- Näidiste sobitamisel põhinev faktoriaal

```
fact2 :: Int -> Int
fact2 0 = 1
fact2 n = n * fact2 (n-1)
```

- Valvuritel põhinev faktoriaal ...

```
fact3 :: Int -> Int
fact3 n
  | n==0      = 1
  | otherwise = n * fact3 (n-1)
```

- Valvuritel põhinev faktoriaal mis ei lähe tsükklisse

```
fact4 :: Int -> Int
fact4 n
  | n == 0 = 1
  | n >= 1 = n * fact4 (n-1)
```

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:

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

  või

```
(@) :: Int -> Int -> Int
(@) 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 `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
```

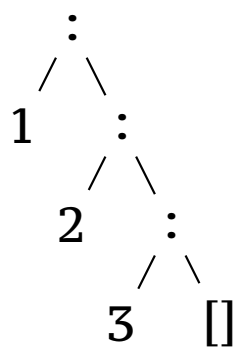
Kõige triviaalsem väärtus Haskellis on `()`.

- 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:

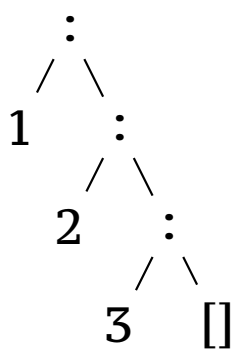
`[1, 2, 3] ==`



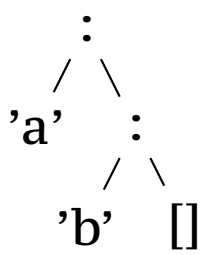
- Järjendi loomiseks on kaks konstruktorit, mis vastavad (list-tüüpi) puu tippudele.
  - `[] :: [a]`
  - `(:) :: a -> [a] -> [a]`
  - Näide: `[3,2,1] == 3 : 2 : 1 : []`

Listi näited

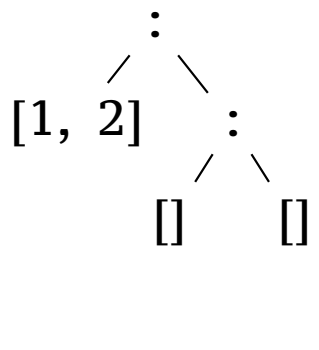
[1, 2, 3] =



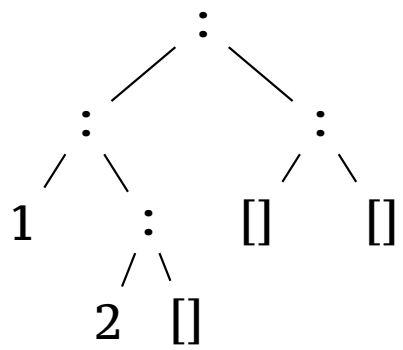
['a', 'b'] =



[[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
```

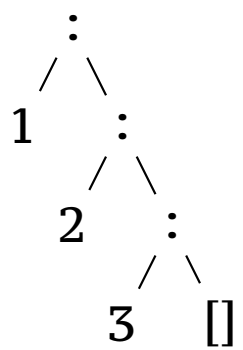
- Listi ümberpööramine

```
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!
- $\Longrightarrow$ 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
   - `(\x -> x+x) 5` lihtsustub avaldiseks `5+5`
4. funktsioonirakendust

   `f` e_1 `...` e_n (kus `f` x_1 `...` x_n `= b`)

   $\Downarrow$

   `b[`$x_1 \to e_1$`][`$x_2 \to e_2$`] ... [`$x_n \to e_n$`]`

   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((λ x → y+x+1) z) = {y, z}
  - FV((λ q → y+x+1) z) = {x, y, z}
  - FV((λ x → x) x) = {x}
- Seotud muutujate nimed pole olulised — α-teisendus.
  - (λ x → x + 1) = (λ a → a + 1)
- $\mathrm{FV}(x) = \{x\}$
- $\mathrm{FV}(c) = \emptyset$
- $\mathrm{FV}(t_1\ t_2) = \mathrm{FV}(t_1) \cup \mathrm{FV}(t_2)$
- $\mathrm{FV}(\lambda\, x \to t) = \mathrm{FV}(t) \setminus \{x\}$

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

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

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

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

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

kus z on "värske" muutujanimi

Näide

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

$$
\begin{aligned}
(\lambda\ \mathtt{x}\ \mathtt{y} \to \mathtt{x+y})\ \mathtt{y} &= (\lambda\ \mathtt{x} \to \lambda\ \mathtt{y} \to \mathtt{x+y})\ \mathtt{y} \\
&\rightsquigarrow (\lambda\ \mathtt{y} \to \mathtt{x+y})[\mathtt{x} \to \mathtt{y}] \\
&\rightsquigarrow \cancel{(\lambda\ \mathtt{y} \to \mathtt{y+y})} \\
&\rightsquigarrow (\lambda\ \mathtt{z} \to \mathtt{x+z})[\mathtt{x} \to \mathtt{y}] \\
&\rightsquigarrow (\lambda\ \mathtt{z} \to \mathtt{y+z})
\end{aligned}
$$

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:

$$
\begin{aligned}
\texttt{main} &\rightsquigarrow \texttt{double (1+1)} \\
&\rightsquigarrow \texttt{(1+1) + (1+1)} \\
&\rightsquigarrow \texttt{2 + (1+1)} \\
&\rightsquigarrow \texttt{2 + 2} \\
&\rightsquigarrow \texttt{4}
\end{aligned}
$$

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:

$$
\begin{aligned}
\texttt{fact 2} &\rightsquigarrow \texttt{2 * fact (2-1)} \\
&\rightsquigarrow \texttt{2 * fact 1} \\
&\rightsquigarrow \texttt{2 * (1 * fact (1-1))} \\
&\rightsquigarrow \texttt{2 * (1 * fact 0)} \\
&\rightsquigarrow \texttt{2 * (1 * 1)} \\
&\rightsquigarrow \texttt{2 * 1} \\
&\rightsquigarrow \texttt{2}
\end{aligned}
$$

- Formaalselt on järjekord defineeritud funktsiooniga fookus ja kaalutlus. (konspektist)

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:

$$\texttt{map}\ f\ [x_1, x_2, \ldots, x_n] = [f\ x_1, f\ x_2, \ldots, f\ x_n]$$

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

Näiteks `map` saab interpreteerida kui `(a -> b) -> ([a] -> [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]
```