \forall y.\, p(y)$	no

- Omitting parentheses: 1. $\neg$
  2. $\wedge$ } associates to left
  3. $\vee$ }
  4. $\to$ } associates to right
  · Quantifiers extend to right as far as possible

$A \vee B \wedge \neg C \to A \vee B$	$\big(A \vee (B \wedge (\neg C))\big) \to (A \vee B)$
$A \to B \vee A \to C$	$A \to \big((B \vee A) \to C\big)$
$A \wedge \forall x.\, B(x) \vee C$	$A \wedge \big(\forall x.\, (B(x) \vee C)\big)$

- Structure: $S = (U_S, I_S)$ — U_S: universe, I_S: function mapping
- Interpretation: $\mathcal{I} = (S, V)$ — V: Evaluation

- If evaluation v evaluates the structure S to True, it is called a model
- If every evaluation v_i is true, then it's valid
- A structure is satisfiable, if there exists at least one model

$$\frac{}{\Gamma, A \vdash A}\ \textit{axiom} \qquad \frac{\Gamma, A \vdash B}{\Gamma \vdash A \to B}\ \to I \qquad \frac{\Gamma \vdash A \to B \qquad \Gamma \vdash A}{\Gamma \vdash B}\ \to E$$

$$\frac{\Gamma \vdash \bot}{\Gamma \vdash A}\ \bot E \qquad \frac{\Gamma, A \vdash \bot}{\Gamma \vdash \neg A}\ \neg I \qquad \frac{\Gamma \vdash \neg A \qquad \Gamma \vdash A}{\Gamma \vdash B}\ \neg E$$

$$\frac{\Gamma \vdash A \qquad \Gamma \vdash B}{\Gamma \vdash A \wedge B}\ \wedge I \qquad \frac{\Gamma \vdash A \wedge B}{\Gamma \vdash A}\ \wedge EL \qquad \frac{\Gamma \vdash A \wedge B}{\Gamma \vdash B}\ \wedge ER$$

$$\frac{\Gamma \vdash A}{\Gamma \vdash A \vee B}\ \vee IL \qquad \frac{\Gamma \vdash B}{\Gamma \vdash A \vee B}\ \vee IR$$

$$\frac{\Gamma \vdash A \vee B \qquad \Gamma, A \vdash C \qquad \Gamma, B \vdash C}{\Gamma \vdash C}\ \vee E \qquad \frac{\Gamma \vdash A(x)}{\Gamma \vdash \forall x.\, A(x)}\ \forall I^{*}$$

$$\frac{\Gamma \vdash \forall x.\, A(x)}{\Gamma \vdash A(t)}\ \forall E$$

$$\frac{\Gamma \vdash A(t)}{\Gamma \vdash \exists x.\, A(x)}\ \exists I \qquad \frac{\Gamma \vdash \exists x.\, A(x) \qquad \Gamma, A(x) \vdash B}{\Gamma \vdash B}\ \exists E^{**}$$

Derivation rules for first order logic

Note that $*$ and $**$ are not part of the $\forall I$ and $\exists E$ rules' names; they signify the following side conditions:

$*$ x must not appear freely in Γ

$**$ x must not appear freely in Γ and in B

Lists

- Empty list: `[]`
- Non-empty-list: `(x : xs)` (`x :: T` and `xs :: [T]`) — `(x:xs)`
- Short hand: `1 : (2 : (3 : []))` written as `[1, 2, 3]` — Built via induction
- `['a', 'b', 'c'] == "abc"`
- `[n, p .. m]` count from n to m in steps of p-n
- `[3..6] = [3,4,5,6]`
- Append to lists:
  - `[2] ++ [3,4] = [2,3,4]`
    - ↳ For adding lists together
  - `2 : [3,4] = [2,3,4]`
    - ↳ For adding one element

Pattern Matching

- check if an argument has the proper form
- Binding together variable and value
- Example: `(x:xs)` matching with `[2,3,4]`
  - ↳ `x = 2`
  - `xs = [3,4]`

Constants: -2, '1', $True$, []
Variables: x, foo ← Each variable can occur at most once
Wild card: _ ← Don't care what value is
Tuples: $(p_1, p_2, \ldots, p_k)$, where p_i are patterns
Lists: $(p_1 : p_2)$, where p_i are patterns

Zip-function

$$zip\ [2,3,4]\ [4,5,78] \quad = \quad [(2,4),(3,5),(4,78)]$$

- Extra elements in a longer list are discarded*

```
zip (x:xs) (y:ys) = (x,y) : zip xs ys
zip _      _      = []
```

3*

- xs and ys can also be [], if we reached the end on both lists

---

Insertion sort

```
isort :: [Int] -> [Int]
isort []     = []
isort (x:xs) = ins x (isort xs)

ins :: Int -> [Int] -> [Int]
ins a [] = [a]
ins a (x:xs)
  | a <= x    = a : (x : xs)
  | otherwise = x : ins a xs
```

```
isort [3,9,2]
    = ins 3 (isort [9,2])
    = ins 3 (ins 9 (isort [2]))
    = ins 3 (ins 9 (ins 2 (isort [])))
    = ins 3 (ins 9 (ins 2 []))
    = ins 3 (ins 9 [2])
    = ins 3 (2 : (ins 9 []))
    = 2 : (ins 3 (ins 9 []))
    = 2 : (ins 3 [9])
    = 2 : 3 : [9] = [2,3,9]
```

List-comprehension

```
palindromes :: [String] -> [String]
palindromes list = [w1 ++ w2 | w1 <- list, w2 <-list, w1++w2 == reverse (w1++w2) ]
```

- [n `mod` 2 == 0 | n ← [2,4,7]] = [T, T, F]
- [2 * x | x ← [0,1,2,3], x `mod` 2 == 0, x > 3]

2. Wie die Elemente verarbeitet werden sollen 1. Vorsortierung

---

Types

- type Person = String ← Always uppercase
- type Book = String
- type Database = [(Person, Book)] (List, Tuple)

- Which books has person Jonas borrowed?

```
books:: Database -> Person -> [Book]

books db p = [bk | (per,bk) <- db, per == p]
```

Induction

1. Prove P(n) for all natural numbers n
2. Base case: prove P(0)
3. Induction hypothesis: P(n) holds
4. Step case: To prove P(n+1) with help of I.H.

Induction over lists

1. Prove P(xs) for all xs in [T] (xs :: [T])
2. Base case: prove P([]) (empty list)
3. Induction hypothesis: P(xs) holds
4. Step case: To prove P(x:xs) with help of I.H.

Polymorphism

- length :: [t] -> Int
  - ↳ sign for polymorphism, because [t] can be a list of any type t

---

Higher order functions

- First order: · Arguments are base types
  · Int -> Int
- Second order: · Arguments can be themselves functions
  · (Int -> Int) -> [Int]
- Third order: · Arguments may be functions, whose arguments are functions

  Function

  · ((Int -> Int) -> Int) -> [Int]

  Function

Map

- Takes a list and return a list
- map :: (a -> b) -> [a] -> [b]
  (a -> b): function; [a]: inserted list; [b]: returned list
- map f [] = []
- map f (x:xs) = f x : map f xs
- times2 x = 2*x / times2 :: Int -> Int
- map times2 [1,2,3] = [2,4,6] } Example
- Similar to comprehension: map f (x:xs) ≙ [f x | x ← xs]

---

foldr

- Takes a list and returns a folded number

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

foldr f z []     = z
foldr f z (x:xs) = f x (foldr f z xs)

sumList xs = foldr (+) 0 xs
```

} f x y : f x (foldr ...)
x, y: curr, next; x, y

- f takes as input two numbers, applies the function (e.g addition) and returns one number
- z is the base case, e.g. 0
- foldr (+) 0 [1,2,3]
= (+) 1 (foldr (+) 0 [2,3])
= (+) 1 ((+) 2 (foldr (+) 0 [3]))
= (+) 1 ((+) 2 ((+) 3 (foldr (+) 0 [])))
= (+) 1 ((+) 2 ((+) 3 0))
= (+) 1 ((+) 2 (3 + 0))
= (+) 1 ((+) 2 3)
= (+) 1 (2 + 3)
= (+) 1 5
= 1 + 5 = 6

Difference foldr / foldl

- foldr: right associative
  - ↳ foldr f z (x:xs) = f x (foldr f z xs)
- foldl: left associative
  - ↳ foldl f z (x:xs) = foldl f (f z x) xs
    - (f z x): this gets the new z value
- only makes a difference, if function is not associative

Reverse

```
rev []     = []
rev (x:xs) = rev xs ++ [x]
```

Concat

- Unifies several lists to one

```
concat xs = foldr (++) [] xs
? concat [[1,2,3],[4],[5,6]]
[1,2,3,4,5,6] :: [Int]
```

λ-expression

- Haskell provides notation to write functions inline

```
times2 x = 2*x
map times2 [1,2,3]
```

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

- Also supports lists `\x xs -> xs ++ [x]`

Function composition

- `(.) :: (b -> c) -> (a -> b) -> (a -> c)`
  - (b -> c): f(x)
  - (a -> b): g(x)
  - (a -> c): f(g(x))
- `(f . g) x = f (g x)`

functions can be returned as functions

Functions as values

- A function can be returned
- But functions cannot be displayed, but they can be applied

Arguments of functions

- Each function has only one argument/return-value
- multiply :: Int -> Int -> Int means multiply :: Int -> (Int -> Int)
- multiply 2 3 means (multiply 2) 3
- -> is right-associative
- Example: g :: Int -> Int -> Int
  g x y = x * y + 17

  g :: Int -> (Int -> Int)
  g x = \y -> x * y + 17

Curry / uncurry

```
curry f    = f' where f' x1 x2  = f (x1,x2)
uncurry f' = f  where f (x1,x2) = f' x1 x2
```

- f (x,y) = x * y + 17
  - Normal: f (3,4)
  - Curry: curry f 3 4
- g x y = x * y + 17
  - Normal: f 3 4
  - Uncurry: uncurry f (3,4)

Haskell div

- 366 `div` 10 = 36
  - ↳ div rounds to zero
  - ↳ div :: Int -> Int -> Int

Induction prove structure

1. Let P(n) := .....
2. We show ∀n : Nat. ..... by induction
3. Base case: show P(0)
4. Step case: · Let n : Nat be arbitrary
   · show that P(n) implies P(n+1)
   · We assume that the induction hypothesis
   e.g. aux n = fibonacci n, fi...
   · now, we show P(n+1)

Data modeling

- Algebraic data types
  · Declare new types
- Enumeration types
  · Type being a finite set of elements
  · data Season = Spring | Summer | Fall | Winter
  · data Month = January | February | ...
  ↳ whichSeason :: Month -> Season
- Product type
  · data People = Person Name Age ← constructor
  · type People = (Name, Age) } Type-Tuples
- Enumeration + Product types
  · data Shape = Circle Double | Rectangle Int Int
- Integration with classes
  · data Foo = D1 | D2 | D3
    deriving (Eq, Ord, Enum, Show)

- Recursive types
  - data Expr a = Lit Int | Add (Expr a) (Expr a) | Sub (Expr a) (Expr a)
    ↳ 2 + 3 ≙ Add (Lit 2) (Lit 3)
- Integration with classes for e.g. Eq
  - instance Eq Foo where
    D 1 == D 1 = True
    _ == _ = False

Lazy Evaluation

- Expressions evaluated only when necessary
- Some expressions may never be evaluated, this can save large amounts of time
- Duplicated expressions are only evaluated once, and then shared
- Arguments evaluated as far as needed to determine pattern match
- Functions are evaluated top-down
  ↳ outermost operator first
- Lazy evaluation enables finite representation of infinite data
  ↳ addFirstTwo (x : xx : _)
  addFirstTwo

Prime Numbers

- sieve (x : xs) = x : (sieve (filter (\a -> a `mod` x /= 0) xs)
  ↳ sieve [2..]
- isPrime n = [x | x ← [1..n], x `mod` n = 1] == [1,n]

Mini-Haskell

$$\frac{}{\ldots, x : \tau, \ldots \vdash x :: \tau}\ \textit{Var} \qquad \frac{\Gamma, x : \sigma \vdash t :: \tau}{\Gamma \vdash \lambda x.\, t :: \sigma \to \tau}\ \textit{Abs}$$

$$\frac{\Gamma \vdash t_1 :: \sigma \to \tau \qquad \Gamma \vdash t_2 :: \sigma}{\Gamma \vdash t_1\ t_2 :: \tau}\ \textit{App} \qquad \frac{\Gamma \vdash t :: Int}{\Gamma \vdash \textbf{iszero}\ t :: Bool}\ \textit{iszero}$$

$$\frac{}{\Gamma \vdash n :: Int}\ \textit{Int} \qquad \frac{}{\Gamma \vdash True :: Bool}\ \textit{True} \qquad \frac{}{\Gamma \vdash False :: Bool}\ \textit{False}$$

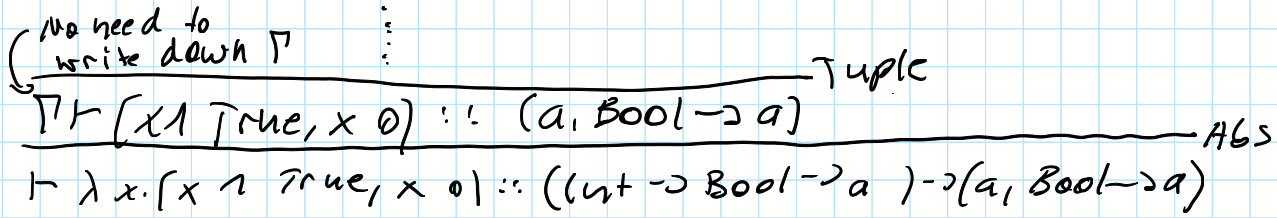
$$\frac{\Gamma \vdash t_1 :: Int \qquad \Gamma \vdash t_2 :: Int}{\Gamma \vdash (t_1\ \textbf{op}\ t_2) :: Int}\ \textit{BinOp} \qquad \text{for } \textbf{op} \in \{+, *\}$$

$$\frac{\Gamma \vdash t_0 :: Bool \qquad \Gamma \vdash t_1 :: \tau \qquad \Gamma \vdash t_2 :: \tau}{\Gamma \vdash \textbf{if}\ t_0\ \textbf{then}\ t_1\ \textbf{else}\ t_2 :: \tau}\ \textit{if}$$

$$\frac{\Gamma \vdash t_1 :: \tau_1 \qquad \Gamma \vdash t_2 :: \tau_2}{\Gamma \vdash (t_1, t_2) :: (\tau_1, \tau_2)}\ \textit{Tuple} \qquad \frac{\Gamma \vdash t :: (\tau_1, \tau_2)}{\Gamma \vdash \textbf{fst}\ t :: \tau_1}\ \textit{fst} \qquad \frac{\Gamma \vdash t :: (\tau_1, \tau_2)}{\Gamma \vdash \textbf{snd}\ t :: \tau_2}\ \textit{snd}$$

Most general type

- Use Mini-Haskell to derive
- Achtung: Bei z.B (+) oder (>) nicht vergessen
  - num a => ....
  - ord a => ....
- Wenn man statement beweisen muss:



Structural induction for Trees

$$\frac{\Gamma \vdash P(\mathit{Leaf}) \quad \Gamma, P(l), P(r) \vdash P(\mathit{Node}\ a\ l\ r)}{\Gamma \vdash \forall x \in \mathit{Tree}\ t.\ P(x)}\ a, l, r \text{ not free in } \Gamma \text{ or } P$$

- P(t) := ...
- Base case P(Leaf)
- Step case P(Node x l r): · P(l) and P(r) holds } I.H.

Parser

```
data Expr = Lit Int | Add Expr Expr
                    | Sub Expr Expr

eval :: Expr -> Int

eval (Lit n)      = n
eval (Add e1 e2) = (eval e1) + (eval e2)
eval (Sub e1 e2) = (eval e1) - (eval e2)
```

- Mutual selection: p ||| q = prs (\s -> parse p s ++ parse q s
  ↳ Apply both parsers
- Alternative selection: p +++ q = prs (\s -> case parse p s of
  [] -> parse q s
  res -> res)
  ↳ If first parser fails, try second
- Sequencing: first parser p then parser q to result

Lazy evaluation for applications

f g = [λ x -> x (λ y -> x y)] [λ x -> (λ y -> y) x]
= [λ x -> (λ y -> y) x] (λ y -> [λ x -> (λ y -> y) x] y)
= (λ y -> y) (λ y -> [λ x -> (λ y -> y) x] y)
= (λ y -> [λ x -> (λ y -> y) x] y)
↳ Nur so lange substituieren, bis nur noch eine Klammer da steht
↳ Nicht innerhalb von Klammern substituieren

Eager evaluation for application

f g = [λx -> x (λy -> x y)] [λx -> (λy -> y) x]
= [λx -> x (λy -> x y)] [λx -> x]
= [λx -> x] (λy -> [λx -> x] y)
= [λx -> x] (λy -> y)
= [λy -> y]

↳ solange substituieren, bis man nicht mehr kann
↳ von links nach rechts einsetzen

Give the fold function for the following data-type

- data G a b = A int | B (G a b) Bool (G a b) | C a b

A ist ein Constructor, welcher ein Int nimmt, und ein G a b zurückgibt

Jeder Constructor gibt ein G a b zurück

- A :: int -> G a b
- B :: (G a b) -> Bool -> (G a b) -> (G a b)
- C :: a -> b -> (G a b)
- G a b = C (unsere Umwandlungsfunktion)
- f A :: int -> c
- f B :: c -> Bool -> c -> c
- f C :: a -> b -> c

↳ foldG :: (int -> c) -> (c -> Bool -> c -> c) -> (a -> b -> c) -> G a b -> c

(int -> c): A; (c -> Bool -> c -> c): B; (a -> b -> c): C; G a b: input; c: output

```
foldG fA fB fC input = aux input
    where
        aux (A number) = fA number
        aux (B g1 b g2) = fB (aux g1) b (aux g2)
        aux (C a b) = fC a b
```

Monads

```
data Maybe a = Nothing | Just a

safeDiv :: Int -> Int -> Maybe Int
safeDiv n d
  | d /= 0    = Just (n `div` d)
  | otherwise = Nothing
```

Instantiation

```
instance (Show a) => Show (F a)
        where
            show = foldF (\a -> show a) (\s -> "(Not " ++ s ++ ")") (..)(..)
```