

## 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`

- Infixoperaatoreid saab ise juurde defineerida:

```
| (⓪) :: Int -> Int -> Int
| x ⓪ y = 2*x+y
```

või

```
| (⓪) :: Int -> Int -> Int
| (⓪) x y = 2*x+y
```

- Mõni kord eeldame, et funktsioonirakendused on prefixsed.

## Infix Operaatorid II

- Infixoperaatoril on prioriteet (i.k. precedence)
  - Näiteks:  $3*4+5 == (3*4)+5$ , kuna  $*$  on tasemel 7 ja  $+$  tasemel 6
- Infixoperaatoril 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-<br>edence | Left associative<br>operators        | Non-associative<br>operators               | Right associative<br>operators |
|-----------------|--------------------------------------|--------------------------------------------|--------------------------------|
| 9               | !!                                   |                                            | .                              |
| 8               |                                      |                                            | ^, ^^, **                      |
| 7               | *, /, 'div',<br>'mod', 'rem', 'quot' |                                            |                                |
| 6               | +, -                                 |                                            |                                |
| 5               |                                      |                                            | :, ++                          |
| 4               |                                      | ==, /=, <, <=, >, >=,<br>'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:

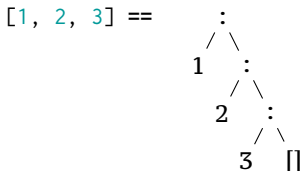
```
f :: (Int, Char, String) -> Int  
f (x,c,ys) = x + 1
```

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

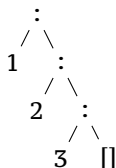
- Järjendi tüübiks on “[a]”, kus “a” on järjendi elementide tüüp.
  - Näiteks: [Int], [Char], [[Float]]
- Järjendeid saab kirjutada nii (tüüpide kirjutamine vabatahtlik):
  - [1, 2, 3] :: [Int], [3] :: [Int], [] :: [Int]
  - [True, False, False] :: [Bool], [False] :: [Bool], [] :: [Bool]
- Haskelli listid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



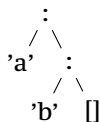
- Järjendi loomiseks on kaks konstruktorit, mis vastavad (list-tüüpi) puu tippudele.
  - [] :: [a]
  - (:) :: a -> [a] -> [a]
  - Näide: [3,2,1] == 3 : 2 : 1 : []

## Listi näited

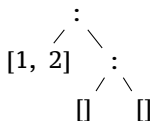
[1, 2, 3] =



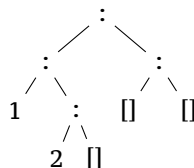
['a', 'b'] =



[[1, 2], []] =



=



## 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 "ülevalt 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
```

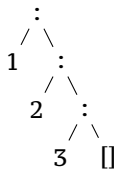
- Listi ümberpööramine

```
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  $\lambda$ -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

$f \ e_1 \ \dots \ e_n$  (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!