

Haskell on ...

- tugevalt ning staatiliselt tüübitud,
- laisk ja puhas
- funktsionaalne keel.

Funktsionaalne keel

- Ilma funktsioonideta (meetodite, protseduurideta) ei saa!

Java: `Math.max(4, 7)`

Python: `max(4, 7)`

Haskell: `max 4 7`

- Sõna "funktsioon" asemel kasutatakse ka *abstraktsioon*.
- FP eelistab olemasolevate vahendite üldistust.
Näiteks *if*-lause saame ise defineerida:

```
kui True   t f = t  
kui False t f = f
```

- Turingi masin: programm on staatiline ja andmed dünaamilised
- FP: Samamoodi, kuidas programmeerimiskeeles saab käsitleda andmeid, peab saama käsitleda ka alamprogramme.
- $\implies$ kõrgemat järku funktsioonid
`map (max 3) [2,3,4]  $\rightsquigarrow$  [max 3 2, max 3 3, max 3 4]  $\rightsquigarrow$  [3,3,4]`
- Kõrgemat järku funktsioonid võimaldavad meil programmi osadest mõelda kõrgemal abstraktsiooni tasemel ehk siis suuremate tükkidena.

Tugevalt ja staatiliselt tüübitud

- Mida rohkem programmeerimiskeel lubab seda parem?
 - PostScript (1982) vs. Portable Document Format (1993)
- Piiramise üks võimalus on *tüübisüsteemiga*.
- Staatiline tüübikontroll — kompileerimise (või interpreteerimise) käigus
- Tugevalt tüübitud keele puhul väljastatakse kohe veateade.
- Nõrgalt tüübitud keele võib püüda näiteks sõnet arvuks teisendada.
- Testimise vajadus mingil määral väiksem?

Puhas keel

- Puhas — kõrvaltoimevaba ehk funktsiooni kutse tulemus sõltub ainult parameetrite väärtustest.
- $\implies$ iga funktsiooni saab testida teistest eraldi
- mittepuhtaid funktsioone (nagu juhuarvude genereerimine) pole võimalik funktsioonidena defineerida
- $\implies$ on vaja kasutada *monaade*

Laisk keel

Olgu meil selline kood:

```
tagastaViis x = 5  
topelt x = x+x
```

- *agaras keeles* arvutatakse parameeter enne funktsiooni kutset

`tagastaViis (1+1)  $\rightsquigarrow$  tagastaViis 2  $\rightsquigarrow$  5`

`topelt (1+1)  $\rightsquigarrow$  topelt 2  $\rightsquigarrow$  2+2  $\rightsquigarrow$  4`

- *laisas keeles* tehakse funktsioonikutse asendus enne

`tagastaViis (1+1)  $\rightsquigarrow$  5`

`topelt (1+1)  $\rightsquigarrow$  (1+1)+(1+1)  $\rightsquigarrow$  2+(1+1)  $\rightsquigarrow$  2+2  $\rightsquigarrow$  4`

- Haskell teeb tegelikult hoopis nii

`topelt (1+1)  $\rightsquigarrow$  x+x where x=1+1  $\rightsquigarrow$  x+x where x=2 == 2+2  $\rightsquigarrow$  4`

Infix Operaatorid I

- Infix operaatorid (näiteks +) ja -tüübid (näiteks ->) kirjutatakse argumentide vahele, mitte argumentide ette.
 - Näiteks: `5 + 2`, `2*pi*r^2`, `Float -> Int`
- Infixoperaatori kasutamiseks prefix-vormis tuleb see panna sulgudesse. Näiteks: `3 + 4 == (+) 3 4`
- Infix operaatoreid saab ise juurde defineerida:
| `(@) :: Int -> Int -> Int`
| `x @ y = 2*x+y`
või
| `(@) :: Int -> Int -> Int`
| `(@) x y = 2*x+y`
- Mõnikord eeldame, et funktsioonirakendused on prefixsed.

Infix Operaatorid II

- Infix operaatoril on prioriteet (i.k. precedence)
 - Näiteks: $3*4+5 == (3*4)+5$, kuna $*$ on tasemel 7 ja $+$ tasemel 6
- Infix operaatoril võib olla assotsiatiivsus
 - näiteks, $2**3**4 == 2**(3**4)$, $2 - 3 - 4 == (2-3)-4$

- Neid seatakse nn. fixity deklaratsioonidega:

```
infixl 7 *  
infixl 6 +  
infixl 6 -  
infixr 8 **
```

- Fixity deklaratsioone saab vaadata ghci abil:
 - Näiteks käsuga “:i (+)”
- Prefix operaatoreid saab rakendada infix-selt tagurpidi-ülakomade (`) vahel
 - Näiteks: $5 \text{ `div` } 2$

Eeldefineeritud operaatorite fixity (Haskell 98)

Prec- edence	Left associative operators	Non-associative operators	Right associative operators
9	!!		.
8			^, ^^, **
7	*, /, 'div', 'mod', 'rem', 'quot'		
6	+, -		
5			:, ++
4		==, /=, <, <=, >, >=, 'elem', 'notElem'	
3			&&
2			
1	>>, >>=		
0			\$, \$!, 'seq'

(Funktsioonirakenduse *pretsedence* on 10.)

Ennikud ja *Unit* tüüp

Olemasolevatest tüüpidest saab luua paare ja muid ennikuid:

- `(1, 'a')`, `((1.1, 8, 'x'), False)`
- enniku tüüp konstrueeritakse komponentide tüüpidest
`(1, 'a') :: (Int, Char)`
- paarides olevat info saab kätte mustrisobitusega:

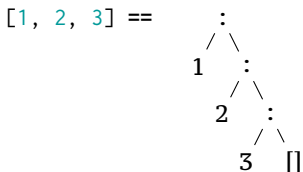
```
| f :: (Int, Char, String) -> Int  
| f (x,c,ys) = x + 1
```

Kõige triviaalsem väärtus Haskellis on `()`.

- Selle tüüp on samuti `()` ehk `() :: ()`
- Kasutatakse juhtudel, kui pole vaja informatsiooni edastada.
- Paljudes keeltes kasutatakse tüüpi `void`

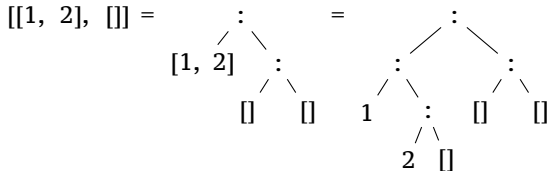
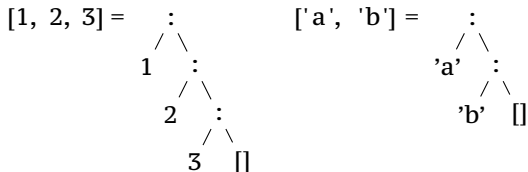
Järjendid e. listid

- Järjendi tüübiks on “[a]”, kus “a” on järjendi elementide tüüp.
 - Näiteks: [Int], [Char], [[Float]]
- Järjendeid saab kirjutada nii (tüüpide kirjutamine vabatahtlik):
 - [1, 2, 3] :: [Int], [3] :: [Int], [] :: [Int]
 - [True, False, False] :: [Bool], [False] :: [Bool], [] :: [Bool]
- Haskelli listid on arvuti mälus esindatud puudena:



- Järjendi loomiseks on kaks konstruktorit, mis vastavad (list-tüüpi) puu tippudele.
 - [] :: [a]
 - (:) :: a -> [a] -> [a]
 - Näide: [3,2,1] == 3 : 2 : 1 : []

Listi näited



Järjendi e. listid

- Konstrueerimise vastand on mustrisobitus. Nii saame vaadata, millise konstruktoriga antud väärtus loodi.

```
length []      = 0
length (x:xs) = 1 + length xs
```

või

```
length xs =
  case xs of
    []      -> 0
    (x:xs) -> 1 + length xs
```

- Mustreid sobitatakse "ülevallt all", kuni esimese sobiva leidmiseni!
- Arvutuskäik:

```
length [1,2,3]
== length (1 : (2 : (3 : [])))
==> 1 + length (2 : (3 : []))
==> 1 + (1 + length (3 : []))
==> 1 + (1 + (1 + length []))
==> 1 + (1 + (1 + 0))
==> 3
```

Listide näited

- Korrutis

```
product [] = 1
product (x:xs) = x * product xs
```

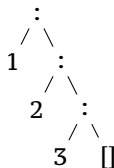
- Listi ümberpööramine

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

- Järjest asetsevate võrdsete elementide loendamine

```
countEq (x:y:xs)
  | x==y      = 1 + countEq (y:xs)
  | otherwise = countEq (y:xs)
countEq _     = 0
```

Tähelepanekud



- **Elemendi lisamine listi algusse väga kiire!**
 - kiire == konstante aeg ja mälu
 - Põhjendus: argumente ei ole vaja kopeerida.
- **Viimase elemendi selekteerimine aeglane!**
 - aeglane == lineaarselt listi pikkusega
- **Järjendi lõppu lisamine aeglane!**
 - aeglane == aeg ja mälu lineaarne listi pikkusega (tuleb kopeerida)
 - Laiks väärtustamine päästab mõnel juhul!
- $\implies$ Listid Haskellis on kasutatavad iteraatoritena

Loe lisaks: RWH, lk 10..12, 23..26; LYaH, peatükk 2 lõpp

Anonüümsed funktsioonid e. lambdad

- Defineerides funktsiooni

```
█ f x y z = ...
```

teisendab kompilaator koodi selliseks

```
█ f = \ x y z -> ...
```

- Langjoon tähistab lambda arvutuses kasutatavat λ -sümbolit.
- Sellist süntaksit saab ka ise kasutada.
 - Näiteks, funktsiooni argumendina:

```
█ map (\ x -> x+1) [1,2,3,4]
```

Laisk väärtustamine

programm: rida deklaratsioone. Näiteks:

```
double x = x + x  
main = double (1+1)
```

redex: redutseeritav avaldis – programmis olev avaldis, mida saab lihtsustada. Näiteks: 1+1

kontekst: kõik definitsioonid, mis kehtivad redex-i asukohas.

Näited ülevaloleva programmi kohta:

- 1+1 puhul on kontekstis kogu programm, kuid neid definitsioone ei lähe vaja.
- järgmisel sammul, kui redexiks $x + x$ on selle näite puhul kontekstis ka $x = 2$
- double (1+1) puhul on kontekstis kogu programm, kuid vaja läheb vaid double definitsiooni.

Laisk väärtustamine

Lihtsustada saab

- ① sisseehitatud operaatoreid, aga ainult siis, kui argumendid on juba normaalkujul.
 - $1 + x$ ei saa lihtsustada
 - $1 + 1$ lihtsustub avaldiseks 2
- ② nimesid, mis on kontekstis defineeritud
 - x lihtsustub väärtuseks $y+1$, kui kontekstis on $x = y+1$
- ③ argumendi rakendamist lambda-avaldisele
 - $(\lambda x \rightarrow x+x) \ 5$ lihtsustub avaldiseks $5+5$
- ④ funktsioonirakendust

$$f \ e_1 \ \dots \ e_n \quad (\text{kus } f \ x_1 \ \dots \ x_n = b)$$

$$\Downarrow$$

$$b[x_1 \rightarrow e_1][x_2 \rightarrow e_2] \ \dots \ [x_n \rightarrow e_n]$$

Näiteks: `double (1+1)` lihtsustub avaldiseks `(1+1)+(1+1)`

- Avaldis on *normaalkujul*, kui teda ei saa enam lihtsustada!

"Vabad" muutujad

- ... on muutujad, mis pole seotud lambdaga.
 - $FV((\lambda x \rightarrow y+x+1) z) = \{y, z\}$
 - $FV((\lambda q \rightarrow y+x+1) z) = \{x, y, z\}$
 - $FV((\lambda x \rightarrow x) x) = \{x\}$
- Seotud muutujate nimed pole olulised — α -teisendus.
 - $(\lambda x \rightarrow x + 1) = (\lambda a \rightarrow a + 1)$
- $FV(x) = \{x\}$
- $FV(c) = \emptyset$
- $FV(t_1 t_2) = FV(t_1) \cup FV(t_2)$
- $FV(\lambda x \rightarrow t) = FV(t) \setminus \{x\}$

Asendamine: $e[v \rightarrow e']$

muutujad: $x[y \rightarrow t] = \begin{cases} t & \text{kui } x \doteq y \\ x & \text{muidu} \end{cases}$

konstandid: $c[y \rightarrow t] = c$

rakendus: $(t_1 \ t_2)[y \rightarrow t] = t_1[y \rightarrow t] \ t_2[y \rightarrow t]$

lambda: $(\lambda x \rightarrow t_1)[y \rightarrow t] = \begin{cases} (\lambda x \rightarrow t_1) & \text{kui } x \equiv y \\ (\lambda z \rightarrow e[x \rightarrow z])[y \rightarrow t] & \text{kui } x \in \text{FV}(t) \\ \lambda x \rightarrow t_1[y \rightarrow t] & \text{muidu} \end{cases}$

kus z on "värske" muutuja

Näide

Asendus tuleb teha reeglite järgi, et vältida muutujate püüdmist (i.k *variable capture*)!

$$\begin{aligned}(\lambda x y \rightarrow x+y) y &= (\lambda x \rightarrow \lambda y \rightarrow x+y) y \\&\rightsquigarrow (\lambda y \rightarrow x+y)[x \rightarrow y] \\&\rightsquigarrow \textcolor{red}{(\lambda y \rightarrow y+y)} \\&\rightsquigarrow \textcolor{green}{(\lambda z \rightarrow x+z)[x \rightarrow y]} \\&\rightsquigarrow \textcolor{green}{(\lambda z \rightarrow y+z)}\end{aligned}$$

Redutseerimise järjekord I

Redutseerimiseks valida kõige välimisem avaldis. Kui välist avaldist ei saa redutseerida, võtame ette selle alamavaldised, suunaga vasakult-paremale.

Programm:

```
double x = x + x  
main = double (1+1)
```

Reduktsioon:

```
main  $\rightsquigarrow$  double (1+1)  
       $\rightsquigarrow$  (1+1) + (1+1)  
       $\rightsquigarrow$  2 + (1+1)  
       $\rightsquigarrow$  2 + 2  
       $\rightsquigarrow$  4
```

Redutseerimise järjekord II

```
fact 0 = 1  
fact x = x * fact (x-1)
```

- Kui lihtsustamist takistab mustrisobitus, redutseerime sobitatavat avaldist seni, kuni saame otsustada, milline juht valida.

Reduktsioon:

```
fact 2  $\rightsquigarrow$  2 * fact (2-1)  
            $\rightsquigarrow$  2 * fact 1  
            $\rightsquigarrow$  2 * (1 * fact (1-1))  
            $\rightsquigarrow$  2 * (1 * fact 0)  
            $\rightsquigarrow$  2 * (1 * 1)  
            $\rightsquigarrow$  2 * 1  
            $\rightsquigarrow$  2
```

- Seda järjekorda saab küll formaliseerida (konspektis), aga soovitame lähtuda intuitsioonist.

Kõrgemat järku funktsioon

... on funktsioon, mis võtab argumendiks (või tagastab) funktsiooni.

Näiteks

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

Funktsiooni map illustreerib järgmine võrdus:

$$\text{map } f [x_1, x_2, \dots, x_n] = [f x_1, f x_2, \dots, f x_n]$$

FP keskne mõte: arvutada saab ka arvutustega (e. funktsioonidega).

Näiteks map saab interpreteerida kui $(a \rightarrow b) \rightarrow ([a] \rightarrow [b])$

Näide

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

```
map (+1) [1,2,3] = map (+1) (1:2:3:[])
  ~> (1+1) : map (+1) (2:3:[])
  ~> 2 : map (+1) (2:3:[])
  ~> 2 : (2+1) : map (+1) (3:[])
  ~> 2 : 3 : map (+1) (3:[])
  ~> 2 : 3 : (3+1) : map (+1) []
  ~> 2 : 3 : 4 : map (+1) []
  ~> 2 : 3 : 4 : []
  = [2, 3, 4]
```