Kõigepealt

- COVID19-kohalolekukontroll: `http://cs.ut.ee/reg`
- Avalik veeb: `https://courses.cs.ut.ee/2020/PK`
  - loengute materjalid
  - praktikumide ja kodutööde materjalid
- Kursuse privaat veeb: Moodle
  - kodutööde esitamine
  - loengu- ja baastestid, tööde esitamine

Hinde kujunemine

- Protsendiskaala 0p .. 100p
- 0p..50p $\rightarrow$ F, 51p..60p $\rightarrow$ E, 61p..70p $\rightarrow$ D, ...
  - eeldusel, et baastestid on arvestatud
  - ümardame üles (70.1 -> 71 -> C)
- Eksam (50p)
- Loengutestid (10$*$1p, moodle)
- Kodutööd (40p, moodle)
- Kodutööde, loengutestide asemel soovitatav teha oma projekt.
  - Programmeerida midagi Haskellis ja/või Scalas.
- Haskelli baastest ja Scala baastest

Materjalid

1. konspekt: `http://kodu.ut.ee/~kalmera/haskell/`

2. “Learn You a Haskell for Great Good!” (2011)
   - \+ Väga hea raamat intuitsiooni saamiseks
   - \- Pole põhjalik ega täielik

3. “Real World Haskell” (2008)
   - \+ Põhjalik ja selge!
   - \- Mahukas!

4. “Sissejuhatus Funktsionaalsesse Programmeerimisse” (2010)
   - \+ Eestikeelne
   - \- Osati liiga detailne, osati liiga piiratud

(Haskelli on vahepeal uuendatud! Raamatud on kohati vananenud!)

Programmeerimiskeeled

Kursuse läbinu:

- omandab ülevaatlikud teadmised erinevatest programmeerimise paradigmadest;
- oskab lahendada lihtsaid programmeerimise ülesandeid funktsionaalse programmeerimise abil, sealhulgas saab aru ja oskab kasutada kõrgemat järku polümorfseid funktsioone;
- oskab mitme-paradigma keeles kasutada koos funktsionaalse, imperatiivse ja objekt-orienteeritud paradigma tehnikaid.

Administrativia

- Loengud
  - T10-12, 1019, Kalmer Apinis (kalmera@ut.ee)
- Praksid
  - K10-12, 1019, Simmo Saan
- Loe pikemalt: courses.cs.ut.ee/2020/PK

Miks uued programmeerimiskeeled?

- Keel C on vähemalt sama võimas, ükskõik mis teine programmeerimiskeel! (järeldub Church-Turingi teesist)

Miks uued programmeerimiskeeled?

- Keel C on vähemalt sama võimas, ükskõik mis teine programmeerimiskeel! (järeldub Church-Turingi teesist)

- Vastus: komponentide ja algoritmide taaskasutus! Keegi ei kirjuta programme “nullist”! Mida lihtsam on erinevaid teeke omavahel ühendada, seda parem.

Miks uued programmeerimiskeeled?

- Keel C on vähemalt sama võimas, ükskõik mis teine programmeerimiskeel! (järeldub Church-Turingi teesist)
- Vastus: komponentide ja algoritmide taaskasutus! Keegi ei kirjuta programme “nullist”! Mida lihtsam on erinevaid teeke omavahel ühendada, seda parem.
- Vastus: korrektsuse tõestamise võimalus. Meil on palju koodi aga me ei saa seda usaldada.
- otsime abi teoreetikutelt . . .

Motivatsioon

Taaskasutusel on abiks

- paindlikud modulaarsuse võimalused

Motivatsioon

Taaskasutusel on abiks

- paindlikud modulaarsuse võimalused

Korrektsuse tõestamisel on abiks

- Keel on täpselt defineeritud.
- Range tüübisüsteem.
- Ilmutatud viidatavus ehk saame ignoreerida ebaolulist.
  - Puhas FP – funktsiooni tagastusväärtus sõltub ainul argumentidest.
- Baaskonstruktsioonide hulk on väike.

Motivatsioon

Taaskasutusel on abiks

- paindlikud modulaarsuse võimalused

Korrektsuse tõestamisel on abiks

- Keel on täpselt defineeritud.
- Range tüübisüsteem.
- Ilmutatud viidatavus ehk saame ignoreerida ebaolulist.
  - Puhas FP – funktsiooni tagastusväärtus sõltub ainul argumentidest.
- Baaskonstruktsioonide hulk on väike.

$\Longrightarrow$ Õpime Funktsionaalset Programmeerimist!

Funktsionaalne Programmeerimine (FP)

- FP motivatsioon
  - Probleem: võimalused on välja arendamata ja üksteisega konfliktis
  - Võimaluste lisamise asemel peaks üldistama olemasolevaid!
  - Aritmeetilised operatsioonid (funktsioonid) on möödapääsmatud!

Funktsionaalne Programmeerimine (FP)

- FP motivatsioon
  - Probleem: võimalused on välja arendamata ja üksteisega konfliktis
  - Võimaluste lisamise asemel peaks üldistama olemasolevaid!
  - Aritmeetilised operatsioonid (funktsioonid) on möödapääsmatud!
- Tüüpimise motivatsioon
  - Probleem: Harva(?) esinevad vead erijuhtudest.
  - Võimaldab vältida vigu ja anda edasi kompileerimisaegset infot.

Funktsionaalne Programmeerimine (FP)

- FP motivatsioon
  - Probleem: võimalused on välja arendamata ja üksteisega konfliktis
  - Võimaluste lisamise asemel peaks üldistama olemasolevaid!
  - Aritmeetilised operatsioonid (funktsioonid) on möödapääsmatud!
- Tüüpimise motivatsioon
  - Probleem: Harva(?) esinevad vead erijuhtudest.
  - Võimaldab vältida vigu ja anda edasi kompileerimisaegset infot.
- Puhta FP motivatsioon
  - Selleks et uurida keerulisi struktuure, peab olema võimalus neid keelata!
  - Näide: hajus ja paralleelne arvutus

Funktsionaalne Programmeerimine (FP)

- FP motivatsioon
  - Probleem: võimalused on välja arendamata ja üksteisega konfliktis
  - Võimaluste lisamise asemel peaks üldistama olemasolevaid!
  - Aritmeetilised operatsioonid (funktsioonid) on möödapääsmatud!
- Tüüpimise motivatsioon
  - Probleem: Harva(?) esinevad vead erijuhtudest.
  - Võimaldab vältida vigu ja anda edasi kompileerimisaegset infot.
- Puhta FP motivatsioon
  - Selleks et uurida keerulisi struktuure, peab olema võimalus neid keelata!
  - Näide: hajus ja paralleelne arvutus
- Laisa FP motivatsioon
  - Teoreetiliselt paindlikum kui agar väärtustamine (FP magistriaine).
  - Näide: lõpmatute listidega arvutamine

Funktsionaalne Programmeerimine (FP)

- FP motivatsioon
  - Probleem: võimalused on välja arendamata ja üksteisega konfliktis
  - Võimaluste lisamise asemel peaks üldistama olemasolevaid!
  - Aritmeetilised operatsioonid (funktsioonid) on möödapääsmatud!
- Tüüpimise motivatsioon
  - Probleem: Harva(?) esinevad vead erijuhtudest.
  - Võimaldab vältida vigu ja anda edasi kompileerimisaegset infot.
- Puhta FP motivatsioon
  - Selleks et uurida keerulisi struktuure, peab olema võimalus neid keelata!
  - Näide: hajus ja paralleelne arvutus
- Laisa FP motivatsioon
  - Teoreetiliselt paindlikum kui agar väärtustamine (FP magistriaine).
  - Näide: lõpmatute listidega arvutamine

Loe lisaks: RWH, eessõna

Loogiline Programmeerimine (LP)

- Deklaratiivne programmeerimine, nagu FP-gi.
  - Ei keskendus sellele kuidas arvutada vaid mis on tõene.

Loogiline Programmeerimine (LP)

- Deklaratiivne programmeerimine, nagu FP-gi.
  - Ei keskendus sellele kuidas arvutada vaid mis on tõene.
- Loodud 70ndatel tehisintelekti ja lingvistika tarbeks.

Loogiline Programmeerimine (LP)

- Deklaratiivne programmeerimine, nagu FP-gi.
  - Ei keskendus sellele kuidas arvutada vaid mis on tõene.
- Loodud 70ndatel tehisintelekti ja lingvistika tarbeks.
- Baasoperatsiooniks relatsioonid, mitte funktsioonid.
  - Ühele argumendile mitu tagastusväärtust.
  - Saab arvutada tagurpidi: tagastusväärtuse järgi argumenti jne.

Loogiline Programmeerimine (LP)

- Deklaratiivne programmeerimine, nagu FP-gi.
  - Ei keskendus sellele kuidas arvutada vaid mis on tõene.
- Loodud 70ndatel tehisintelekti ja lingvistika tarbeks.
- Baasoperatsiooniks relatsioonid, mitte funktsioonid.
  - Ühele argumendile mitu tagastusväärtust.
  - Saab arvutada tagurpidi: tagastusväärtuse järgi argumenti jne.
- Ei sobi hästi algoritmide implementeerimiseks.
- Relatsioone saab käsitleda FPs kui hulka tagastava funktsiooni.

Imperatiivne Programmeerimine

- Kasvanud välja protsessori käskude üldistamisest.
- Defineerib, mis järjekorras täidetakse programmi seisundit teisendavaid lauseid.

Imperatiivne Programmeerimine

- Kasvanud välja protsessori käskude üldistamisest.
- Defineerib, mis järjekorras täidetakse programmi seisundit teisendavaid lauseid.
- Seda olete juba õppinud: Python, Java jne.

Imperatiivne Programmeerimine

- Kasvanud välja protsessori käskude üldistamisest.
- Defineerib, mis järjekorras täidetakse programmi seisundit teisendavaid lauseid.
- Seda olete juba õppinud: Python, Java jne.

- Kriitika: palju vabadust, projekti kasvades ei saa koodist aru.
  - paralleelarvutuse lisamine keerukam
  - lihtne õpetada kuid see annab halba eeskuju

Imperatiivne Programmeerimine

- Kasvanud välja protsessori käskude üldistamisest.
- Defineerib, mis järjekorras täidetakse programmi seisundit teisendavaid lauseid.
- Seda olete juba õppinud: Python, Java jne.

- Kriitika: palju vabadust, projekti kasvades ei saa koodist aru.
  - paralleelarvutuse lisamine keerukam
  - lihtne õpetada kuid see annab halba eeskuju

$\Longrightarrow$ õpime juurde FP-d ja Haskelli

Haskell

- Haskell erineb Pythonist ja Javast väga palju. Varasemad oskused Pythonist või Javast ei ole otse rakendtavad!

Haskell

- Haskell erineb Pythonist ja Javast väga palju. Varasemad oskused Pythonist või Javast ei ole otse rakendtavad!

- Seetõttu on Haskelli targem õppida kui matemaatikat/algebrat, mitte kui programmeerimist.

Haskell

- Haskell erineb Pythonist ja Javast väga palju. Varasemad oskused Pythonist või Javast ei ole otse rakendtavad!
- Seetõttu on Haskelli targem õppida kui matemaatikat/algebrat, mitte kui programmeerimist.
- Väga tähtis on, et te töötaksite juba algusest peale kaasa. Alustame väga lihtsate programmidega ja jõuame alles kursuse lõpuks mingile arvestatavale tasemele.

Haskell

- Haskelli programm koosneb definitsioonidest
  - Näiteks, loome faili test.hs:

```
a = 40.0
b = 30.0
c = sqrt (a^2 + b^2)
```

Haskell

- Haskelli programm koosneb definitsioonidest
  - Näiteks, loome faili test.hs:

```
a = 40.0
b = 30.0
c = sqrt (a^2 + b^2)
```

- Definitsioone saab lugeda interaktiivsesse keskkonda:

```
> stack ghci test.hs
```

Haskell

- Haskelli programm koosneb definitsioonidest
  - Näiteks, loome faili test.hs:

```
a = 40.0
b = 30.0
c = sqrt (a^2 + b^2)
```

- Definitsioone saab lugeda interaktiivsesse keskkonda:

```
> stack ghci test.hs
```

- ... ja siis käivitada

```
*Main> c
50.0
```

Haskell

- Haskelli programm koosneb definitsioonidest
  - Näiteks, loome faili test.hs:

```
a = 40.0
b = 30.0
c = sqrt (a^2 + b^2)
```

- Definitsioone saab lugeda interaktiivsesse keskkonda:

```
> stack ghci test.hs
```

- ... ja siis käivitada

```
*Main> c
50.0
```

- Mida see programm arvutab?

Haskell

- Haskelli programm koosneb definitsioonidest
  - Näiteks, loome faili test.hs:

```
a = 40.0
b = 30.0
c = sqrt (a^2 + b^2)
```

- Definitsioone saab lugeda interaktiivsesse keskkonda:

```
> stack ghci test.hs
```

- ... ja siis käivitada

```
*Main> c
50.0
```

- Mida see programm arvutab?
- Mis juhtub, kui muuta definitsioonide järjekorda?

Haskell

- Haskelli programm koosneb definitsioonidest
  - Näiteks, loome faili test.hs:

```
a = 40.0
b = 30.0
c = sqrt (a^2 + b^2)
```

- Definitsioone saab lugeda interaktiivsesse keskkonda:

```
> stack ghci test.hs
```

- ... ja siis käivitada

```
*Main> c
50.0
```

- Mida see programm arvutab?
- Mis juhtub, kui muuta definitsioonide järjekorda?

Loe lisaks: RWH, peatükk 1; LYaH, peatükk 2, Starting Out

Tüübid

- Igal defineeritaval nimel on tüüp. Enamasti ei pea tüüpe juurde kirjutama, kuid dokumenteerimise eesmärgil on seda siiski soovitatav aegajalt teha.

Tüübid

- Igal defineeritaval nimel on tüüp. Enamasti ei pea tüüpe juurde kirjutama, kuid dokumenteerimise eesmärgil on seda siiski soovitatav aegajalt teha.

- Näiteks:

```
a, b, c :: Float
a = 40
b = 30
c = sqrt (a^2 + b^2)
```

  või

```
a :: Float
a = 40
b :: Float
b = 30
c :: Float
c = sqrt (a^2 + b^2)
```

Tüübid

- Igal defineeritaval nimel on tüüp. Enamasti ei pea tüüpe juurde kirjutama, kuid dokumenteerimise eesmärgil on seda siiski soovitatav aegajalt teha.
- Näiteks:

```
a, b, c :: Float
a = 40
b = 30
c = sqrt (a^2 + b^2)
```

või

```
a :: Float
a = 40
b :: Float
b = 30
c :: Float
c = sqrt (a^2 + b^2)
```

Loe lisaks: LYaH, peatükk 3, Types and Typeclasses, Believe the type

Funktsioonide defineerimine

- Funktsioone defineeritakse samuti võrdusmärgiga. Võetakse abiks formaalsed parameetrid.
  - näide:

```
pyth :: Float -> Float -> Float
pyth x y = sqrt (x^2 + y^2)
```

Funktsioonide defineerimine

- Funktsioone defineeritakse samuti võrdusmärgiga. Võetakse abiks formaalsed parameetrid.
  - näide:

```
pyth :: Float -> Float -> Float
pyth x y = sqrt (x^2 + y^2)
```

- Haskell järgib matemaatilist notatsiooni, kuid on erandeid:
  - "$f(x, y)$" asemel kirjutame "$f\ x\ y$"

Funktsioonide defineerimine

- Funktsioone defineeritakse samuti võrdusmärgiga. Võetakse abiks formaalsed parameetrid.
  - näide:

```
pyth :: Float -> Float -> Float
pyth x y = sqrt (x^2 + y^2)
```

- Haskell järgib matemaatilist notatsiooni, kuid on erandeid:
  - "$f(x, y)$" asemel kirjutame "$f\ x\ y$"
- Funktsiooni tüüp on "α -> β", kus α on sisendi tüüp ja β tulemuse tüüp

Faktoriaali näide

- If-avaldisel põhinev faktoriaal

```
fact1 :: Int -> Int
fact1 n = if n==0 then 1 else n * fact1 (n-1)
```

Faktoriaali näide

- If-avaldisel põhinev faktoriaal

```
fact1 :: Int -> Int
fact1 n = if n==0 then 1 else n * fact1 (n-1)
```

- `fact1` väärtustamine

```
fact1 2
==> if 2 == 0 then 1 else 2 * fact1 (2-1)
==> if 2 == 0 then 1 else 2 * fact1 1
==> 2 * fact1 1
==> 2 * (if 1 == 0 then 1 else 1 * fact1 (1-1))
==> 2 * (if 1 == 0 then 1 else 1 * fact1 0)
==> 2 * (1 * fact1 0)
==> 2 * (1 * (if 0 == 0 then 1 else 0 * fact1 (-1)))
==> 2 * (1 * 1)
==> 2
```

- 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)
```

- Akumulaatorit kasutav, where-konstruktsiooniga

```
fact5 :: Int -> Int
fact5 n = fact5' 1 n
       where fact5' a 0 = a
             fact5' a m = fact5' (a*m) (m-1)
```

- Akumulaatorit kasutav, where-konstruktsiooniga

```
fact5 :: Int -> Int
fact5 n = fact5' 1 n
       where fact5' a 0 = a
             fact5' a m = fact5' (a*m) (m-1)
```

- Akumulaatorit kasutav, let-konstruktsiooniga

```
fact6 :: Integer -> Integer
fact6 n =
  let fact6' = \ a n -> case n of
                          0 -> a
                          _ -> fact6' (a*n) (n-1)
  in fact6' 1 n
```

- Akumulaatorit kasutav, where-konstruktsiooniga

```
fact5 :: Int -> Int
fact5 n = fact5' 1 n
       where fact5' a 0 = a
             fact5' a m = fact5' (a*m) (m-1)
```

- Akumulaatorit kasutav, let-konstruktsiooniga

```
fact6 :: Integer -> Integer
fact6 n =
  let fact6' = \ a n -> case n of
                          0 -> a
                          _ -> fact6' (a*n) (n-1)
  in fact6' 1 n
```

- product funktsiooni ja jada kasutav faktoriaal

```
fact7 :: Int -> Int
fact7 n = product [1..n]
```

- Akumulaatorit kasutav, where-konstruktsiooniga

```
fact5 :: Int -> Int
fact5 n = fact5' 1 n
       where fact5' a 0 = a
             fact5' a m = fact5' (a*m) (m-1)
```

- Akumulaatorit kasutav, let-konstruktsiooniga

```
fact6 :: Integer -> Integer
fact6 n =
  let fact6' = \ a n -> case n of
                          0 -> a
                          _ -> fact6' (a*n) (n-1)
  in fact6' 1 n
```

- product funktsiooni ja jada kasutav faktoriaal

```
fact7 :: Int -> Int
fact7 n = product [1..n]
```

Loe lisaks: LYaH, peatükk 4

Vindi ülekeeramine . . .

Vindi ülekeeramine ...

- Püsipunktikombinaatori abil defineeritud faktoriaal

```
fact9 :: Integer -> Integer
fact9 = fixedPt f
  where f g 0     = 1
        f g n     = n * g (n-1)
        fixedPt f = g where g = f g
```

- “The Evolution of a Haskell Programmer”
  - https://www.willamette.edu/~fruehr/haskell/evolution.html

Programmeerimise paradigmad

Saab jagada kaheks:

- imperatiivsed e. mis operatsioone teha
  - Lisaks jaotatakse: protseduuraalsed ja objektorienteeritud

Programmeerimise paradigmad

Saab jagada kaheks:

- imperatiivsed e. mis operatsioone teha
  - Lisaks jaotatakse: protseduuraalsed ja objektorienteeritud
- deklaratiivsed e. lahenduse (tõe) kirjeldamine
  - Lisaks jaotatakse: funktsionaalne ja loogiline

Programmeerimise paradigmad

Saab jagada kaheks:

- imperatiivsed e. mis operatsioone teha
  - Lisaks jaotatakse: protseduuraalsed ja objektorienteeritud
- deklaratiivsed e. lahenduse (tõe) kirjeldamine
  - Lisaks jaotatakse: funktsionaalne ja loogiline

Erinevused:

- Imperatiivne kasvas välja protsessori käsustiku abstaheerimisest.
- Deklaratiivne kasvas välja matemaatikast.

Programmeerimise paradigmad

Saab jagada kaheks:

- imperatiivsed e. mis operatsioone teha
  - Lisaks jaotatakse: protseduuraalsed ja objektorienteeritud
- deklaratiivsed e. lahenduse (tõe) kirjeldamine
  - Lisaks jaotatakse: funktsionaalne ja loogiline

Erinevused:

- Imperatiivne kasvas välja protsessori käsustiku abstaheerimisest.
- Deklaratiivne kasvas välja matemaatikast.

Matemaatik/loogik tahab mõelda

- algebralisteststruktuuridest nagu rühmad, monoidid, ring jne.
- funktsioonidest, hulkadest ja relatsioonidest

Programmeerimise paradigmad

Saab jagada kaheks:

- imperatiivsed e. mis operatsioone teha
  - Lisaks jaotatakse: protseduuraalsed ja objektorienteeritud
- deklaratiivsed e. lahenduse (tõe) kirjeldamine
  - Lisaks jaotatakse: funktsionaalne ja loogiline

Erinevused:

- Imperatiivne kasvas välja protsessori käsustiku abstaheerimisest.
- Deklaratiivne kasvas välja matemaatikast.

Matemaatik/loogik tahab mõelda

- algebralisteststruktuuridest nagu rühmad, monoidid, ring jne.
- funktsioonidest, hulkadest ja relatsioonidest

Paljud protsessori instruktsioonid pole lihtsalt modelleeritavad – keerulised erijuhud.

Keerulised erijuhud – näide

Olgu meil Java programm, milles on selline koodirida:

```
int x = Math.abs(y);
```

Kas `x` on positiivne (, null) või negatiivne?

Keerulised erijuhud – näide

Olgu meil Java programm, milles on selline koodirida:

```
int x = Math.abs(y);
```

Kas `x` on positiivne (, null) või negatiivne?

```
Math.abs(Integer.MIN_VALUE) == Integer.MIN_VALUE
```

Funktsionaalsed progekeeled

- Kombinaatorloogika (M. Schönfinkel 1924, H. Curry 1927)
- Lambda-arvutus (A. Church 1936)
- Lisp (J. McCarthy 1958)
- ML ja polümorfne tüübisüsteem (R. Milner 1978)
- Hope, Sasl, Miranda, . . . (1980 – 85)
- Haskell (1988)

Kõrvalepõige: Verifitseeritavad programmid

- Tüübiteooria (B. Russell, 1910-1913)
- Kombinaatorloogika (M. Schönfinkel 1924, H. Curry 1927)
- Lambda-arvutus (A. Church 1936)
- Lihtsalt tüübitud lambda-arvutus (A. Church 1940)
- Intuitionistlik tüübiteooria (Per Martin-Löf, 1972)
- ML ja polümorfne tüübisüsteem (R. Milner 1978)
- NuPRL (1984)
- Coq (1989)

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*.

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.

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.
- $\Longrightarrow$ kõrgemat järku funktsioonid

```
map (max 3) [2,3,4]
```

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.
- $\Longrightarrow$ kõrgemat järku funktsioonid

  `map (max 3) [2,3,4]` $\rightsquigarrow$ `[max 3 2, max 3 3, max 3 4]`

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.
- $\Longrightarrow$ 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?

Tugevalt ja staatiliselt tüübitud

- Mida rohkem programmeerimiskeel lubab seda parem?
  - PostScript (1982) vs. Portable Document Format (1993)

Tugevalt ja staatiliselt tüübitud

- Mida rohkem programmeerimiskeel lubab seda parem?
  - PostScript (1982) vs. Portable Document Format (1993)
- Piiramise üks võimalus on *tüübisüsteemiga*.

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 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.

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.

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.

Puhas keel

- Puhas — kõrvaltoimevaba ehk funktsiooni kutse tulemus sõltub ainult parameetrite väärtustest.
- $\Longrightarrow$ iga funktsiooni saab testida teistest eraldi

Puhas keel

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

Puhas keel

- Puhas — kõrvaltoimevaba ehk funktsiooni kutse tulemus sõltub ainult parameetrite väärtustest.
- $\Longrightarrow$ iga funktsiooni saab testida teistest eraldi
- mittepuhtaid funktsioone (nagu juhuarvude genereerimine) pole võimalik funktsioonidena defineerida
- $\Longrightarrow$ 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`

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`

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`

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`

Laisk keel

Olgu meil selline kood:

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

- *agaras keeles* arvutatakse parameeter enne funktsiooni kutset

  `tagastaViis (1+1)` ⇝ `tagastaViis 2` ⇝ `5`

  `topelt (1+1)` ⇝ `topelt 2` ⇝ `2+2` ⇝ `4`

- *laisas keeles* tehakse funktsioonikutse asendus enne

  `tagastaViis (1+1)` ⇝ `5`

  `topelt (1+1)` ⇝ `(1+1)+(1+1)` ⇝ `2+(1+1)` ⇝ `2+2` ⇝ `4`

- Haskell teeb tegelikult hoopis nii

  `topelt (1+1)` ⇝ `x+x where x=1+1`

Laisk keel

Olgu meil selline kood:

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

- *agaras keeles* arvutatakse parameeter enne funktsiooni kutset

  `tagastaViis (1+1)` ⇝ `tagastaViis 2` ⇝ `5`

  `topelt (1+1)` ⇝ `topelt 2` ⇝ `2+2` ⇝ `4`

- *laisas keeles* tehakse funktsioonikutse asendus enne

  `tagastaViis (1+1)` ⇝ `5`

  `topelt (1+1)` ⇝ `(1+1)+(1+1)` ⇝ `2+(1+1)` ⇝ `2+2` ⇝ `4`

- Haskell teeb tegelikult hoopis nii

  `topelt (1+1)` ⇝ `x+x where x=1+1` ⇝ `x+x where x=2`

Laisk keel

Olgu meil selline kood:

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

- *agaras keeles* arvutatakse parameeter enne funktsiooni kutset

  `tagastaViis (1+1)` ⇝ `tagastaViis 2` ⇝ `5`

  `topelt (1+1)` ⇝ `topelt 2` ⇝ `2+2` ⇝ `4`

- *laisas keeles* tehakse funktsioonikutse asendus enne

  `tagastaViis (1+1)` ⇝ `5`

  `topelt (1+1)` ⇝ `(1+1)+(1+1)` ⇝ `2+(1+1)` ⇝ `2+2` ⇝ `4`

- Haskell teeb tegelikult hoopis nii

  `topelt (1+1)` ⇝ `x+x where x=1+1` ⇝ `x+x where x=2` == `2+2` ⇝ `4`

Infix Operaatorid I

- Infix operaatorid (näiteks +) ja -tüübid (näiteks ->) kirjutatakse argumentide vahele, mitte argumentide ette.

Infix Operaatorid I

- Infix operaatorid (näiteks +) ja -tüübid (näiteks ->) kirjutatakse argumentide vahele, mitte argumentide ette.
  - Näiteks: `5 + 2`, `2*pi*r^2`, `Float -> Int`

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 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
```

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)

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 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

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`

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 **
```

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:

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 (+)”

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

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 `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:

Ennikud ja *Unit* tüüp

Olemasolevatest tüüpidest saab luua paare ja muid ennikuid:

- `(1,'a'),((1.1,8,'x'),False)`

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)`

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
```

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 `()`.

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 `() :: ()`

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.

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`

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]]`

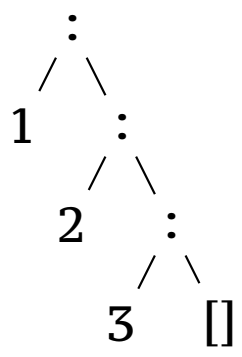
Järjendid e. listid

- 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]`

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:

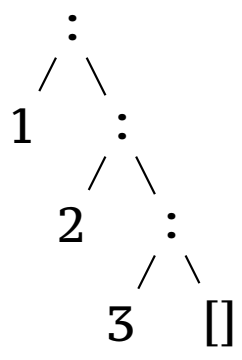
`[1, 2, 3]` ==



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:

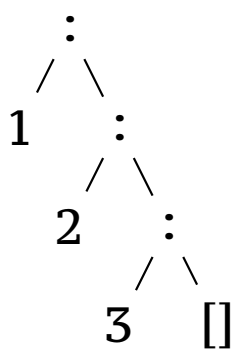
`[1, 2, 3] ==`



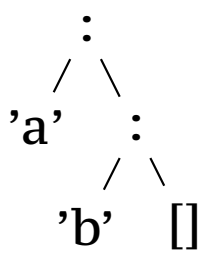
- Järjendi loomiseks on kaks konstruktorit, mis vastavad (list-tüüpi) puu tippudele.
  - `[] :: [a]`
  - `(:) :: a -> [a] -> [a]`
  - Näide: `[3,2,1] == 3 : 2 : 1 : []`

Listi näited

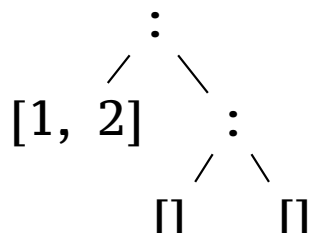
[1, 2, 3] =



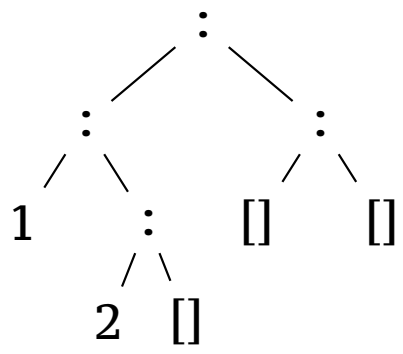
['a', 'b'] =



[[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
```

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!

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
```

Listide näited

- Korrutis

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

- Listi ümberpööramine

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

Listide näited

- Korrutis

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

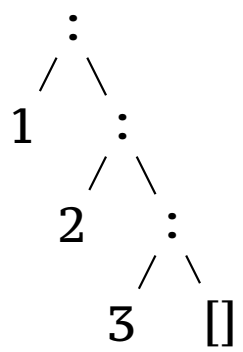
- Listi ümberpööramine

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

- Järjest asetsevate võrdsete elementide loendamine

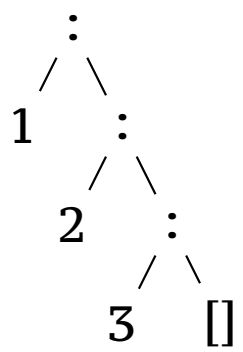
```
countEq (x:y:xs)
  | x==y       =  1 + countEq (y:xs)
  | otherwise  =  countEq (y:xs)
countEq _      =  0
```

Tähelepanekud



- Elemendi lisamine listi algusse väga kiire!
  - kiire == konstante aeg ja mälu
  - Põhjendus: argumente ei ole vaja kopeerida.

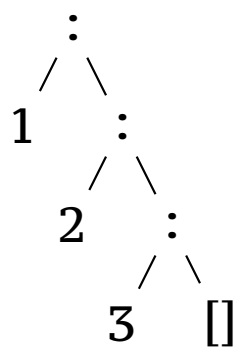
Tähelepanekud



- Elemendi lisamine listi algusse väga kiire!
  - kiire == konstante aeg ja mälu
  - Põhjendus: argumente ei ole vaja kopeerida.

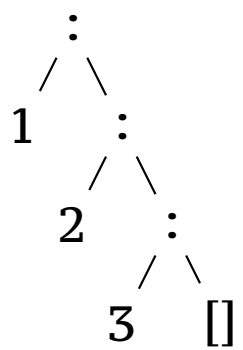
- Viimase elemendi selekteerimine aeglane!
  - aeglane == lineaarselt listi pikkusega

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!

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.

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.

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)
```

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`

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.

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.

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`

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`

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`

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`

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)`

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}

"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}

"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)

"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" muutuja

Näide

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

$$(\lambda\ \texttt{x}\ \texttt{y} \rightarrow \texttt{x+y})\ \texttt{y}\ =$$

Näide

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

$$(\lambda\ \mathtt{x}\ \mathtt{y} \rightarrow \mathtt{x+y})\ \mathtt{y} = (\lambda\ \mathtt{x} \rightarrow \lambda\ \mathtt{y} \rightarrow \mathtt{x+y})\ \mathtt{y}$$
$$\rightsquigarrow$$

Näide

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

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

Näide

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

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

Näide

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

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

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.

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:

$$\texttt{main} \rightsquigarrow \texttt{double (1+1)} \rightsquigarrow$$

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:

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

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:

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

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:

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

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:

$$\texttt{fact}\ 2 \rightsquigarrow$$

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:

$$\texttt{fact 2} \rightsquigarrow \texttt{2 * fact (2-1)} \rightsquigarrow$$

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:

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

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:

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

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:

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

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:

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

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:

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

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:

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

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

Kõrgemat järku funktsioon

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

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
```

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:

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

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] =
```

Näide

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

$$\texttt{map (+1) [1,2,3]} = \texttt{map (+1) (1:2:3:[])}$$

$$\rightsquigarrow$$

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:[])
                 ⇝
```

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:[])
                 ⇝
```

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:[])
                 ⇝
```

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:[])
                 ⇝
```

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) []
                 ⇝
```

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) []
                 ⇝
```

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 : []
                 =
```

Näide

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

$$
\begin{aligned}
\texttt{map (+1) [1,2,3]} &= \texttt{map (+1) (1:2:3:[])} \\
&\rightsquigarrow \texttt{(1+1) : map (+1) (2:3:[])} \\
&\rightsquigarrow \texttt{2 : map (+1) (2:3:[])} \\
&\rightsquigarrow \texttt{2 : (2+1) : map (+1) (3:[])} \\
&\rightsquigarrow \texttt{2 : 3 : map (+1) (3:[])} \\
&\rightsquigarrow \texttt{2 : 3 : (3+1) : map (+1) []} \\
&\rightsquigarrow \texttt{2 : 3 : 4 : map (+1) []} \\
&\rightsquigarrow \texttt{2 : 3 : 4 : []} \\
&= \texttt{[2, 3, 4]}
\end{aligned}
$$

Kõrgemat järku funktsioon II

Näiteks

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

Funktsiooni `foldr` illustreerib järgmine võrdus:

$$\texttt{foldr}\ (+)\ b\ [x_1, x_2, \ldots, x_n] = x_1 + (x_2 + (\ldots + (x_n + b)))$$

Kõrgemat järku funktsioon II

Näiteks

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

Funktsiooni `foldr` illustreerib järgmine võrdus:

$$\texttt{foldr}\ (+)\ b\ [x_1, x_2, \ldots, x_n] = x_1 + (x_2 + (\ldots + (x_n + b)))$$

Funktsiooni `map` saab defineerida läbi `foldr`-i

```
map f xs = foldr g [] xs
   where g x y = (f x) : y
```

ehk

$$\texttt{foldr}\ g\ []\ [x_1, x_2, \ldots, x_n] = x_1 `g` (x_2 `g` (\ldots `g` (x_n `g` [])))) = f\ x_1 : f\ x_2 : \ldots : f\ x_n : []$$

Näide

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

```
foldr (+) 0 (1:2:3:[]) ⇝
```

Näide

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

```
foldr (+) 0 (1:2:3:[]) ⇝ (+) 1 (foldr (+) 0 (2:3:[]))
                       ⇝
```

Näide

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

```
foldr (+) 0 (1:2:3:[]) ⇝ (+) 1 (foldr (+) 0 (2:3:[]))
                       ⇝ (+) 1 ((+) 2 (foldr (+) 0 (3:[])))
                       ⇝
```

Näide

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

```
foldr (+) 0 (1:2:3:[]) ⇝ (+) 1 (foldr (+) 0 (2:3:[]))
                       ⇝ (+) 1 ((+) 2 (foldr (+) 0 (3:[])))
                       ⇝ (+) 1 ((+) 2 ((+) 3 (foldr (+) 0 [])))
                       ⇝
```

Näide

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

```
foldr (+) 0 (1:2:3:[]) ⇝ (+) 1 (foldr (+) 0 (2:3:[]))
                       ⇝ (+) 1 ((+) 2 (foldr (+) 0 (3:[])))
                       ⇝ (+) 1 ((+) 2 ((+) 3 (foldr (+) 0 [])))
                       ⇝ (+) 1 ((+) 2 ((+) 3 0))
                       ⇝
```

Näide

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

```
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)
                       ⇝
```

Näide

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

```
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)
                       ⇝ (+) 1 5
                       ⇝
```

Näide

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

```
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)
                       ⇝ (+) 1 5
                       ⇝ 6
```

Näide

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

head xs = foldr (\ a b -> a) undefined xs
```

head (3:4:5:[]) ⇝

Näide

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

head xs = foldr (\ a b -> a) undefined xs
```

```
head (3:4:5:[]) ⇝ foldr (\a b -> a) undefined (3:4:5:[])
                ⇝
```

Näide

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

head xs = foldr (\ a b -> a) undefined xs
```

```
head (3:4:5:[]) ⇝ foldr (\a b -> a) undefined (3:4:5:[])
                ⇝ (\a b -> a) 3 (foldr (\a b -> a) undefined (4:5:[]))
                ⇝
```

Näide

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

head xs = foldr (\ a b -> a) undefined xs
```

```
head (3:4:5:[]) ⇝ foldr (\a b -> a) undefined (3:4:5:[])
                ⇝ (\a b -> a) 3 (foldr (\a b -> a) undefined (4:5:[]))
                ⇝ (\b -> 3) (foldr (\a b -> a) undefined (4:5:[]))
                ⇝
```

Näide

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

head xs = foldr (\ a b -> a) undefined xs
```

```
head (3:4:5:[]) ⇝ foldr (\a b -> a) undefined (3:4:5:[])
                ⇝ (\a b -> a) 3 (foldr (\a b -> a) undefined (4:5:[]))
                ⇝ (\b -> 3) (foldr (\a b -> a) undefined (4:5:[]))
                ⇝ 3
```

Kõrgemat järku funktsioon III

Näiteks

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

Funktsiooni `foldl` illustreerib järgmine võrdus:

$$\texttt{foldl}\ (+)\ b\ [x_1, x_2, \ldots, x_n] = (((b + x_1) + x_2) + \ldots) + x_n$$

Kõrgemat järku funktsioon III

Näiteks

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

Funktsiooni `foldl` illustreerib järgmine võrdus:

$$\texttt{foldl}\ (+)\ b\ [x_1, x_2, \ldots, x_n] = (((b + x_1) + x_2) + \ldots) + x_n$$

Funktsiooni `reverse` saab defineerida läbi `foldl`-i

```
reverse xs = foldl g [] xs
  where g x y = y : x
```

ehk

$$\texttt{foldl}\ g\ []\ [x_1, x_2, \ldots, x_n] = ((([]\ `g`\ x_1)\ `g`\ x_2)\ `g`\ \ldots)\ `g`\ x_n = x_n : \ldots : x_2 : x_1 : []$$

Näide

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

reverse xs = foldl g [] xs
  where g x y = y : x
```

```
reverse (1:2:3:[]) =
```

Näide

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

reverse xs = foldl g [] xs
  where g x y = y : x
```

$$\texttt{reverse (1:2:3:[])} = \texttt{foldl g [] (1:2:3:[])}$$

$$\rightsquigarrow$$

Näide

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

reverse xs = foldl g [] xs
  where g x y = y : x
```

$$
\begin{aligned}
\texttt{reverse (1:2:3:[])} &= \texttt{foldl g [] (1:2:3:[])} \\
&\rightsquigarrow \texttt{foldl g (g [] 1) (2:3:[])} \\
&\rightsquigarrow
\end{aligned}
$$

Näide

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

reverse xs = foldl g [] xs
  where g x y = y : x
```

$$
\begin{aligned}
\texttt{reverse}\ (1{:}2{:}3{:}[]) &= \texttt{foldl g [] (1:2:3:[])} \\
&\rightsquigarrow \texttt{foldl g (g [] 1) (2:3:[])} \\
&\rightsquigarrow \texttt{foldl g (g (g [] 1) 2) (3:[])} \\
&\rightsquigarrow
\end{aligned}
$$

Näide

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

reverse xs = foldl g [] xs
  where g x y = y : x
```

$$
\begin{aligned}
\texttt{reverse}\ (1{:}2{:}3{:}[]) &= \texttt{foldl g [] (1:2:3:[])} \\
&\rightsquigarrow \texttt{foldl g (g [] 1) (2:3:[])} \\
&\rightsquigarrow \texttt{foldl g (g (g [] 1) 2) (3:[])} \\
&\rightsquigarrow \texttt{foldl g (g (g (g [] 1) 2) 3) []} \\
&\rightsquigarrow
\end{aligned}
$$

Näide

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

reverse xs = foldl g [] xs
  where g x y = y : x
```

$$
\begin{aligned}
\texttt{reverse}\ (1{:}2{:}3{:}[]) &= \texttt{foldl g [] } (1{:}2{:}3{:}[]) \\
&\rightsquigarrow \texttt{foldl g (g [] 1) } (2{:}3{:}[]) \\
&\rightsquigarrow \texttt{foldl g (g (g [] 1) 2) } (3{:}[]) \\
&\rightsquigarrow \texttt{foldl g (g (g (g [] 1) 2) 3) []} \\
&\rightsquigarrow \texttt{g (g (g [] 1) 2) 3} \\
&\rightsquigarrow
\end{aligned}
$$

Näide

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

reverse xs = foldl g [] xs
  where g x y = y : x
```

```
reverse (1:2:3:[]) = foldl g [] (1:2:3:[])
                   ⇝ foldl g (g [] 1) (2:3:[])
                   ⇝ foldl g (g (g [] 1) 2) (3:[])
                   ⇝ foldl g (g (g (g [] 1) 2) 3) []
                   ⇝ g (g (g [] 1) 2) 3
                   ⇝ 3 : (g (g [] 1) 2)
                   ⇝
```

Näide

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

reverse xs = foldl g [] xs
  where g x y = y : x
```

```
reverse (1:2:3:[]) = foldl g [] (1:2:3:[])
                   ⇝ foldl g (g [] 1) (2:3:[])
                   ⇝ foldl g (g (g [] 1) 2) (3:[])
                   ⇝ foldl g (g (g (g [] 1) 2) 3) []
                   ⇝ g (g (g [] 1) 2) 3
                   ⇝ 3 : (g (g [] 1) 2)
                   ⇝ 3 : 2 : (g [] 1)
                   ⇝
```

Näide

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

reverse xs = foldl g [] xs
  where g x y = y : x
```

$$
\begin{aligned}
\texttt{reverse}\ (1{:}2{:}3{:}[]) &= \texttt{foldl g [] (1:2:3:[])} \\
&\rightsquigarrow \texttt{foldl g (g [] 1) (2:3:[])} \\
&\rightsquigarrow \texttt{foldl g (g (g [] 1) 2) (3:[])} \\
&\rightsquigarrow \texttt{foldl g (g (g (g [] 1) 2) 3) []} \\
&\rightsquigarrow \texttt{g (g (g [] 1) 2) 3} \\
&\rightsquigarrow \texttt{3 : (g (g [] 1) 2)} \\
&\rightsquigarrow \texttt{3 : 2 : (g [] 1)} \\
&\rightsquigarrow \texttt{3 : 2 : 1 : []}
\end{aligned}
$$

Näide

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

last xs = foldl g u
  where g a b = b
        u = undefined
```

```
last (3:4:5:[]) ⇝
```

Näide

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

last xs = foldl g u
  where g a b = b
        u = undefined
```

```
last (3:4:5:[]) ⇝ foldl g [] (3:4:5:[])
                ⇝
```

Näide

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

last xs = foldl g u
  where g a b = b
        u = undefined
```

```
last (3:4:5:[]) ⇝ foldl g [] (3:4:5:[])
                ⇝ foldl g (g [] 3) (4:5:[])
                ⇝
```

Näide

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

last xs = foldl g u
  where g a b = b
        u = undefined
```

```
last (3:4:5:[]) ⇝ foldl g [] (3:4:5:[])
                ⇝ foldl g (g [] 3) (4:5:[])
                ⇝ foldl g (g (g [] 3) 4) (5:[])
                ⇝
```

Näide

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

last xs = foldl g u
  where g a b = b
        u = undefined
```

```
last (3:4:5:[]) ⇝ foldl g [] (3:4:5:[])
                ⇝ foldl g (g [] 3) (4:5:[])
                ⇝ foldl g (g (g [] 3) 4) (5:[])
                ⇝ foldl g (g (g (g [] 3) 4) 5) []
                ⇝
```

Näide

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

last xs = foldl g u
  where g a b = b
        u = undefined
```

```
last (3:4:5:[]) ⇝ foldl g [] (3:4:5:[])
                ⇝ foldl g (g [] 3) (4:5:[])
                ⇝ foldl g (g (g [] 3) 4) (5:[])
                ⇝ foldl g (g (g (g [] 3) 4) 5) []
                ⇝ g (g (g [] 3) 4) 5
                ⇝
```

Näide

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

last xs = foldl g u
  where g a b = b
        u = undefined
```

```
last (3:4:5:[]) ⇝ foldl g [] (3:4:5:[])
                ⇝ foldl g (g [] 3) (4:5:[])
                ⇝ foldl g (g (g [] 3) 4) (5:[])
                ⇝ foldl g (g (g (g [] 3) 4) 5) []
                ⇝ g (g (g [] 3) 4) 5
                ⇝ 5
```

Kõrgemat järku funktsioonid kui disainimustrid

- `foldr` on listi itereerimise muster.

```
f :: [Double] -> Double
f (x:xs) = x + f xs
f []     = 0
```

vs.

```
f = foldr (+) 0
```

- Lihtsate funktsioonide puhul on ka rekursiivne lahendus selge.
- Pikema ülesande puhul on hea eraldada listi töötlemine.

Miks eelistada kõrgemat järku funktsioone

- Programmeerijatena ei jõua me kaugele mõeldes *bittide*-tasemel.

Miks eelistada kõrgemat järku funktsioone

- Programmeerijatena ei jõua me kaugele mõeldes *bittide*-tasemel.
- Peame jõudma kõrgemale tasemele — abstraktsemalt

Miks eelistada kõrgemat järku funktsioone

- Programmeerijatena ei jõua me kaugele mõeldes *bittide*-tasemel.
- Peame jõudma kõrgemale tasemele — abstraktsemalt

```
f :: [Double] -> Double
f (0:_)  = 0
f (x:xs) = x + f xs
f []     = 0
```

vs.

```
f = sum . takeWhile (/=0)
```

Loe lisaks: RWH, peatükk 4, lk 84..99

Curry stiil ja fun. osaline rakendamine

- Selle asemel, et kirjutada

```
uusFunktsioon x y z = olemasolevFunktsioon (x+1) y z
```

tuleks Haskellis kirjuldada

```
uusFunktsioon x = olemasolevFunktsioon (x+1)
```

Curry stiil ja fun. osaline rakendamine

- Selle asemel, et kirjutada

```
uusFunktsioon x y z = olemasolevFunktsioon (x+1) y z
```

tuleks Haskellis kirjuldada

```
uusFunktsioon x = olemasolevFunktsioon (x+1)
```

  - Pane Tähele: funktsioon `olemasolevFunktsioon` on osaliselt rakendatud. Mõlemad funktsioonid võtavad (vähemalt!) kolm argumenti.

Curry stiil ja fun. osaline rakendamine

- Selle asemel, et kirjutada

```
uusFunktsioon x y z = olemasolevFunktsioon (x+1) y z
```

tuleks Haskellis kirjuldada

```
uusFunktsioon x = olemasolevFunktsioon (x+1)
```

  - Pane Tähele: funktsioon `olemasolevFunktsioon` on osaliselt rakendatud. Mõlemad funktsioonid võtavad (vähemalt!) kolm argumenti.

- Realistlikum näide:

```
f xs = sum (takeWhile (/=0) xs)
```

või siis

```
f = sum . takeWhile (/=0)
```

Curry stiil ja fun. osaline rakendamine

- Selle asemel, et kirjutada

```
uusFunktsioon x y z = olemasolevFunktsioon (x+1) y z
```

  tuleks Haskellis kirjuldada

```
uusFunktsioon x = olemasolevFunktsioon (x+1)
```

  - Pane Tähele: funktsioon `olemasolevFunktsioon` on osaliselt rakendatud. Mõlemad funktsioonid võtavad (vähemalt!) kolm argumenti.

- Realistlikum näide:

```
f xs = sum (takeWhile (/=0) xs)
```

  või siis

```
f = sum . takeWhile (/=0)
```

- Soovitus: Kirjuta funktsioone nii, et neid oleks võimalik ka osaliselt rakendada.
  - Funktsiooni `(a,b) -> c` karritud kuju on `a -> b -> c`.

Curry stiil ja fun. osaline rakendamine

- Selle asemel, et kirjutada

```
uusFunktsioon x y z = olemasolevFunktsioon (x+1) y z
```

tuleks Haskellis kirjuldada

```
uusFunktsioon x = olemasolevFunktsioon (x+1)
```

  - Pane Tähele: funktsioon `olemasolevFunktsioon` on osaliselt rakendatud. Mõlemad funktsioonid võtavad (vähemalt!) kolm argumenti.

- Realistlikum näide:

```
f xs = sum (takeWhile (/=0) xs)
```

või siis

```
f = sum . takeWhile (/=0)
```

- Soovitus: Kirjuta funktsioone nii, et neid oleks võimalik ka osaliselt rakendada.
  - Funktsiooni `(a,b) -> c` karritud kuju on `a -> b -> c`.

- Saab viia ka liiga kaugele:

```
max3 :: Int -> Int -> Int -> Int
max3  =  (. max) . ((.) . max)
```

Haskelli tüübisüsteem

Haskelli on *staatiliselt tüübitud*, *tugevalt tüübitud*, *tüübituletusega* programmeerimiskeel.

Haskelli tüübisüsteem

Haskelli on *staatiliselt tüübitud*, *tugevalt tüübitud*, *tüübituletusega* programmeerimiskeel.

- staatiliselt tüübitud — tüübid teada kompileerimise ajal

Haskelli tüübisüsteem

Haskelli on *staatiliselt tüübitud*, *tugevalt tüübitud*, *tüübituletusega* programmeerimiskeel.

- staatiliselt tüübitud — tüübid teada kompileerimise ajal
- tugevalt tüübitud
  - garantii, et väärtused on just seda tüüpi, millega nad end esitavad
  - tüüpe ei teisendata automaatselt. (nagu C-s int -> float)

S.t. rohkem eeltööd, et tüübivead eemaldada. Programm on aga siis töökindlam.

Haskelli tüübisüsteem

Haskelli on *staatiliselt tüübitud*, *tugevalt tüübitud*, *tüübituletusega* programmeerimiskeel.

- staatiliselt tüübitud — tüübid teada kompileerimise ajal
- tugevalt tüübitud
  - garantii, et väärtused on just seda tüüpi, millega nad end esitavad
  - tüüpe ei teisendata automaatselt. (nagu C-s int -> float)

S.t. rohkem eeltööd, et tüübivead eemaldada. Programm on aga siis töökindlam.

- tüübituletus — enamasti kirjutatakse tüüpe ainult selleks, et kontrollida, kas kompilaator saab programmist samamoodi aru nagu programmeerija.

Tüübid

- Baastüübid
  - `Int`, `Integer`, `Char`, `Double`, `Float`

Tüübid

- Baastüübid
  - `Int`, `Integer`, `Char`, `Double`, `Float`
- Funktsiooni tüüp
  - `Int -> Char`, `Float -> Float`, `Int -> (Char -> Int)`

Tüübid

- Baastüübid
  - `Int`, `Integer`, `Char`, `Double`, `Float`
- Funktsiooni tüüp
  - `Int -> Char`, `Float -> Float`, `Int -> (Char -> Int)`
- Tüübimuutujad – algavad väiketähega ja tähistavad ükskõik millist konkreetset tüüpi. (polümorfism, i.k. *polymorfism*)
  - `a -> Bool`, `a -> a`,

Tüübid

- Baastüübid
  - `Int`, `Integer`, `Char`, `Double`, `Float`
- Funktsiooni tüüp
  - `Int -> Char`, `Float -> Float`, `Int -> (Char -> Int)`
- Tüübimuutujad – algavad väiketähega ja tähistavad ükskõik millist konkreetset tüüpi. (polümorfism, i.k. *polymorfism*)
  - `a -> Bool`, `a -> a`,

parameetriline polümorfism: Haskellis ei saa polümorfne funktsioon teha kindlaks, mis konkreetne tüüp tüübimuutujal parajasti on. S.t. funktsioon on defineeritud olenemata konkreetset tüübist.

Tüübid

- Baastüübid
  - `Int`, `Integer`, `Char`, `Double`, `Float`
- Funktsiooni tüüp
  - `Int -> Char`, `Float -> Float`, `Int -> (Char -> Int)`
- Tüübimuutujad – algavad väiketähega ja tähistavad ükskõik millist konkreetset tüüpi. (polümorfism, i.k. *polymorfism*)
  - `a -> Bool`, `a -> a`,

parameetriline polümorfism: Haskellis ei saa polümorfne funktsioon teha kindlaks, mis konkreetne tüüp tüübimuutujal parajasti on. S.t. funktsioon on defineeritud olenemata konkreetset tüübist.

- Mis funktsioonid võivad olla tüüpidega `a -> a`, `a -> Bool` või `a -> b -> a`?

Loe lisaks: RWH, peatükk 2, lk 17..27

Mitu erinevat *puhast* väärtust?

- $|\text{Bool}| = 2$: True, False
- $|\text{Int}| \geq 2^{30}$: $-2^{29}, \ldots, 2^{29} - 1$
- $|\text{Integer}| = |\mathbb{N}|$: $0, 1, -1, \ldots$
- $|\text{Char}| = 2^{16}$: 'a','b','c',...

- $|\alpha \to \beta| = \beta^{\alpha}$

- Näiteks $|\text{Bool} \to \text{Bool}| = 2^2 = 4$
  1. $\{\text{True} \to \text{True}, \quad \text{False} \to \text{True}\}$
  2. $\{\text{True} \to \text{False}, \quad \text{False} \to \text{False}\}$
  3. $\{\text{True} \to \text{True}, \quad \text{False} \to \text{False}\}$
  4. $\{\text{True} \to \text{False}, \quad \text{False} \to \text{True}\}$

Mitu erinevat *puhast* väärtust?

- $|\text{Int}| \geq 2^{30}$: $-2^{29}, \ldots, 2^{29} - 1$
- $|\text{Integer}| = |\mathbb{N}|$: $0, 1, -1, \ldots$
- $|\text{Char}| = 2^{16}$: 'a','b','c',...

- $|\alpha \to \beta| = \beta^{\alpha}$

- Näiteks $|\text{Bool} \to \text{Bool}| = 2^2 = 4$
  1. $\{\text{True} \to \text{True}, \quad \text{False} \to \text{True}\}$
  2. $\{\text{True} \to \text{False}, \quad \text{False} \to \text{False}\}$
  3. $\{\text{True} \to \text{True}, \quad \text{False} \to \text{False}\}$
  4. $\{\text{True} \to \text{False}, \quad \text{False} \to \text{True}\}$

Mitu erinevat *puhast* väärtust?

- $|\text{Integer}| = |\mathbb{N}|$: $0, 1, -1, \ldots$
- $|\text{Char}| = 2^{16}$: 'a','b','c',...

- $|\alpha \to \beta| = \beta^{\alpha}$

- Näiteks $|\text{Bool} \to \text{Bool}| = 2^2 = 4$
  1. $\{\text{True} \to \text{True}, \quad \text{False} \to \text{True}\}$
  2. $\{\text{True} \to \text{False}, \quad \text{False} \to \text{False}\}$
  3. $\{\text{True} \to \text{True}, \quad \text{False} \to \text{False}\}$
  4. $\{\text{True} \to \text{False}, \quad \text{False} \to \text{True}\}$

Mitu erinevat *puhast* väärtust?

- $|\text{Char}| = 2^{16}$: 'a','b','c',...
- $|\alpha \to \beta| = \beta^{\alpha}$
- Näiteks $|\text{Bool} \to \text{Bool}| = 2^2 = 4$
  1. $\{\text{True} \to \text{True}, \quad \text{False} \to \text{True}\}$
  2. $\{\text{True} \to \text{False}, \quad \text{False} \to \text{False}\}$
  3. $\{\text{True} \to \text{True}, \quad \text{False} \to \text{False}\}$
  4. $\{\text{True} \to \text{False}, \quad \text{False} \to \text{True}\}$

Mitu erinevat *puhast* väärtust?

- $|\alpha \to \beta| = \beta^{\alpha}$
- Näiteks $|\text{Bool} \to \text{Bool}| = 2^2 = 4$
  1. $\{\text{True} \to \text{True}, \quad \text{False} \to \text{True}\}$
  2. $\{\text{True} \to \text{False}, \quad \text{False} \to \text{False}\}$
  3. $\{\text{True} \to \text{True}, \quad \text{False} \to \text{False}\}$
  4. $\{\text{True} \to \text{False}, \quad \text{False} \to \text{True}\}$

Mitu erinevat *puhast* väärtust?

- Näiteks $|\text{Bool} \to \text{Bool}| = 2^2 = 4$
  1. $\{\text{True} \to \text{True}, \quad \text{False} \to \text{True}\}$
  2. $\{\text{True} \to \text{False}, \quad \text{False} \to \text{False}\}$
  3. $\{\text{True} \to \text{True}, \quad \text{False} \to \text{False}\}$
  4. $\{\text{True} \to \text{False}, \quad \text{False} \to \text{True}\}$

Mitu erinevat Haskelli väärtust?

- iga väärtus saab olla undefined, mis lõpetab programmi töö
- ei saa kontrollida, kas mingi väärtus on undefined
- ... sellisel juhul on tagastusväärtus undefined

- Näiteks Bool $\to$ Bool:

1.	$\{\text{True} \to \text{True},$	$\text{False} \to \text{True},$	$\text{undefined} \to \text{undefined}\}$
2.	$\{\text{True} \to \text{False},$	$\text{False} \to \text{False},$	$\text{undefined} \to \text{undefined}\}$
3.	$\{\text{True} \to \text{True},$	$\text{False} \to \text{False},$	$\text{undefined} \to \text{undefined}\}$
4.	$\{\text{True} \to \text{False},$	$\text{False} \to \text{True},$	$\text{undefined} \to \text{undefined}\}$
5.	$\{\text{True} \to \text{undefined},$	$\text{False} \to \text{True},$	$\text{undefined} \to \text{undefined}\}$
6.	$\{\text{True} \to \text{undefined},$	$\text{False} \to \text{False},$	$\text{undefined} \to \text{undefined}\}$
7.	$\{\text{True} \to \text{True},$	$\text{False} \to \text{undefined},$	$\text{undefined} \to \text{undefined}\}$
8.	$\{\text{True} \to \text{False},$	$\text{False} \to \text{undefined},$	$\text{undefined} \to \text{undefined}\}$
9.	$\{\text{True} \to \text{undefined},$	$\text{False} \to \text{undefined},$	$\text{undefined} \to \text{undefined}\}$
10.	$\{\text{True} \to \text{True},$	$\text{False} \to \text{True},$	$\text{undefined} \to \text{True}\}$
11.	$\{\text{True} \to \text{False},$	$\text{False} \to \text{False},$	$\text{undefined} \to \text{False}\}$

- $|\alpha \to \beta|_H = (|\beta| + 1)^{|\alpha|} + |\beta| + \ldots$

Uute tüüpide loomine

Uusi tüüpe saab luua kolmel viisil:

data e. uue algebralise andmetüübi loomine,

type e. tüübi sünonüümi loomine ja

newtype e. olemasolevale tüübile uue ja eristatava nime tegemine.

Uute tüüpide loomine

Uusi tüüpe saab luua kolmel viisil:

- **data** e. uue algebralise andmetüübi loomine,
- **type** e. tüübi sünonüümi loomine ja
- **newtype** e. olemasolevale tüübile uue ja eristatava nime tegemine.

Näiteks sõned on Haskellis tähtede järejendi sünonüüm!

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

S.t. Tähtede listid on sõned ja sõned on tähtede listid!

```
['a','X','8'] :: String
"Tere" :: [Char]
```

Uute tüüpide loomine

Uusi tüüpe saab luua kolmel viisil:

- **data** e. uue algebralise andmetüübi loomine,
- **type** e. tüübi sünonüümi loomine ja
- **newtype** e. olemasolevale tüübile uue ja eristatava nime tegemine.

Näiteks sõned on Haskellis tähtede järejendi sünonüüm!

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

S.t. Tähtede listid on sõned ja sõned on tähtede listid!

```
['a','X','8'] :: String
"Tere" :: [Char]
```

Olemasolvast tüübist saab teha uue!

```
newtype Sõne = TeeSõne String
```

Nüüd saame kasutada konstruktorit `TeeSõne :: String -> Sõne`, s.t.

```
TeeSõne "Tere" :: Sõne
```

Algebralised andmetüübid

Tõeväärtuste tüüp `Bool` on defineeritud järgnevalt:

```
data Bool = True | False
```

Algebralised andmetüübid

Tõeväärtuste tüüp `Bool` on defineeritud järgnevalt:

```
data Bool = True | False
```

Sellest koodireast loeme välja järgnevat:

- defineeritakse uus andmetüüp `Bool`;

Algebralised andmetüübid

Tõeväärtuste tüüp `Bool` on defineeritud järgnevalt:

```
data Bool = True | False
```

Sellest koodireast loeme välja järgnevat:

- defineeritakse uus andmetüüp `Bool`;
- defineeritakse konstruktor `True :: Bool`;

Algebralised andmetüübid

Tõeväärtuste tüüp `Bool` on defineeritud järgnevalt:

```
data Bool = True | False
```

Sellest koodireast loeme välja järgnevat:

- defineeritakse uus andmetüüp `Bool`;
- defineeritakse konstruktor `True :: Bool`;
- defineeritakse konstruktor `False :: Bool`.

Algebralised andmetüübid

Tõeväärtuste tüüp `Bool` on defineeritud järgnevalt:

```
data Bool = True | False
```

Sellest koodireast loeme välja järgnevat:

- defineeritakse uus andmetüüp `Bool`;
- defineeritakse konstruktor `True :: Bool`;
- defineeritakse konstruktor `False :: Bool`.

Hiljem saab mustrisobitusega vaadata, millise konstruktoriga väärtus loodi:

```
f True  = ...
f False = ...
```

Näide

```
data Tõeväärtus = Tõene | Väär   deriving Show

test1 :: Tõeväärtus
test1 = Tõene
```

Näide

```
data Tõeväärtus = Tõene | Väär   deriving Show

test1 :: Tõeväärtus
test1 = Tõene

test2 :: Tõeväärtus
test2 = Väär
```

Näide

```
data Tõeväärtus = Tõene | Väär   deriving Show

test1 :: Tõeväärtus
test1 = Tõene

test2 :: Tõeväärtus
test2 = Väär

eitus :: Tõeväärtus -> Tõeväärtus
eitus Tõene = Väär
eitus Väär  = Tõene
```

Näide

```
data Tõeväärtus = Tõene | Väär   deriving Show

test1 :: Tõeväärtus
test1 = Tõene

test2 :: Tõeväärtus
test2 = Väär

eitus :: Tõeväärtus -> Tõeväärtus
eitus Tõene = Väär
eitus Väär  = Tõene

conj :: Tõeväärtus -> Tõeväärtus -> Tõeväärtus
conj Tõene Tõene = Tõene
conj _     _     = Väär
```

Näide

```
data Tõeväärtus = Tõene | Väär   deriving Show

test1 :: Tõeväärtus
test1 = Tõene

test2 :: Tõeväärtus
test2 = Väär

eitus :: Tõeväärtus -> Tõeväärtus
eitus Tõene = Väär
eitus Väär  = Tõene

conj :: Tõeväärtus -> Tõeväärtus -> Tõeväärtus
conj Tõene Tõene = Tõene
conj _     _     = Väär

disj :: Tõeväärtus -> Tõeväärtus -> Tõeväärtus
disj x y = eitus (eitus x `conj` eitus y)
```

Algebraliste andmetüüpide defineerimine

```
data UusTüüp a b = Konstr1 Int | Konstr2 a Char b | Konstr3
```

Algebraliste andmetüüpide defineerimine

```
data UusTüüp a b = Konstr1 Int | Konstr2 a Char b | Konstr3
```

Ehk siis:

- defineeritakse uued andmetüübid `UusTüüp a b`; näiteks
  - `UusTüüp Int Int`,
  - `UusTüüp Char (UusTüüp Char Int)`;

Algebraliste andmetüüpide defineerimine

```
data UusTüüp a b = Konstr1 Int | Konstr2 a Char b | Konstr3
```

Ehk siis:

- defineeritakse uued andmetüübid `UusTüüp a b`; näiteks
  - `UusTüüp Int Int`,
  - `UusTüüp Char (UusTüüp Char Int)`;
- defineeritakse konstruktor `Konstr1 :: Int -> UusTüüp a b`;

Algebraliste andmetüüpide defineerimine

```
data UusTüüp a b = Konstr1 Int | Konstr2 a Char b | Konstr3
```

Ehk siis:

- defineeritakse uued andmetüübid `UusTüüp a b`; näiteks
  - `UusTüüp Int Int`,
  - `UusTüüp Char (UusTüüp Char Int)`;
- defineeritakse konstruktor `Konstr1 :: Int -> UusTüüp a b`;
- defineeritakse konstruktor
  `Konstr2 :: a -> Char -> b -> UusTüüp a b`;

Algebraliste andmetüüpide defineerimine

```
data UusTüüp a b = Konstr1 Int | Konstr2 a Char b | Konstr3
```

Ehk siis:

- defineeritakse uued andmetüübid `UusTüüp a b`; näiteks
  - `UusTüüp Int Int`,
  - `UusTüüp Char (UusTüüp Char Int)`;
- defineeritakse konstruktor `Konstr1 :: Int -> UusTüüp a b`;
- defineeritakse konstruktor
  `Konstr2 :: a -> Char -> b -> UusTüüp a b`;
- defineeritakse konstruktor `Konstr3 :: UusTüüp a b`;

Algebraliste andmetüüpide defineerimine

```
data UusTüüp a b = Konstr1 Int | Konstr2 a Char b | Konstr3
```

Ehk siis:

- defineeritakse uued andmetüübid `UusTüüp a b`; näiteks
  - `UusTüüp Int Int`,
  - `UusTüüp Char (UusTüüp Char Int)`;
- defineeritakse konstruktor `Konstr1 :: Int -> UusTüüp a b`;
- defineeritakse konstruktor
  `Konstr2 :: a -> Char -> b -> UusTüüp a b`;
- defineeritakse konstruktor `Konstr3 :: UusTüüp a b`;

Mustrisobitus:

```
f Konstr3          = ...
f (Konstr1 x)      = ...
f (Konstr2 x y z) = ...
```

Näide

```
type Radius =  Float
type Width  =  Float
type Height =  Float

data Shape  =  Circle Radius  |  Rect Width Height      deriving Show

area :: Shape -> Float
area (Circle r)  =  pi * r^2
area (Rect w h)  =  w * h

s1 :: Shape
s1 = Circle 1.5

s2 :: Shape
s2 = Rect 0.75 3.0

test :: Float
test = area s1 + area s2
```

Näide

```
data Tree a = Empty  |  Branch a (Tree a) (Tree a)      deriving Show

flatten :: Tree a -> [a]
flatten Empty            = []
flatten (Branch x t1 t2) = flatten t1 ++ [x] ++ flatten t2

puu1 :: Tree Char
puu1 = Branch 'b' (Branch 'a' Empty Empty) (Branch 'c' Empty Empty)

puu2 = Branch 'd' puu1 Empty

puu3 = Branch 'x' puu3 puu3   -- lõpmatu puu

test = flatten puu1
```

newtype vs data

```
data    MyString1 = MyString1 String
newtype MyString2 = MyString2 String
```

- $|\text{MyString1}| = |\text{String}| + 1$
- MyString1 undefined /= undefined

- $|\text{MyString2}| = |\text{String}|$
- MyString2 undefined == undefined

Näide: Listid

Listide tüübiperet võime ette kujutada järgnevalt:

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

Näide: Listid

Listide tüübiperet võime ette kujutada järgnevalt:

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

Ehk siis:

- Iga tüübi `a` jaoks eksisteerib list `[a]`;
- tühja listi konstruktor `[] :: [a]`;
- mittetühja listi konstruktor `(:) :: a -> [a] -> [a]`.

Näide:

```
list_inf :: [Int]
list_inf = 1 : list_inf
```

Näide: nurjumisvõimalusega funktsioonid

Tüübipere `Maybe` on defineeritud järgnevalt:

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

Näide: nurjumisvõimalusega funktsioonid

Tüübipere `Maybe` on defineeritud järgnevalt:

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

Listist otsimise funktsioon:

```
lookup x []                    = Nothing
lookup x ((y,z):ys)| x==y      = Just z
                   | otherwise = lookup x ys
```

Näide: nurjumisvõimalusega funktsioonid

Tüübipere `Maybe` on defineeritud järgnevalt:

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

Listist otsimise funktsioon:

```
lookup x []                    = Nothing
lookup x ((y,z):ys)| x==y      = Just z
                   | otherwise = lookup x ys
```

Näiteks:

- `lookup 4 [(3,'x'),(4,'y')] == Just 'y'`
- `lookup 2 [(3,'x'),(4,'y')] == Nothing`

Näide: nurjumisvõimalusega funktsioonid

Tüübipere `Maybe` on defineeritud järgnevalt:

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

Listist otsimise funktsioon:

```
lookup x []                    = Nothing
lookup x ((y,z):ys)| x==y      = Just z
                   | otherwise = lookup x ys
```

Näiteks:

- `lookup 4 [(3,'x'),(4,'y')] == Just 'y'`
- `lookup 2 [(3,'x'),(4,'y')] == Nothing`

Tähelepanekud:

- `Maybe a` väärtusi on ühe võrra rohkem kui `a` väärtusi

Loe lisaks: RWH, peatükk 2, lk 41..55..

Näide: Tüüpide summa

Tüübid `Either a b` on defineeritud järgnevalt:

```
data Either a b = Left a | Right b deriving (Eq, Ord, Read, Show)
```

Näide: Tüüpide summa

Tüübid `Either a b` on defineeritud järgnevalt:

```
data  Either a b  =  Left a | Right b   deriving (Eq, Ord, Read, Show)
```

Kasutatakse näiteks siis kui on vaja edastada veateadet:

```
lookup' x []                      = Left "Elementi ei leitud!"
lookup' x ((y,z):ys)| x==y        = Right z
                    | otherwise = lookup' x ys
```

Näide: Tüüpide summa

Tüübid `Either a b` on defineeritud järgnevalt:

```
data  Either a b  =  Left a | Right b   deriving (Eq, Ord, Read, Show)
```

Kasutatakse näiteks siis kui on vaja edastada veateadet:

```
lookup' x []                      = Left "Elementi ei leitud!"
lookup' x ((y,z):ys)| x==y        = Right z
                    | otherwise = lookup' x ys
```

Näiteks:

- `lookup' 4 [(3,'x'),(4,'y')] == Right 'y'`
- `lookup' 2 [(3,'x'),(4,'y')] == Left "Elementi ei leitud"!

Näide: Tüüpide summa

Tüübid `Either a b` on defineeritud järgnevalt:

```
data Either a b = Left a | Right b deriving (Eq, Ord, Read, Show)
```

Kasutatakse näiteks siis kui on vaja edastada veateadet:

```
lookup' x []                      = Left "Elementi ei leitud!"
lookup' x ((y,z):ys)| x==y        = Right z
                    | otherwise = lookup' x ys
```

Näiteks:

- `lookup' 4 [(3,'x'),(4,'y')] == Right 'y'`
- `lookup' 2 [(3,'x'),(4,'y')] == Left "Elementi ei leitud"!`

Tähelepanekud:

- `Either a b` väärtusi on sama palju kui `a` väärtusi pluss `b` väärtusi.

Kirjetüüp

Selle mustri asemel

```
data Raamat = Raamat
                String   -- pealkiri
                [String] -- autorid
                Int      -- aasta
pealkiri (Raamat p _ _)   = p
autorid  (Raamat  _ as _) = as
aasta    (Raamat _ _ a)   = a

lisaAutor :: Raamat -> String -> Raamat
lisaAutor (Raamat p as a) x = Raamat p (x:as) a
```

Kirjetüüp

Selle mustri asemel

```
data Raamat = Raamat
                String   -- pealkiri
                [String] -- autorid
                Int      -- aasta
pealkiri (Raamat p _ _)   = p
autorid  (Raamat  _ as _) = as
aasta    (Raamat _ _ a)   = a

lisaAutor :: Raamat -> String -> Raamat
lisaAutor (Raamat p as a) x = Raamat p (x:as) a
```

saab Haskellis kirjutada ka nii

```
data Raamat = Raamat {
    pealkiri :: String
    autorid  :: [String]
    aasta    :: Int
  }
lisaAutor raamat x = raamat { autorid = x : autorid r }
```

Aritmeetilised jadad

- `aSeq1 = [1..5]`

Aritmeetilised jadad

- `aSeq1 = [1..5]`
  - `[1, 2, 3, 4, 5]`

Aritmeetilised jadad

- `aSeq1 = [1..5]`
  - `[1, 2, 3, 4, 5]`
- `aSeq2 = [0,2..10]`

Aritmeetilised jadad

- `aSeq1 = [1..5]`
  - `[1, 2, 3, 4, 5]`
- `aSeq2 = [0,2..10]`
  - `[0, 2, 4, 6, 8, 10]`

Aritmeetilised jadad

- `aSeq1 = [1..5]`
  - `[1, 2, 3, 4, 5]`
- `aSeq2 = [0,2..10]`
  - `[0, 2, 4, 6, 8, 10]`
- `aSeq3 = [0,2..11]`

Aritmeetilised jadad

- `aSeq1 = [1..5]`
  - `[1, 2, 3, 4, 5]`
- `aSeq2 = [0,2..10]`
  - `[0, 2, 4, 6, 8, 10]`
- `aSeq3 = [0,2..11]`
  - `[0, 2, 4, 6, 8, 10]`

Aritmeetilised jadad

- `aSeq1 = [1..5]`
  - `[1, 2, 3, 4, 5]`
- `aSeq2 = [0,2..10]`
  - `[0, 2, 4, 6, 8, 10]`
- `aSeq3 = [0,2..11]`
  - `[0, 2, 4, 6, 8, 10]`
- `aSeq4 = [5..1]`

Aritmeetilised jadad

- `aSeq1 = [1..5]`
  - `[1, 2, 3, 4, 5]`
- `aSeq2 = [0,2..10]`
  - `[0, 2, 4, 6, 8, 10]`
- `aSeq3 = [0,2..11]`
  - `[0, 2, 4, 6, 8, 10]`
- `aSeq4 = [5..1]`
  - `[]`

Aritmeetilised jadad

- `aSeq1 = [1..5]`
  - `[1, 2, 3, 4, 5]`
- `aSeq2 = [0,2..10]`
  - `[0, 2, 4, 6, 8, 10]`
- `aSeq3 = [0,2..11]`
  - `[0, 2, 4, 6, 8, 10]`
- `aSeq4 = [5..1]`
  - `[]`
- `aSeq5 = [10,7..(-3)]`

Aritmeetilised jadad

- `aSeq1 = [1..5]`
  - `[1, 2, 3, 4, 5]`
- `aSeq2 = [0,2..10]`
  - `[0, 2, 4, 6, 8, 10]`
- `aSeq3 = [0,2..11]`
  - `[0, 2, 4, 6, 8, 10]`
- `aSeq4 = [5..1]`
  - `[]`
- `aSeq5 = [10,7..(-3)]`
  - `[10, 7, 4, 1, -2]`

Aritmeetilised jadad

- `aSeq1 = [1..5]`
  - `[1, 2, 3, 4, 5]`
- `aSeq2 = [0,2..10]`
  - `[0, 2, 4, 6, 8, 10]`
- `aSeq3 = [0,2..11]`
  - `[0, 2, 4, 6, 8, 10]`
- `aSeq4 = [5..1]`
  - `[]`
- `aSeq5 = [10,7..(-3)]`
  - `[10, 7, 4, 1, -2]`
- `aSeq6 = [1..]`

Aritmeetilised jadad

- `aSeq1 = [1..5]`
  - `[1, 2, 3, 4, 5]`
- `aSeq2 = [0,2..10]`
  - `[0, 2, 4, 6, 8, 10]`
- `aSeq3 = [0,2..11]`
  - `[0, 2, 4, 6, 8, 10]`
- `aSeq4 = [5..1]`
  - `[]`
- `aSeq5 = [10,7..(-3)]`
  - `[10, 7, 4, 1, -2]`
- `aSeq6 = [1..]`
  - `[1, 2, 3, 4, 5, 6, 7, ...` lõpmatu list!

Aritmeetilised jadad

- `aSeq1 = [1..5]`
  - `[1, 2, 3, 4, 5]`
- `aSeq2 = [0,2..10]`
  - `[0, 2, 4, 6, 8, 10]`
- `aSeq3 = [0,2..11]`
  - `[0, 2, 4, 6, 8, 10]`
- `aSeq4 = [5..1]`
  - `[]`
- `aSeq5 = [10,7..(-3)]`
  - `[10, 7, 4, 1, -2]`
- `aSeq6 = [1..]`
  - `[1, 2, 3, 4, 5, 6, 7, ...` lõpmatu list!
- `aSeq7 = ['A'..'Z']`

Aritmeetilised jadad

- `aSeq1 = [1..5]`
  - `[1, 2, 3, 4, 5]`
- `aSeq2 = [0,2..10]`
  - `[0, 2, 4, 6, 8, 10]`
- `aSeq3 = [0,2..11]`
  - `[0, 2, 4, 6, 8, 10]`
- `aSeq4 = [5..1]`
  - `[]`
- `aSeq5 = [10,7..(-3)]`
  - `[10, 7, 4, 1, -2]`
- `aSeq6 = [1..]`
  - `[1, 2, 3, 4, 5, 6, 7, ...` lõpmatu list!
- `aSeq7 = ['A'..'Z']`
  - `"ABCDEFGHIJKLMNOPQRSTUVWXYZ"`

Listikomprehensioon

- `[x*x | x <- [1..5]] = [1, 4, 9, 16, 25]`

Listikomprehensioon

- `[x*x | x <- [1..5]] = [1, 4, 9, 16, 25]`
  - Terminoloogia: `x <- [1..5]` on "generaator"

Listikomprehensioon

- `[x*x | x <- [1..5]] = [1, 4, 9, 16, 25]`
  - Terminoloogia: `x <- [1..5]` on "generaator"
- `[x*x | x <- [1..10], even x] = [4, 16, 36, 64, 100]`

Listikomprehensioon

- `[x*x | x <- [1..5]] = [1, 4, 9, 16, 25]`
  - Terminoloogia: `x <- [1..5]` on "generaator"
- `[x*x | x <- [1..10], even x] = [4, 16, 36, 64, 100]`
  - Terminoloogia: `even x` on "valvur"

Listikomprehensioon

- `[x*x | x <- [1..5]] = [1, 4, 9, 16, 25]`
  - Terminoloogia: `x <- [1..5]` on "generaator"
- `[x*x | x <- [1..10], even x] = [4, 16, 36, 64, 100]`
  - Terminoloogia: `even x` on "valvur"
- `[i | (i,c) <- zip [1..] "HaSKell", isUpper c] = [1, 3, 4]`

Listikomprehensioon

- `[x*x | x <- [1..5]] = [1, 4, 9, 16, 25]`
  - Terminoloogia: `x <- [1..5]` on "generaator"
- `[x*x | x <- [1..10], even x] = [4, 16, 36, 64, 100]`
  - Terminoloogia: `even x` on "valvur"
- `[i | (i,c) <- zip [1..] "HaSKell", isUpper c] = [1, 3, 4]`
- ```
  [(x,y) | x <- [1..2], y <- [1..3]] =
      [(1,1), (1,2), (1,3),(2,1), (2,2), (2,3)]
  ```
  ```

Listikomprehensioon

- `[x*x | x <- [1..5]] = [1, 4, 9, 16, 25]`
  - Terminoloogia: `x <- [1..5]` on "generaator"
- `[x*x | x <- [1..10], even x] = [4, 16, 36, 64, 100]`
  - Terminoloogia: `even x` on "valvur"
- `[i | (i,c) <- zip [1..] "HaSKell", isUpper c] = [1, 3, 4]`
- ```
  [(x,y) | x <- [1..2], y <- [1..3]] =
      [(1,1), (1,2), (1,3),(2,1), (2,2), (2,3)]
  ```
- ```
  [(x,y) | y <- [1..3], x <- [1..2]] =
      [(1,1),(2,1),(1,2),(2,2),(1,3),(2,3)]
  ```

Listikomprehensioon

- `[x*x | x <- [1..5]] = [1, 4, 9, 16, 25]`
  - Terminoloogia: `x <- [1..5]` on "generaator"
- `[x*x | x <- [1..10], even x] = [4, 16, 36, 64, 100]`
  - Terminoloogia: `even x` on "valvur"
- `[i | (i,c) <- zip [1..] "HaSKell", isUpper c] = [1, 3, 4]`
- ```
  [(x,y) | x <- [1..2], y <- [1..3]] =
      [(1,1), (1,2), (1,3),(2,1), (2,2), (2,3)]
  ```
- ```
  [(x,y) | y <- [1..3], x <- [1..2]] =
      [(1,1),(2,1),(1,2),(2,2),(1,3),(2,3)]
  ```
  - NB! Parempoolne generaator muutub vasakpoolsest kiiremini!

Listikomprehensioon

- `[x*x | x <- [1..5]] = [1, 4, 9, 16, 25]`
  - Terminoloogia: `x <- [1..5]` on "generaator"
- `[x*x | x <- [1..10], even x] = [4, 16, 36, 64, 100]`
  - Terminoloogia: `even x` on "valvur"
- `[i | (i,c) <- zip [1..] "HaSKell", isUpper c] = [1, 3, 4]`
- ```
  [(x,y) | x <- [1..2], y <- [1..3]] =
      [(1,1), (1,2), (1,3),(2,1), (2,2), (2,3)]
  ```
- ```
  [(x,y) | y <- [1..3], x <- [1..2]] =
      [(1,1),(2,1),(1,2),(2,2),(1,3),(2,3)]
  ```
  - NB! Parempoolne generaator muutub vasakpoolsest kiiremini!
- ```
  [(x,y) | x <- [1..4], y <- [x+1..4]] =
      [(1,2), (1,3), (1,4),(2,3), (2,4), (3,4)]
  ```
  ```

Listikomprehensioon

- `[x*x | x <- [1..5]] = [1, 4, 9, 16, 25]`
  - Terminoloogia: `x <- [1..5]` on "generaator"
- `[x*x | x <- [1..10], even x] = [4, 16, 36, 64, 100]`
  - Terminoloogia: `even x` on "valvur"
- `[i | (i,c) <- zip [1..] "HaSKell", isUpper c] = [1, 3, 4]`
- ```
  [(x,y) | x <- [1..2], y <- [1..3]] =
      [(1,1), (1,2), (1,3),(2,1), (2,2), (2,3)]
  ```
- ```
  [(x,y) | y <- [1..3], x <- [1..2]] =
      [(1,1),(2,1),(1,2),(2,2),(1,3),(2,3)]
  ```
  - NB! Parempoolne generaator muutub vasakpoolsest kiiremini!
- ```
  [(x,y) | x <- [1..4], y <- [x+1..4]] =
      [(1,2), (1,3), (1,4),(2,3), (2,4), (3,4)]
  ```
- `[y*z | x <- [1..4], let z = x+1, y <- [z..4]] = [4, 6, 8, 9, 12, 16]`
  ```

Tüübiklassid I

Haskellis tuleb tihti kirjutada funktsioone, mis erinevad ainult natuke tüübi poolest:

```
equalChar   ::   Char ->   Char -> Bool
equalInt    ::    Int ->    Int -> Bool
equalString :: String -> String -> Bool
```

Tüübiklassid I

Haskellis tuleb tihti kirjutada funktsioone, mis erinevad ainult natuke tüübi poolest:

```
equalChar   ::   Char ->   Char -> Bool
equalInt    ::    Int ->    Int -> Bool
equalString :: String -> String -> Bool
```

Sellist koodi *ei saa* kirjutada parameetrilise polümorfse funktsiooniga, kuna funktsioonide implementatsioon on erinev:

```
equal :: a -> a -> Bool
equal = ???  -- pole def. mis töötaks iga tüübi korral
```

Tüübiklassid I

Haskellis tuleb tihti kirjutada funktsioone, mis erinevad ainult natuke tüübi poolest:

```
equalChar   ::   Char ->   Char -> Bool
equalInt    ::    Int ->    Int -> Bool
equalString :: String -> String -> Bool
```

Sellist koodi *ei saa* kirjutada parameetrilise polümorfse funktsiooniga, kuna funktsioonide implementatsioon on erinev:

```
equal :: a -> a -> Bool
equal = ???  -- pole def. mis töötaks iga tüübi korral
```

Selleks ongi loodud tüübiklassid

```
class Equal a where
  equal :: a -> a -> Bool
instance Equal Char where
  equal = ... -- :: Char -> Char -> Bool
instance Equal Int where
  equal = ... -- :: Int -> Int -> Bool
...
```

Näide

```
class Equal a where
  equal :: a -> a -> Bool

data ValgusFoor = Punane | Kollane | Roheline

instance Equal ValgusFoor where
  equal Punane   Punane   = True
  equal Kollane  Kollane  = True
  equal Roheline Roheline = True
  equal _        _        = False

test :: Bool
test = Kollane `equal' Roheline   -- False
```

Tüübiklassid II

Haskelli standardteegis on defineeritud tüübiklass `Eq`:

```
class Eq a where
  (==), (/=) :: a -> a -> Bool

  -- Minimaalne definitioon peab sisaldama:
  -- (==) või (/=)
  x /= y = not (x == y) -- vaikedefinitsioon
  x == y = not (x /= y) -- vaikedefinitsioon
```

Tüübiklassid II

Haskelli standardteegis on defineeritud tüübiklass `Eq`:

```
class Eq a where
  (==), (/=) :: a -> a -> Bool

  -- Minimaalne definitioon peab sisaldama:
  -- (==) või (/=)
  x /= y = not (x == y) -- vaikedefinitsioon
  x == y = not (x /= y) -- vaikedefinitsioon
```

Võrldusoperaatori tüüp on:

```
(==) :: Eq a => a -> a -> Bool
```

s.t me saame operaatorit kasutada, kui tema argumentitüübil on defineeritud `Eq` instants!

Tüübiklassid II

Haskelli standardteegis on defineeritud tüübiklass `Eq`:

```
class Eq a where
  (==), (/=) :: a -> a -> Bool

  -- Minimaalne definitioon peab sisaldama:
  -- (==) või (/=)
  x /= y = not (x == y) -- vaikedefinitsioon
  x == y = not (x /= y) -- vaikedefinitsioon
```

Võrldusoperaatori tüüp on:

```
(==) :: Eq a => a -> a -> Bool
```

s.t me saame operaatorit kasutada, kui tema argumentitüübil on defineeritud `Eq` instants!

Kõikidel polümorfsetel funktsioonidel, mis kasutavad võrdust peab tüübi kontekst olema `Eq`:

```
lookup :: Eq a => a -> [(a, b)] -> Maybe b
```

Näide

```
data ValgusFoor = Punane | Kollane | Roheline

instance Eq ValgusFoor where
  Punane   == Punane   = True
  Kollane  == Kollane  = True
  Roheline == Roheline = True
  _        == _        = False

test :: Bool
test = Kollane == Roheline   -- False
```

Tüübiklasside automaatne defineerimine

Osade tüübiklasside definitsioonid on keerukamad, kui tundub, et nad peaks olema

Tüübiklasside automaatne defineerimine

Osade tüübiklasside definitsioonid on keerukamad, kui tundub, et nad peaks olema

```
class Read a where
  readsPrec :: Int -> ReadS a
  readList :: ReadS [a]
  GHC.Read.readPrec :: Text.ParserCombinators.ReadPrec.ReadPrec a
  GHC.Read.readListPrec ::  Text.ParserCombinators.ReadPrec.ReadPrec [a]

class Show a where
  showsPrec :: Int -> a -> ShowS
  show :: a -> String
  showList :: [a] -> ShowS
```

Tüübiklasside automaatne defineerimine

Osade tüübiklasside definitsioonid on keerukamad, kui tundub, et nad peaks olema

```
class Read a where
  readsPrec :: Int -> ReadS a
  readList :: ReadS [a]
  GHC.Read.readPrec :: Text.ParserCombinators.ReadPrec.ReadPrec a
  GHC.Read.readListPrec ::  Text.ParserCombinators.ReadPrec.ReadPrec [a]

class Show a where
  showsPrec :: Int -> a -> ShowS
  show :: a -> String
  showList :: [a] -> ShowS
```

... kuigi, neid tüübiklasse kasutatakse suuresti ainult kahe funktsiooni jaoks:

```
read :: Read a => String -> a  -- parsib väärtuse sõnest
show :: Show a => a -> String  -- muudab väärtuse sõneks
```

Tüübiklasside automaatne defineerimine

Osade tüübiklasside definitsioonid on keerukamad, kui tundub, et nad peaks olema

```
class Read a where
  readsPrec :: Int -> ReadS a
  readList :: ReadS [a]
  GHC.Read.readPrec :: Text.ParserCombinators.ReadPrec.ReadPrec a
  GHC.Read.readListPrec ::  Text.ParserCombinators.ReadPrec.ReadPrec [a]

class Show a where
  showsPrec :: Int -> a -> ShowS
  show :: a -> String
  showList :: [a] -> ShowS
```

... kuigi, neid tüübiklasse kasutatakse suuresti ainult kahe funktsiooni jaoks:

```
read :: Read a => String -> a  -- parsib väärtuse sõnest
show :: Show a => a -> String  -- muudab väärtuse sõneks
```

Standardteegi tüübiklasse nagu `Eq`, `Show`, `Read`, `Ord`, `Enum` saab lasta defineerida automaatselt. Näiteks:

Tüübiklasside automaatne defineerimine

Osade tüübiklasside definitsioonid on keerukamad, kui tundub, et nad peaks olema

```
class Read a where
  readsPrec :: Int -> ReadS a
  readList :: ReadS [a]
  GHC.Read.readPrec :: Text.ParserCombinators.ReadPrec.ReadPrec a
  GHC.Read.readListPrec ::  Text.ParserCombinators.ReadPrec.ReadPrec [a]

class Show a where
  showsPrec :: Int -> a -> ShowS
  show :: a -> String
  showList :: [a] -> ShowS
```

... kuigi, neid tüübiklasse kasutatakse suuresti ainult kahe funktsiooni jaoks:

```
read :: Read a => String -> a  -- parsib väärtuse sõnest
show :: Show a => a -> String  -- muudab väärtuse sõneks
```

Standardteegi tüübiklasse nagu `Eq`, `Show`, `Read`, `Ord`, `Enum` saab lasta defineerida automaatselt. Näiteks:

```
data Loom = Kass | Koer | Muu String    deriving (Show, Eq)
```

Loe lisaks: RWH, peatükk 6; LYaH, peatükk 8

Näide

```
data ValgusFoor = Punane | Kollane | Roheline  deriving (Eq)

test :: Bool
test = Kollane == Roheline   -- False
```

Standardsed tüübiklassid

- `Eq`
  - `(==), (/=) :: Eq a => a -> a -> Bool`
- `Ord`
  - `(<=) :: Ord a => a -> a -> Bool`
  - `min, max :: Ord a => a -> a -> a` ...
- `Show`
  - `show :: Show a => a -> String` ...
- `Read`
  - `read :: Read a => String -> a` ...
- `Ix`
  - `range :: (a, a) -> [a]`
  - `index :: (a, a) -> a -> Int` ...

Enum tüübiklass

Enumeratsioone kirjeldab järgnev tüübiklass:

```
class Enum a where
  succ :: a -> a
  pred :: a -> a
  toEnum :: Int -> a
  fromEnum :: a -> Int
  enumFrom :: a -> [a]                    -- [x ..]      = enumFrom x
  enumFromThen :: a -> a -> [a]           -- [x, y ..]   = enumFromThen x y
  enumFromTo :: a -> a -> [a]             -- [x .. y]    = enumFromTo x y
  enumFromThenTo :: a -> a -> a -> [a]    -- [x, y .. z] = enumFromThenTo x y z

class Bounded a where
  minBound :: a
  maxBound :: a
```

Enum tüübiklass

Enumeratsioone kirjeldab järgnev tüübiklass:

```
class Enum a where
  succ :: a -> a
  pred :: a -> a
  toEnum :: Int -> a
  fromEnum :: a -> Int
  enumFrom :: a -> [a]                    -- [x ..]      = enumFrom x
  enumFromThen :: a -> a -> [a]           -- [x, y ..]   = enumFromThen x y
  enumFromTo :: a -> a -> [a]             -- [x .. y]    = enumFromTo x y
  enumFromThenTo :: a -> a -> a -> [a]    -- [x, y .. z] = enumFromThenTo x y z

class Bounded a where
  minBound :: a
  maxBound :: a
```

- Tüübiklass `Enum` on tuletatav, kui konstruktoritel pole argumente!

```
data Värvid = Punane | Sinine | Kollane   deriving (Enum)
```

Enum tüübiklass

Enumeratsioone kirjeldab järgnev tüübiklass:

```
class Enum a where
  succ :: a -> a
  pred :: a -> a
  toEnum :: Int -> a
  fromEnum :: a -> Int
  enumFrom :: a -> [a]                    -- [x ..]      = enumFrom x
  enumFromThen :: a -> a -> [a]           -- [x, y ..]   = enumFromThen x y
  enumFromTo :: a -> a -> [a]             -- [x .. y]    = enumFromTo x y
  enumFromThenTo :: a -> a -> a -> [a]    -- [x, y .. z] = enumFromThenTo x y z

class Bounded a where
  minBound :: a
  maxBound :: a
```

- Tüübiklass `Enum` on tuletatav, kui konstruktoritel pole argumente!

```
data Värvid = Punane | Sinine | Kollane   deriving (Enum)
```

- `Enum`-id on tihti ka `Bounded`

Abstraktsed andmestruktuurid

- Mitmed senimaani vaadatud andmestruktuurid on implementeeritud Haskellis.
  - `Bool`, `()`, `Maybe a`, `[a]`, `Either a b` jne.
  - Konstruktorid otse kasutatavad.

Abstraktsed andmestruktuurid

- Mitmed senimaani vaadatud andmestruktuurid on implementeeritud Haskellis.
  - `Bool`, `()`, `Maybe a`, `[a]`, `Either a b` jne.
  - Konstruktorid otse kasutatavad.
- Osad on implementeeritd madalamal tasemel, näiteks:
  - `Int`, `Integer`, `Char`, `Float`, `Double`
  - Konstruktorid peidetud — tüübid abstraktsed!

Abstraktsed andmestruktuurid

- Mitmed senimaani vaadatud andmestruktuurid on implementeeritud Haskellis.
  - `Bool`, `()`, `Maybe a`, `[a]`, `Either a b` jne.
  - Konstruktorid otse kasutatavad.
- Osad on implementeeritd madalamal tasemel, näiteks:
  - `Int`, `Integer`, `Char`, `Float`, `Double`
  - Konstruktorid peidetud — tüübid abstraktsed!
- Vahetevahel on mõistlik luua ise abstraktseid andmestruktuure.

Abstraktsed andmestruktuurid

- Mitmed senimaani vaadatud andmestruktuurid on implementeeritud Haskellis.
  - `Bool`, `()`, `Maybe a`, `[a]`, `Either a b` jne.
  - Konstruktorid otse kasutatavad.
- Osad on implementeeritd madalamal tasemel, näiteks:
  - `Int`, `Integer`, `Char`, `Float`, `Double`
  - Konstruktorid peidetud — tüübid abstraktsed!
- Vahetevahel on mõistlik luua ise abstraktseid andmestruktuure.
  - Näiteks `Data.Map.Lazy`, `Data.Set`
    - Sellisel puhul ei ekspordita moodulist konstruktorite nimesid

Abstraktsed andmestruktuurid

- Mitmed senimaani vaadatud andmestruktuurid on implementeeritud Haskellis.
  - `Bool`, `()`, `Maybe a`, `[a]`, `Either a b` jne.
  - Konstruktorid otse kasutatavad.
- Osad on implementeeritd madalamal tasemel, näiteks:
  - `Int`, `Integer`, `Char`, `Float`, `Double`
  - Konstruktorid peidetud — tüübid abstraktsed!
- Vahetevahel on mõistlik luua ise abstraktseid andmestruktuure.
  - Näiteks `Data.Map.Lazy`, `Data.Set`
    - Sellisel puhul ei ekspordita moodulist konstruktorite nimesid
    - … kuid eksporditakse muid funktsioone:

```
empty :: Set a
null  :: Set a -> Bool
singleton :: a -> Set a
insert :: Ord a => a -> Set a -> Set a
delete :: Ord a => a -> Set a -> Set a
foldr  :: (a -> b -> b) -> b -> Set a -> b
...
```

Sisend-väljund Haskellis

- Haskelli puhtad funktsioonid ei võimalda teha mittepuhtaid arvutusi.
  - Puhas — funktsiooni tulemus sõltub ainult argumentide väärtusest.
  - Ei saa teha näiteks juhuarvude funktsiooni `random :: () -> Int`

Sisend-väljund Haskellis

- Haskelli puhtad funktsioonid ei võimalda teha mittepuhtaid arvutusi.
  - Puhas — funktsiooni tulemus sõltub ainult argumentide väärtusest.
  - Ei saa teha näiteks juhuarvude funktsiooni `random :: () -> Int`
- Lahendus: `IO` *monaad*
  - ’`IO a`’ tüüpi väärtus — “masin mis arvutab `a` tüüpi väärtuse”

Sisend-väljund Haskellis

- Haskelli puhtad funktsioonid ei võimalda teha mittepuhtaid arvutusi.
  - Puhas — funktsiooni tulemus sõltub ainult argumentide väärtusest.
  - Ei saa teha näiteks juhuarvude funktsiooni `random :: () -> Int`
- Lahendus: `IO` *monaad*
  - ’`IO a`’ tüüpi väärtus — “masin mis arvutab `a` tüüpi väärtuse”
  - `return :: a -> IO a` — masina tagastab esimese argumendi väärtuse

Sisend-väljund Haskellis

- Haskelli puhtad funktsioonid ei võimalda teha mittepuhtaid arvutusi.
  - Puhas — funktsiooni tulemus sõltub ainult argumentide väärtusest.
  - Ei saa teha näiteks juhuarvude funktsiooni `random :: () -> Int`
- Lahendus: `IO` *monaad*
  - ’`IO a`’ tüüpi väärtus — “masin mis arvutab `a` tüüpi väärtuse”
  - `return :: a -> IO a` — masina tagastab esimese argumendi väärtuse
  - `(>>=) :: IO a -> (a -> IO b) -> IO b` — masin käivitab esimese argumendi ja rakendab tulemuse teisele

Sisend-väljund Haskellis

- Haskelli puhtad funktsioonid ei võimalda teha mittepuhtaid arvutusi.
  - Puhas — funktsiooni tulemus sõltub ainult argumentide väärtusest.
  - Ei saa teha näiteks juhuarvude funktsiooni `random :: () -> Int`
- Lahendus: `IO` *monaad*
  - ’`IO a`’ tüüpi väärtus — “masin mis arvutab `a` tüüpi väärtuse”
  - `return :: a -> IO a` — masina tagastab esimese argumendi väärtuse
  - `(>>=) :: IO a -> (a -> IO b) -> IO b` — masin käivitab esimese argumendi ja rakendab tulemuse teisele
  - … lisaks baasfunktsioonid nagu `putStrLn :: String -> IO ()` ja `getLine :: IO String`.

Sisend-väljund Haskellis

- Haskelli puhtad funktsioonid ei võimalda teha mittepuhtaid arvutusi.
  - Puhas — funktsiooni tulemus sõltub ainult argumentide väärtusest.
  - Ei saa teha näiteks juhuarvude funktsiooni `random :: () -> Int`
- Lahendus: `IO` *monaad*
  - ’`IO a`’ tüüpi väärtus — “masin mis arvutab `a` tüüpi väärtuse”
  - `return :: a -> IO a` — masina tagastab esimese argumendi väärtuse
  - `(>>=) :: IO a -> (a -> IO b) -> IO b` — masin käivitab esimese argumendi ja rakendab tulemuse teisele
  - … lisaks baasfunktsioonid nagu `putStrLn :: String -> IO ()` ja `getLine :: IO String`.
- Nii saab kombineerida olemasolevaid `IO` “masinaid”. Näiteks:

```
main :: IO ()
main = randomRIO (1, 10) >>= classify >>= putStrLn
  where classify :: Int -> IO String
        classify x | odd x     = return "paaritu"
                   | otherwise = return "paaris"
```

do-süntaks I

Eelneval slaidil olnud koodi on keeruline lugeda ja kirjutada:

```
main :: IO ()
main = randomRIO (1, 10) >>= classify >>= putStrLn
  where classify :: Int -> IO String
        classify x | odd x     = return "paaritu"
                   | otherwise = return "paaris"
```

do-süntaks I

Eelneval slaidil olnud koodi on keeruline lugeda ja kirjutada:

```
main :: IO ()
main = randomRIO (1, 10) >>= classify >>= putStrLn
  where classify :: Int -> IO String
        classify x | odd x     = return "paaritu"
                   | otherwise = return "paaris"
```

Sama saab saavutada järgnevalt

```
main :: IO ()
main = do
  r <- randomRIO (1, 10)
  c <- classify r
  putStrLn c
    where classify :: Int -> IO String
          classify x | odd x     = return "paaritu"
                     | otherwise = return "paaris"
```

või

```
main = do
  r <- randomRIO (1, 10)
  if odd r
    then putStrLn "paaris"
    else putStrLn "paaritu"
```

do-süntaks II

Näide

```
proc = do
  s <- getLine
  let n = read s
      n2 = 2*n
  putStrLn ("Kaks korda " ++ s ++ " on " ++ show n2)
```

Do-süntaks algab **do**-võtmesõnaga, millele järgnevad *järjest töödeldavad* laused.

do-süntaks II

Näide

```
proc = do
  s <- getLine
  let n = read s
      n2 = 2*n
  putStrLn ("Kaks korda " ++ s ++ " on " ++ show n2)
```

Do-süntaks algab **do**-võtmesõnaga, millele järgnevad *järjest töödeldavad* laused.

- Laused mustriga x `<-` p, kus p :: `IO` a siis x :: a,

do-süntaks II

Näide

```
proc = do
  s <- getLine
  let n = read s
      n2 = 2*n
  putStrLn ("Kaks korda " ++ s ++ " on " ++ show n2)
```

Do-süntaks algab **do**-võtmesõnaga, millele järgnevad *järjest töödeldavad* laused.

- Laused mustriga x `<-` p, kus p :: `IO` a siis x :: a,
- **let** laused ning

do-süntaks II

Näide

```
proc = do
  s <- getLine
  let n = read s
      n2 = 2*n
  putStrLn ("Kaks korda " ++ s ++ " on " ++ show n2)
```

Do-süntaks algab **do**-võtmesõnaga, millele järgnevad *järjest töödeldavad* laused.

- Laused mustriga x `<-` p, kus p :: `IO` a siis x :: a,
- **let** laused ning
- avaldised e, mille tüüp on `IO a`.

Mitme **do** kasutamine

- **do** seob kokku IO-avaldised, kuid ei saa vaadata konstruktsioonide sisse
- S.t. ühe avaldise jaoks pole do-d vaja
  - `main = putStrLn "Hello World"!`
- Hargenmise puhul võib olla vaja kasutada mitut do-d:

```
main = do
  putStrLn "Kirjuta midagi!"
  xs <- getLine
  if (xs=="")
    then putStr "Sõnakuulmatu!"
    else do
      putStrLn "Tänan!"
      putStrLn ("Kirjutasid: " ++ xs)
```

- do-süntaks on lihtne, kuid vajab harjutamist!

do tähendus

- `a >>= f` on sama mis

```
do x <- a
   f x
```

- `a >> b` on sama mis

```
do a
   b
```

-

```
do a                    do do a
   b   on sama mis             b
   c                        c
```

- Fixity:

```
infixl 1 >>
infixl 1 >>=
```

Näide

```
main = do
 putStrLn "Kirjuta midagi!"
 xs <- getLine
 if (xs=="")
   then putStr "Sõnakuulmatu!"
   else do
     putStrLn "Tänan!"
     putStrLn ("Kirjutasid: " ++ xs)
```

on sama mis

```
main =
 putStrLn "Kirjuta midagi!" >>
 getLine  >>= (\ xs ->
 if (xs=="")
   then putStr "Sõnakuulmatu!"
   else
     putStrLn "Tänan!" >>
     putStrLn ("Kirjutasid: " ++ xs)
 )
```

Näide

```
main =
 putStrLn "Kirjuta midagi!" >>
 getLine  >>= (\ xs ->
 if (xs=="")
   then putStr "Sõnakuulmatu!"
   else
     putStrLn "Tänan!" >>
     putStrLn ("Kirjutasid: " ++ xs)
 )
```

on sama mis

```
main =
   putStrLn "Kirjuta midagi!" >>  getLine  >>= ifThenElse
     where ifThenElse xs = if (xs=="") then case1 else case2 xs
           case1         = putStr "Sõnakuulmatu!"
           case2 xs      = putStrLn "Tänan!" >> putStrLn ("Kirjutasid: " ++ xs)
```

Loe lisaks: RWH, peatükk 7 (algus)

Reduktsioon IO monaadis

- Üldised reeglid kehtivad aga saab natuke lihtsustada.

Reduktsioon IO monaadis

- Üldised reeglid kehtivad aga saab natuke lihtsustada.
- Kui redexiks on avaldis *e* tüübist `IO a`, tuleb kõrvalefekt enda peas teha ja asendada tulemus tagasi avaldisse.
  - Näiteks `putStrLn "Tere!"` trükib välja "Tere!", peale mille tuleb avaldis asendada väärtusega `()`.

Reduktsioon IO monaadis

- Üldised reeglid kehtivad aga saab natuke lihtsustada.
- Kui redexiks on avaldis *e* tüübist `IO a`, tuleb kõrvalefekt enda peas teha ja asendada tulemus tagasi avaldisse.
  - Näiteks `putStrLn "Tere!"` trükib välja "Tere!", peale mille tuleb avaldis asendada väärtusega `()`.
- Kuna IO sunnib peale kindla järjestuse võime endi tööd natuke lihtsustada: redutseeritavaid IO avaldisi ei pea iga sammu järel programmi tagasi paigutama. Näiteks

```
printFirst 0 _      = return ()
printFirst n (x:xs) = do  print x
                          printFirst (n-1) xs

main = do  printFirst 3 [1..]
           putStrLn "Kõik!"
```

Reduktsioon IO monaadis

- Üldised reeglid kehtivad aga saab natuke lihtsustada.
- Kui redexiks on avaldis *e* tüübist `IO a`, tuleb kõrvalefekt enda peas teha ja asendada tulemus tagasi avaldisse.
  - Näiteks `putStrLn "Tere!"` trükib välja "Tere!", peale mille tuleb avaldis asendada väärtusega `()`.
- Kuna IO sunnib peale kindla järjestuse võime endi tööd natuke lihtsustada: redutseeritavaid IO avaldisi ei pea iga sammu järel programmi tagasi paigutama. Näiteks

```
printFirst 0 _      = return ()
printFirst n (x:xs) = do  print x
                          printFirst (n-1) xs

main = do  printFirst 3 [1..]
           putStrLn "Kõik!"
```

- Kõigepealt arvutame `printFirst 3 [1..]` ja alles siis pöördume tagasi `main`-i juurde

Monaadidest üldisemalt

Haskellis in monaad on ühe muutujaga tüübipere, mille jaoks on defineeritud järgnevad funktsioonid:

```
class Functor f where
  fmap :: (a -> b) -> f a -> f b
class Functor f => Applicative f where
  pure :: a -> f a
  (<*>) :: f (a -> b) -> f a -> f b
class Applicative m => Monad m where
  (>>=) :: m a -> (a -> m b) -> m b
  return = pure   -- monaadis kasutatakse funktsiooni return
```

Monaadidest üldisemalt

Haskellis in monaad on ühe muutujaga tüübipere, mille jaoks on defineeritud järgnevad funktsioonid:

```
class Functor f where
  fmap :: (a -> b) -> f a -> f b
class Functor f => Applicative f where
  pure :: a -> f a
  (<*>) :: f (a -> b) -> f a -> f b
class Applicative m => Monad m where
  (>>=) :: m a -> (a -> m b) -> m b
  return = pure   -- monaadis kasutatakse funktsiooni return
```

Intuitsioon: Tüüp `m a` on nagu konteiner, kuhu saab `a` tüüpi väärtust hoida. (Aga igal `m a` ei pruugi sisaldada `a` tüüpi väärtust.)

Monaadidest üldisemalt

Haskellis in monaad on ühe muutujaga tüübipere, mille jaoks on defineeritud järgnevad funktsioonid:

```
class Functor f where
  fmap :: (a -> b) -> f a -> f b
class Functor f => Applicative f where
  pure :: a -> f a
  (<*>) :: f (a -> b) -> f a -> f b
class Applicative m => Monad m where
  (>>=) :: m a -> (a -> m b) -> m b
  return = pure   -- monaadis kasutatakse funktsiooni return
```

Intuitsioon: Tüüp `m a` on nagu konteiner, kuhu saab `a` tüüpi väärtust hoida. (Aga igal `m a` ei pruugi sisaldada `a` tüüpi väärtust.)

- Tahame vältida eksplitsiitselt väärtuse konteinerist välja võtmist.

Functor

```
class Functor f where
  fmap :: (a -> b) -> f a -> f b
```

Võrrandid:

```
fmap id  ==  id
fmap (f . g)  ==  fmap f . fmap g
```

Intuitsioon: igasugune konteiner f a, kus a väärtus ei interakteeru konteineriga

Näide: listid, `Maybe`

Functor

```
class Functor f where
  fmap :: (a -> b) -> f a -> f b
```

Võrrandid:

```
fmap id  ==  id
fmap (f . g)  ==  fmap f . fmap g
```

Intuitsioon: igasugune konteiner f a, kus a väärtus ei interakteeru konteineriga

Näide: listid, `Maybe`

Applicative

```
class Functor f => Applicative f where
  pure :: a -> f a
  (<*>) :: f (a -> b) -> f a -> f b
```

Võrrandid:

```
                pure id <*> v == v
pure (.) <*> u <*> v <*> w == u <*> (v <*> w)
          pure f <*> pure x == pure (f x)
               u <*> pure y == pure ($ y) <*> u
```

Intuitsioon: saame väärtusi konteinerisse panna + võime konteineri sees funktsioone rakendada

Näide: Pilveserver (AWS)

Applicative

```
class Functor f => Applicative f where
  pure :: a -> f a
  (<*>) :: f (a -> b) -> f a -> f b
```

Võrrandid:

```
                pure id <*> v == v
pure (.) <*> u <*> v <*> w == u <*> (v <*> w)
          pure f <*> pure x == pure (f x)
               u <*> pure y == pure ($ y) <*> u
```

Intuitsioon: saame väärtusi konteinerisse panna + võime konteineri sees funktsioone rakendada

Näide: Pilveserver (AWS)

Monad

```
class Applicative m => Monad m where
  (>>=) :: m a -> (a -> m b) -> m b
  return = pure   -- monaadis kasutatakse funktsiooni return
```

Võrrandid:

```
return a >>= k  ==  k a
  m >>= return  ==  m
m >>= (\x -> k x >>= h)  ==  (m >>= k) >>= h
```

Intuitsioon: järjest rakendatavad masinad

Näide: IO

Monad

```
class Applicative m => Monad m where
  (>>=) :: m a -> (a -> m b) -> m b
  return = pure   -- monaadis kasutatakse funktsiooni return
```

Võrrandid:

```
return a >>= k  ==  k a
  m >>= return  ==  m
m >>= (\x -> k x >>= h)  ==  (m >>= k) >>= h
```

Intuitsioon: järjest rakendatavad masinad

Näide: IO

Maybe monaad

```haskell
data Maybe a = Nothing | Just a deriving (Eq, Ord)
```

Maybe monaad

```
data Maybe a = Nothing | Just a deriving (Eq, Ord)
```

- intuitsioon: `Nothing` – viga, nurjumine

Maybe monaad

```
data Maybe a = Nothing | Just a deriving (Eq, Ord)
```

- intuitsioon: `Nothing` – viga, nurjumine
- definitsioon

```
return x  =  Just x

(Just a) >>= f  =  f a
Nothing  >>= f  =  Nothing
```

Maybe monaad

```
data Maybe a = Nothing | Just a deriving (Eq, Ord)
```

- intuitsioon: `Nothing` – viga, nurjumine
- definitsioon

```
return x  =  Just x

(Just a) >>= f  =  f a
Nothing  >>= f  =  Nothing
```

- Näide (pseudokood):

```
getTaxOwed name = do
  number       <- lookup name phonebook
  registration <- lookup number governmentDatabase
  lookup registration taxDatabase
```

Listi monaad

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

Listi monaad

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

- intuitsioon: mitmesus

Listi monaad

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

- intuitsioon: mitmesus
- definitsioon

```
return x   =  [x]

xs  >>= f  =  concat (map f xs)
```

Listi monaad

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

- intuitsioon: mitmesus
- definitsioon

```
return x   =  [x]

xs  >>= f  =  concat (map f xs)
```

- Näide (pseudokood):

```
sõpradePalgad :: Person -> [Int]
sõpradePalgad isik = do
  sõber <- getFriends isik
  töö   <- getEmployers sõber
  return (getPay töö sõber)
```

Listi monaad

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

- intuitsioon: mitmesus
- definitsioon

```
return x   =  [x]

xs  >>= f  =  concat (map f xs)
```

- Näide (pseudokood):

```
sõpradePalgad :: Person -> [Int]
sõpradePalgad isik = do
  sõber <- getFriends isik
  töö   <- getEmployers sõber
  return (getPay töö sõber)
```

- … see on sama mis listikomprehensioon

Parsimise monaad

```
type Parser a = String -> [(a,String)]
```

Parsimise monaad

```
type Parser a = String -> [(a,String)]
return x = \ s -> [(x,s)]
```

Parsimise monaad

```
type Parser a = String -> [(a,String)]
return x = \ s -> [(x,s)]
p >>= g => \ s -> concatMap (\ (a,s) -> g a s) (p s)
```

Näide:

Parsimise monaad

```
type Parser a = String -> [(a,String)]
return x = \ s -> [(x,s)]
p >>= g => \ s -> concatMap (\ (a,s) -> g a s) (p s)
```

Näide:

```
expr =     do n <- term
              keyw "+"
              m <- expr
              return (n+m)
       <|> term
```

Parsimise monaad

```
type Parser a = String -> [(a,String)]
return x = \ s -> [(x,s)]
p >>= g => \ s -> concatMap (\ (a,s) -> g a s) (p s)
```

Näide:

```
expr =     do n <- term
              keyw "+"
              m <- expr
              return (n+m)
       <|> term
term =     do n <- atom
              keyw "*"
              m <- term
              return (n*m)
       <|> atom
```

Parsimise monaad

```
type Parser a = String -> [(a,String)]
return x = \ s -> [(x,s)]
p >>= g => \ s -> concatMap (\ (a,s) -> g a s) (p s)
```

Näide:

```
expr =     do n <- term
              keyw "+"
              m <- expr
              return (n+m)
       <|> term
term =     do n <- atom
              keyw "*"
              m <- term
              return (n*m)
       <|> atom
atom =     do keyw "("
              e <- expr
              keyw ")"
       <|> parse_int
```

Seisundi monaad

```
State s a   -- s on seisundi tüüp; a väärtuse tüüp
```

Seisundi monaad

```
State s a   -- s on seisundi tüüp; a väärtuse tüüp
```

- definitsioon

```
get :: State s s
put :: s -> State s ()
runState :: State a b -> a -> (a, b)
```

Seisundi monaad

```
State s a   -- s on seisundi tüüp; a väärtuse tüüp
```

- definitsioon

```
get :: State s s
put :: s -> State s ()
runState :: State a b -> a -> (a, b)
```

- Tavaliselt tehakse tüübisünonüüm iga konkreetse alamprogrammi jaoks.

```
type PangaSeisund = [(String,Double)]
type PangaArvutus a = State PangaSeisund a
```

...

Seisundi monaad II

- Näide:

```
type PangaSeisund = [(String,Double)]
type PangaArvutus a = State PangaSeisund a

applyIsik :: String -> (Double -> Double) -> PangaArvutus ()
applyIsik nimi f = do
  s <- get
  put (map g s)
    where g (n,s) | n==nimi   = (n, f s)
                  | otherwise = (n, s)

rahaVälja :: String -> Float -> PangaArvutus Bool
rahaVälja nimi summa = do
  s <- get
  case lookup nimi s of
    Nothing -> return False
    Just r  -> do
      applyIsik nimi (\ x -> x-summa)
      return True
```

Seisundi tüübi implementeerimine

```
type State s a = s -> (s, a)

get :: State s s
get s = (s, s)

put :: s -> State s a
put s _ = (s, ())

runState :: State s a -> s -> (s, a)
runState f = f

return :: a -> State s a
return x s = (s, x)

(>>=) :: State s a -> (a -> State s b) -> State s b
(f >>= g) s = g x s'
    where (s',x) = f s
```

Seisundi tüübi implementeerimine

```
type State s a = s -> (s, a)

get :: s -> (s, s)
get x = (x, x)

put :: s -> s -> (s, ())
put s _ = (s, ())

runState :: (s -> (s, a)) -> s -> (s, a)
runState f = f

return :: a -> s -> (s, a)
return x s = (s, x)

(>>=) :: (s -> (s, a)) -> (a -> s -> (s, b)) -> s -> (s, b)
(f >>= g) s = g x s'
    where (s',x) = f s
```

IO modeleerimine seisundimonaadiga

IO modeleerimine seisundimonaadiga

```
type IO a = State 🌍 a
```

IO modeleerimine seisundimonaadiga

```
type IO a = State 🌍 a
```

ehk

```
type IO a = 🌍 -> (🌍, a)
```

Monaadilised funktsioonid standardprelüüdis

- Üldistatud järjestikustamine

```
sequence  :: Monad m => [m a] -> m [a]
sequence = foldr mcons (return [ ])
  where mcons p q = p >>= \ x -> q >>= \ y -> return (x : y)
```

Monaadilised funktsioonid standardprelüüdis

- Üldistatud järjestikustamine

```
sequence  :: Monad m => [m a] -> m [a]
sequence = foldr mcons (return [ ])
  where mcons p q = p >>= \ x -> q >>= \ y -> return (x : y)
```

```
sequence_ :: Monad m => [m a] -> m ()
sequence_ = foldr (>>) (return ())
```

Näited:

Monaadilised funktsioonid standardprelüüdis

- Üldistatud järjestikustamine

```
sequence  :: Monad m => [m a] -> m [a]
sequence = foldr mcons (return [ ])
  where mcons p q = p >>= \ x -> q >>= \ y -> return (x : y)
```

```
sequence_ :: Monad m => [m a] -> m ()
sequence_ = foldr (>>) (return ())
```

Näited:

```
Main> sequence [print 1, print 'a']
1
'a'
[(), ()]
```

Monaadilised funktsioonid standardprelüüdis

- Üldistatud järjestikustamine

```
sequence  :: Monad m => [m a] -> m [a]
sequence = foldr mcons (return [ ])
  where mcons p q = p >>= \ x -> q >>= \ y -> return (x : y)
```

```
sequence_ :: Monad m => [m a] -> m ()
sequence_ = foldr (>>) (return ())
```

Näited:

```
Main> sequence [print 1, print 'a']
1
'a'
[(), ()]
```

```
Main> sequence_ [print 1, print 'a']
1
'a
```

Monaadilised funktsioonid standardprelüüdis

- Üldistatud järjestikustamine

```
sequence  :: Monad m => [m a] -> m [a]
sequence = foldr mcons (return [ ])
  where mcons p q = p >>= \ x -> q >>= \ y -> return (x : y)
```

```
sequence_ :: Monad m => [m a] -> m ()
sequence_ = foldr (>>) (return ())
```

Näited:

```
Main> sequence [print 1, print 'a']
1
'a'
[(), ()]
```

```
Main> sequence_ [print 1, print 'a']
1
'a
```

```
Main> it
()
```

Monaadilised funktsioonid standardprelüüdis

- Üldistatud järjestikustamine

```
sequence  :: Monad m => [m a] -> m [a]
sequence = foldr mcons (return [ ])
  where mcons p q = p >>= \ x -> q >>= \ y -> return (x : y)
```

```
sequence_ :: Monad m => [m a] -> m ()
sequence_ = foldr (>>) (return ())
```

Näited:

```
Main> sequence [print 1, print 'a']
1
'a'
[(), ()]
```

```
Main> sequence_ [print 1, print 'a']
1
'a
```

```
Main> it
()
```

```
Main> sequence [Just 1, Just 2, Just 3]
Just [1,2,3]
```

Monaadilised funktsioonid standardprelüüdis

- Üldistatud järjestikustamine

```
sequence  :: Monad m => [m a] -> m [a]
sequence = foldr mcons (return [ ])
  where mcons p q = p >>= \ x -> q >>= \ y -> return (x : y)
```

```
sequence_ :: Monad m => [m a] -> m ()
sequence_ = foldr (>>) (return ())
```

Näited:

```
Main> sequence [print 1, print 'a']
1
'a'
[(), ()]
```

```
Main> sequence_ [print 1, print 'a']
1
'a
```

```
Main> it
()
```

```
Main> sequence [Just 1, Just 2, Just 3]
Just [1,2,3]
```

```
Main> sequence [Just 1, Nothing, Just 3]
Nothing
```

Monaadilised funktsioonid standardprelüüdis

- Monaadiline `map`

```
mapM :: Monad m => (a -> m b) -> [a] -> m [b]
mapM f as = sequence (map f as)
mapM_ :: Monad m => (a -> m b) -> [a] -> m ()
mapM_ f as = sequence_ (map f as)
```

Monaadilised funktsioonid standardprelüüdis

- Monaadiline `map`

```
mapM :: Monad m => (a -> m b) -> [a] -> m [b]
mapM f as = sequence (map f as)
mapM_ :: Monad m => (a -> m b) -> [a] -> m ()
mapM_ f as = sequence_ (map f as)
```

Näited:

```
Main>mapM print [1..5]
1
2
3
4
5
[(), (), (), (), ()]
```

Monaadilised funktsioonid standardprelüüdis

- Monaadiline `map`

```
mapM :: Monad m => (a -> m b) -> [a] -> m [b]
mapM f as = sequence (map f as)
mapM_ :: Monad m => (a -> m b) -> [a] -> m ()
mapM_ f as = sequence_ (map f as)
```

Näited:

```
Main>mapM print [1..5]
1
2
3
4
5
[(), (), (), (), ()]
```

```
Main>mapM print (Just 6)
6
Just ()
```

Monaadilised funktsioonid standardprelüüdis

- "for"-tsükkel

```
forM :: Monad m => [a] -> (a -> m b) -> m [b]
forM x y = mapM x y
forM_ :: Monad m => [a] -> (a -> m b) -> m ()
forM_ x y = mapM_ x y
```

Monaadilised funktsioonid standardprelüüdis

- "for"-tsükkel

```
forM :: Monad m => [a] -> (a -> m b) -> m [b]
forM x y = mapM x y
forM_ :: Monad m => [a] -> (a -> m b) -> m ()
forM_ x y = mapM_ x y
```

Näited:

```
main = do
  forM_ [1..10] $ \ i -> do
    let n = fact i
    putStrLn $ show i ++ "! = " ++ show n
```

Monaadilised funktsioonid standardprelüüdis

- Tingimuslik täitmine

```
when :: Monad m => Bool -> m () -> m ()
when p s = if p then s else return ()
unless :: Monad m => Bool -> m () -> m ()
unless p s = when (not p) s
```

Monaadilised funktsioonid standardprelüüdis

- Tingimuslik täitmine

```
when :: Monad m => Bool -> m () -> m ()
when p s = if p then s else return ()
unless :: Monad m => Bool -> m () -> m ()
unless p s = when (not p) s
```

Näited:

```
import System.Environment (getArgs)
import System.Directory (doesDirectoryExist)

main = do names <- getArgs
          forM_ names $ \ dir -> do
                b <- doesDirectoryExist dir
                when b $ putStrLn dir
```

Monaadilised funktsioonid standardprelüüdis

- Monaadiline `filter`

```
filterM :: Monad m -> (a -> m Bool) -> [a] -> m [a]
filterM _ [] = return []
filterM p (x : xs) = do
  b  <- p x
  ys <- filterM p xs
  return (if b then x : ys else ys)
```

Monaadilised funktsioonid standardprelüüdis

- Monaadiline `filter`

```
filterM :: Monad m -> (a -> m Bool) -> [a] -> m [a]
filterM _ [] = return []
filterM p (x : xs) = do
  b  <- p x
  ys <- filterM p xs
  return (if b then x : ys else ys)
```

Näited:

```
main = do names <- getArgs
          dirs  <- filterM doesDirectoryExist names
          mapM_ putStrLn dirs
```

Monaadilised funktsioonid standardprelüüdis

- "Liftimine" (1)

```
liftM :: Monad m => (a -> b) -> m a -> m b
liftM f m = do x <- m
               return (f x)
```

Monaadilised funktsioonid standardprelüüdis

- "Liftimine" (1)

```
liftM :: Monad m => (a -> b) -> m a -> m b
liftM f m = do x <- m
               return (f x)
```

Näited:

```
countLines :: FilePath -> IO Int
countLines = liftM (length . lines) . readFile
```

Monaadilised funktsioonid standardprelüüdis

- "Liftimine" (1)

```
liftM :: Monad m => (a -> b) -> m a -> m b
liftM f m = do x <- m
               return (f x)
```

Näited:

```
countLines :: FilePath -> IO Int
countLines = liftM (length . lines) . readFile
```

Enamasti kasutatakse funktsiooni:

```
fmap :: Functor f => (a -> b) -> f a -> f b
```

Monaadilised funktsioonid standardprelüüdis

- "Liftimine" (2)

```
liftM2 :: Monad m => (a -> b -> c) -> m a -> m b -> m c
liftM2 f m1 m2 = do x1 <- m1
                    x2 <- m2
                    return (f x1 x2)
```

Monaadilised funktsioonid standardprelüüdis

- "Liftimine" (2)

```
liftM2 :: Monad m => (a -> b -> c) -> m a -> m b -> m c
liftM2 f m1 m2 = do x1 <- m1
                    x2 <- m2
                    return (f x1 x2)
```

Näited:

```
Main> liftM2 (+) (Just 1) (Just 2)
Just 3
Main> liftM2 (+) (Just 1) Nothing
Nothing
Main> liftM2 (+) [0, 3] [5, 6, 7]
[5,6,7,8,9,10]
```

Monaadilised funktsioonid standardprelüüdis

- "Liftimine" (2)

```
liftM2 :: Monad m => (a -> b -> c) -> m a -> m b -> m c
liftM2 f m1 m2 = do x1 <- m1
                    x2 <- m2
                    return (f x1 x2)
```

Näited:

```
Main> liftM2 (+) (Just 1) (Just 2)
Just 3
Main> liftM2 (+) (Just 1) Nothing
Nothing
Main> liftM2 (+) [0, 3] [5, 6, 7]
[5,6,7,8,9,10]
```

Analoogiliselt on defineeritud **liftM3**, **liftM4** ja **liftM5** .

Monaadilised funktsioonid standardprelüüdis

- Monaadiline applikatsioon

```
ap :: Monad m => m (a -> b) -> m a -> m b
ap = liftM2 id
```

Monaadilised funktsioonid standardprelüüdis

- Monaadiline applikatsioon

```
ap :: Monad m => m (a -> b) -> m a -> m b
ap = liftM2 id
```

Näited:

```
Main> [(+2)] `ap` [1, 2, 3]
[3, 4, 5]
Main> [(+2), (*2)] `ap` [1, 2, 3]
[3,4,5,2,4,6]
```

Monaadilised funktsioonid standardprelüüdis

- Monaadiline applikatsioon

```
ap :: Monad m => m (a -> b) -> m a -> m b
ap = liftM2 id
```

Näited:

```
Main> [(+2)] `ap` [1, 2, 3]
[3, 4, 5]
Main> [(+2), (*2)] `ap` [1, 2, 3]
[3,4,5,2,4,6]
```

```
liftMn f x1 x2 ... xn = return f `ap` x1 `ap` x2 `ap` ... `ap` xn
```

Aplikatsioon on võimalik ka osade mitte-monaadide puhul! (<*>)

```
return f `ap` x1 `ap` x2 `ap` ... `ap` xn = f <$> x1 <*> x2 <*> ... <*> xn
```

Monaaditeisendajad

- Haskell võimaldab väga täpselt spetsifitseerida erinevaid kõrvalefekte. Iga programmi moodul saab tegeleda ainult selle koodi jaoks relevantsega...

Monaaditeisendajad

- Haskell võimaldab väga täpselt spetsifitseerida erinevaid kõrvalefekte. Iga programmi moodul saab tegeleda ainult selle koodi jaoks relevantsega...

- ... aga kunagi tuleb aeg neid ühendada terviklikuks programmiks.

Monaaditeisendajad

- Haskell võimaldab väga täpselt spetsifitseerida erinevaid kõrvalefekte. Iga programmi moodul saab tegeleda ainult selle koodi jaoks relevantsega…
- … aga kunagi tuleb aeg neid ühendada terviklikuks programmiks.
- Monaadide kombineerimiseks on monaaditeisendajad. Selle asemel, et kasutada monaadi `Maybe` kasutame mõnedel juhtudel monaadi `MaybeT m`, kus `m` on mingi teine monaad.

Monaaditeisendajad

- Haskell võimaldab väga täpselt spetsifitseerida erinevaid kõrvalefekte. Iga programmi moodul saab tegeleda ainult selle koodi jaoks relevantsega…
- … aga kunagi tuleb aeg neid ühendada terviklikuks programmiks.
- Monaadide kombineerimiseks on monaaditeisendajad. Selle asemel, et kasutada monaadi `Maybe` kasutame mõnedel juhtudel monaadi `MaybeT m`, kus `m` on mingi teine monaad.
- Haskelli nõrkus — raske on ette näha, mis informatisiooni on vaja läbi programmi kaasas kanda.

QuickCheck on programmide automaatse testimise raamistik:

- Kontrollib, kas programm vastab etteantud omadustele kasutades juhuslikult genereeritud sisendeid.
- Algselt kirjutatud Haskellis, kuid nüüd ka Scalas, Javas jne.
- Teegis on:
  - omaduste kombineerimise kombinaatorid;
  - juhuslike väärtuste generaatorid standardtüüpide jaoks;
  - kombinaatorid oma andmetüüpide juhuslikuks genereerimiseks.

Liides

```
import Test.QuickCheck

quickCheck :: Testable prop => prop -> IO ()
```

- Funktsioon `quickCheck` võtab argumendiks omaduse mida juhuslikel argumentidel kontrollitakse.
- Vaikimisi 100-l juhuslikul väärtusel.
- Kui test ebaõnnestub, siis trükitakse vastunäide.

Näide

`reverse` omadused

$$\forall z.\text{reverse}\ (\text{reverse}\ z) = z$$

```
prop_RevRev          ::  [Int] -> Bool
prop_RevRev xs       =   reverse (reverse xs) == xs
```

$$\forall a.\forall b.\text{reverse}\ (a{+}{+}b) = (\text{reverse}\ b){+}{+}(\text{reverse}\ a)$$

```
prop_RevApp          ::  [Int] -> [Int] -> Bool
prop_RevApp xs ys    =   reverse (xs ++ ys)
                         == reverse ys ++ reverse xs
```

Näide

`reverse` omadused

Katsetame ...

```
Main> quickCheck prop_RevRev
+++ OK, passed 100 tests.

Main> quickCheck prop_RevApp
+++ OK, passed 100 tests.
```

```
pro
pro

pro
prop_RevApp xs ys    =   reverse (xs ++ ys)
                         == reverse ys ++ reverse xs
```

Näide (järg)

`reverse` omadused

$$\forall a.\forall b.\text{reverse } (a{+}{+}b) = (\text{reverse } b){+}{+}(\text{reverse } a)$$

```
prop_RevApp          ::  [Int] -> [Int] -> Bool
prop_RevApp xs ys    =   reverse (xs ++ ys)
                         == reverse ys ++ reverse xs
```

$$\forall a.\forall b.\text{reverse } (a{+}{+}b) = (\text{reverse } a){+}{+}(\text{reverse } b)$$

```
prop_RevWrong        ::  [Int] -> [Int] -> Bool
prop_RevWrong xs ys  =   reverse (xs ++ ys)
                         == reverse xs ++ reverse ys
```

Näide (järg)

`reverse` omadused

$$\forall a.\forall b.\text{reverse}\ (a{+}{+}b) = (\text{reverse}\ b){+}{+}(\text{reverse}\ a)$$

Katsetame ...

```
Main> quickCheck prop_RevWrong
*** Failed! Falsifiable (after 3 tests and 2 shrinks):
[0]
[1]
```

```
prop_RevWrong          ::  [Int] -> [Int] -> Bool
prop_RevWrong xs ys    =   reverse (xs ++ ys)
                           == reverse xs ++ reverse ys
```

Omadused

```
class Testable prop where
      property :: prop -> Property

instance Testable Bool

instance (Arbitrary a, Show a, Testable prop) =>
         Testable (a -> prop)
```

- Omadused on avaldised, mille tüüb on klassist `Testable`.
- Argumendid peavad olema monomorfset tüüpi.
  - Vaja argumentide genereerimise jaoks.
- Nimed tava järgi prefiksiga `prop_`

Omadused

Näide: sorteerimine pistemeetodil

```
isort :: Ord a => [a] -> [a]
isort = foldr insert []

insert :: Ord a => a -> [a] -> [a]
insert x []                 = [x]
insert x (y:ys) | x <= y    = x : y : ys
                | otherwise = y : insert x ys
```

Omadused

Omadus 1: tulemus peab olema järjestatud

```
prop_sortOrder :: [Int] -> Bool
prop_sortOrder xs = ordered (isort xs)

ordered :: Ord a => [a] -> Bool
ordered (x:y:ys)  = x <= y && ordered (y:ys)
ordered ys        = True
```

Omadused

Omadus 2: operatsioon ei lisa ega kustuta elemente

```
prop_sortElems :: [Int] -> Bool
prop_sortElems xs = sameElems xs (isort xs)

sameElems :: Eq a => [a] -> [a] -> Bool
sameElems xs ys   = null (xs \\ ys) && null (ys \\ xs)
```

Omadused

Testandmete inspekteerimine

```
collect :: (Show a, Testable prop) => a -> prop -> Property
```

- `collect` kogub statistikat testjuhtude kohta.
- Info kuvatakse testi läbimisel.

Omadused

Minut testi on mittetühjad?

```
Main> let p = prop_sortOrder
Main> quickCheck (\ xs -> collect (null xs) (p xs))
+++ OK, passed 100 tests.
93% False
 7% True
```

Omadused

Kui pikad on listid?

```
Main> let l20 xs = length xs `div` 20
Main> quickCheck (\xs -> collect (l20 xs) (p xs))
+++ OK, passed 100 tests:
53% 0
22% 1
14% 2
 7% 3
 4% 4
```

Omadused

Mis täpselt olid argumendid?

```
Main> quickCheck (\ xs -> collect xs (p xs))
+++ OK, passed 100 tests:
 8% []
 1% [97723,95805,-104521,45943,-73844,6249,64936]
...
```

Omadused

piste omadused: säilitab sorteerituse (ver. 1)

```
prop_insertOrder1      :: Int -> [Int] -> Bool
prop_insertOrder1 x xs =  ordered xs `implies`
                               ordered (insert x xs)

implies     :: Bool -> Bool -> Bool
implies x y =  not x || y
```

Omadused

piste omadused: säilitab sorteerituse (ver. 1)

Problem!

```
Main> let p = prop_insertOrder1
Main> quickCheck(\x xs -> collect(ordered xs)(p x xs))
+++ OK, passed 100 tests:
87% False
13% True
```

Omadused

Implikatsioon

```
(==>) :: Testable prop => Bool -> prop -> Property

instance Testable Property
```

- Kombinaator `(==>)` ignoreerib sisendit, kui eeldus pole täidetud, ning genereerib uue väärtused.
- Vaikimisi 500 korda.

Omadused

piste omadused: säilitab sorteerituse (ver. 2)

```
prop_insertOrder2      :: Int -> [Int] -> Property
prop_insertOrder2 x xs =  ordered xs ==>
                               ordered (insert x xs)
```

Omadused

juba parem . . .

```
Main> let p = prop_insertOrder2
Main> quickCheck(\x xs -> collect(ordered xs)(p x xs))
*** Gave up! Passed only 82 tests (100% True).
```

Omadused

Univresaalne kvantifitseerimine

```
forAll :: (Show a, Testable prop) =>
               Gen a -> (a -> prop) -> property
```

- Kombinaator `forAll` saab argumendiks generaatori, millega väärtusi luuakse.
- Saame luua väärtusi, mis vastavad valitud omadustele.

Omadused

piste omadused: säilitab sorteerituse (ver. 3)

```
prop_insertOrder3   :: Int -> Property
prop_insertOrder3 x =  forAll orderedList (\ xs ->
                                  ordered (insert x xs))
```

Omadused

pi

pro
pro

Töötab!!

```
Main> quickCheck (forAll orderedList ordered)
+++ OK, passed 100 tests.
Main> quickCheck prop_insertOrder3
+++ OK, passed 100 tests.
```

Generaatorid

Generaatorid

```
newtype Gen a = ...

instance Monad Gen

instance Functor Gen

instance (Testable prop) => Testable (Gen prop)
```

- Generaatorid on abstraktse tüübiga `Gen`.
- `Gen` on monaad, millel on juurdepääs juhuarvudele.

Generaatorid

Trükib mõned genereeritud väärtused

```
sample :: Show a => Gen a -> IO ()
```

Generaatorite kombineerimine

```
choose      :: Random a => (a, a) -> Gen a
elements    :: [a] -> Gen a
oneof       :: [Gen a] -> Gen a
frequency   :: [(Int, Gen a)] -> Gen a
sized       :: (Int -> Gen a) -> Gen a
vectorOf    :: Int -> Gen a -> Gen [a]
```

Generaatorid

Vaike-generaatorid

```
class Arbitrary a where
  arbitrary :: Gen a
  shrink    :: a -> [a]

  shrink _  =  []
```

- Vaike-generaatoriga tüübid kuuluvad klassi `Arbitrary`.
- Lisaks on klassis meetod `shrink`, mis vähendab väärtust.
  - `shrink` tagastab struktuurselt väiksemate väärtuste listi;
  - kui test ebaõnnestub, proovitakse `shrink` abil argimente vähendada.

Generaatorid

Lihtsad generaatorid

```
instance Arbitrary Bool where
  arbitrary  = choose (False,True)

instance (Arbitrary a, Arbitrary b) => Arbitrary (a,b) where
  arbitrary  = do
    x <- arbitrary
    y <- arbitrary
    return (x,y)


data Color = Red | Blue | Green

instance Arbitrary Color where
  arbitrary  = elements [Red, Blue, Green]
```

Generaatorid

Lihtsad generaatorid

```
instance Arbitrary a => Arbitrary (Maybe a) where
  arbitrary = oneof [ return Nothing
                    , liftM Just arbitrary]
```

Generaatorid

Lihtsad generaatorid

```
instance Arbitrary a => Arbitrary (Maybe a) where
  arbitrary = oneof [ return Nothing
                    , liftM Just arbitrary]
```

Probleem!

Pool väärtustest on `Nothing`!!

Generaatorid

Lihtsad generaatorid

```
instance Arbitrary a => Arbitrary (Maybe a) where
  arbitrary = oneof [ return Nothing
                    , liftM Just arbitrary]
```

Parem versioon

```
instance Arbitrary a => Arbitrary (Maybe a) where
  arbitrary = frequency [ (1, return Nothing)
                        , (3, liftM Just arbitrary)]
```

Generaatorid

Täisarvude genereerimine (ver. 1)

```
instance Arbitrary Int where
  arbitrary = choose (-20, 20)
```

Generaatorid

Täisarvude genereerimine (ver. 1)

```
instance Arbitrary Int where
  arbitrary = choose (-20, 20)
```

Täisarvude genereerimine (ver. 2)

```
instance Arbitrary Int where
  arbitrary = sized (\ n -> choose (-n,n))
```

Generaatorid

Rekursiivsete andmetüüpide genereerimine (ver. 1)

```
data Tree a  =  Leaf a
             |  Node (Tree a) (Tree a)

instance Arbitrary a => Arbitrary (Tree a) where
  arbitrary = frequency [(1, liftM  Leaf arbitrary)
                        ,(2, liftM2 Node arbitrary arbitrary)]
```

Generaatorid

Rekursiivsete andmetüüpide genereerimine (ver. 1)

```
data Tree a  =  Leaf a

instanc
  arbitrary = frequency [(1, liftM Leaf arbitrary)
                        ,(2, liftM2 Node arbitrary arbitrary)]
```

Probleem!

Termineerimine pole kindlustatud!

Generaatorid

Rekursiivsete andmetüüpide genereerimine (ver. 2)

```
instance Arbitrary a => Arbitrary (Tree a) where
  arbitrary = sized arbitraryTree

arbitraryTree   :: Arbitrary a => Int -> Gen (Tree a)
arbitraryTree 0 =  liftM Leaf arbitrary
arbitraryTree n =  frequency [ (1, liftM  Leaf arbitrary)
                             , (4, liftM2 Node t t)]
  where t = arbitraryTree (n `div` 2)
```

NB!

- Teine võrdus võib ka genereerida `Leaf`-e.
- Muidu genereeriks ainult tasakaalus puid.

Generaatorid

Eeldefineeritud erilised generaatorid

```
newtype OrderedList a  = Ordered [a]
instance (Ord a, Arbitrary a) => Arbitrary (OrderedList a)

newtype NonEmptyList a = NonEmpty [a]
instance Arbitrary a => Arbitrary (NonEmptyList a)

newtype Positive a     = Positive a
instance (Num a, Ord a, Arbitrary a) => Arbitrary (Positive a)

newtype NonZero a      = NonZero a
newtype NonNegative a  = NonNegative a
```

Funktsioonide genereerimine

```
class CoArbitrary a where
  coarbitrary :: a -> Gen b -> Gen b

instance (CoArbitrary a, Arbitrary b) => Arbitrary (a -> b)
```

Example

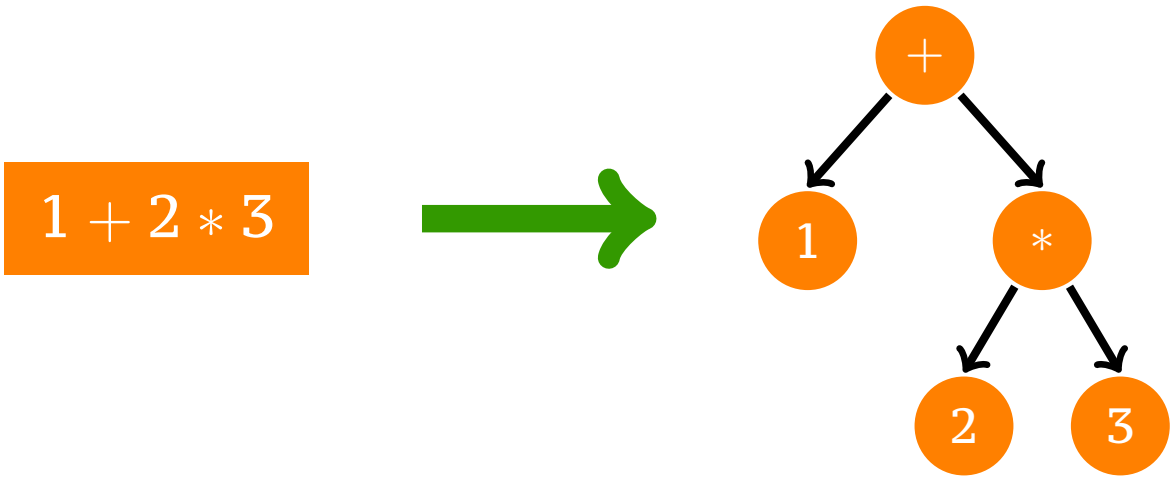
```
variant :: Integral n => n -> Gen a -> Gen a

instance CoArbitrary a => CoArbitrary [a] where
  coarbitrary []     = variant 0
  coarbitrary (x:xs) = variant 1 . coarbitrary (x,xs)
```

- Soovitatav kasutada `variant` et segada juhuslikku genereerimist.
- Erineva esimesed argumendi puhul püütakse tagastatakse erinev väärtus.

Funktsionaalne parser

Parseri ülesanne on muuta sõne abstraktseks süntaksipuuks



Parseri monaad

Monaadi jaoks on vaja **data** või **newtype**

```
newtype Parser a = P {runP :: (String -> [(a,String)])}
```

Parseri monaad

Monaadi jaoks on vaja **data** või **newtype**

```
newtype Parser a = P {runP :: (String -> [(a,String)])}
```

Parser monad

```
instance Monad Parser where
   return a = P $ \str -> [(a, str)]
   p >>= f  = P $ \str -> concat [runP (f a) cs |
                                  (a,cs) <- runP p str]

instance MonadPlus Parser where
   mzero     = P $ \str -> []
   mplus p q = P $ \str -> runP p str ++ runP q str
```

Parseri monaad

Abifunktsioonid

```
item :: Parser Char
item = P $ \str -> [(head str, tail str) | not (null str)]

first    ::  Parser a -> Parser a
first p  =   P $ \str ->  case runP p str of
                            []      ->  []
                            (x:xs)  ->  [x]
```

Lihtsad kombinaatorid

```
sat       ::  (Char -> Bool) -> Parser Char
sat p     =   do c <- item
                 if p c then
                   return c
                 else
                   mzero

char      ::  Char -> Parser Char
char c    =   sat (c ==)

(<|>)     ::  Parser a -> Parser a -> Parser a
p <|> q   =   first (p `mplus` q)
```

Näide

```
parens  ::  Parser Tree
parens  =   do   char '('
                 t1  <- parens
                 char ')'
                 t2  <- parens
                 return (Bin t1 t2)
            <|>  return Nil
```

Lihtsad kombinaatorid

Iteratsion

```
many     ::  Parser a -> Parser [a]
many p   =   many1 p <|> return []

many1    ::  Parser a -> Parser [a]
many1 p  =   do a <- p
                as <- many p
                return (a:as)
```

Lihtsad kombinaatorid

Võtmesõnad

```
string        :: String -> Parser String
string ""     =  return ""
string (c:cs) =  do char c
                    string cs
                    return (c:cs)
```

Identifikaatorid

```
identifier :: Parser String
identifier =  do c   <-  lower
                 cs  <-  many alphanum
                 return (c:cs)
```

Lihtsad kombinaatorid

Naturaalarvud

```
digit   :: Parser Int
digit   =  do c <- sat isDigit
              return (ord c - ord '0')

natural :: Parser Int
natural =  do ds <- many1 digit
              return (foldl1 (\a b -> 10*a + b) ds)
```

Täisarvud

```
integer :: Parser Int
integer =      do {char '-'; n <- natural; return (-n)}
          <|> natural
```

Lihtsad kombinaatorid

Ujukomaarvud

```
fraction :: Parser Double
fraction =  do char '.'
               ds <- many1 digit
               return (foldr op 0 ds)
   where d `op` x = (x + fromIntegral d)/10

floating :: Parser Double
floating =  do i  <-  integer
               f  <-  fraction <|> return 0
               return (fromIntegral i + f)
```

Lihtsad kombinaatorid

tühikud

```
space   :: Parser String
space   = many (sat isSpace)

token   :: Parser a -> Parser a
token p = do a <- p
             space
             return a

keyw cs = token (string cs)
ident   = token identifier
nat     = token natural
int     = token integer
float   = token floating
```

Lihtsad kombinaatorid

Sulud

```
pack :: Parser a -> Parser b -> Parser c -> Parser b
pack s1 p s2 = do s1
                  x <- p
                  s2
                  return x

paren p      = pack (keyw "(") p (keyw ")")
brack p      = pack (keyw "[") p (keyw "]")
block p      = pack (keyw "begin") p (keyw "end")
```

Lihtsad kombinaatorid

Järjendid

```
sepby           ::  Parser a -> Parser b -> Parser [a]
p `sepby` sep   =   (p `sepby1` sep) <|> return []

sepby1          ::  Parser a -> Parser b -> Parser [a]
p `sepby1` sep  =   do  a   <-  p
                        as  <-  many (sep >> p)
                        return (a:as)

commaList p     =   sepby p (keyw ",")
semicList p     =   sepby p (keyw ";")
```

Lihtsad kombinaatorid

Järjendid

```
chainl  :: Parser a -> Parser (a->a->a) -> Parser a
chainl p s = do
        x   <-  p
        ys  <-  many (do {op <- s; y <- p; return (op,y)})
        return (foldl (\a (op,y) -> a `op` y) x ys)

chainr  :: Parser a -> Parser (a->a->a) -> Parser a
chainr p s = do
        ys  <-  many (do {y <- p; op <- s; return (y,op)})
        x   <-  p
        return (foldr (\(y,op) b -> y `op` b) x ys)
```

Terve sisendi parsimine

```
parse :: Parser a -> String -> a
parse p cs  =  case  runP (first (space >> p)) cs of
                     [(x,"")]  ->  x
                     _         ->  error "Parse error"
```

Näide: aritmeetilised avaldised

Aritmeetilised avaldised

Grammatika

```
expr  =  int          |  expr  +  expr  |  expr  -  expr
      |  expr * expr  |  expr  /  expr  |  expr  ^  expr
      |  (expr)
```

Abstraktne süntaksipuu

```
data Expr  =  Num Int
           |  Expr  :+:  Expr
           |  Expr  :-:  Expr
           |  Expr  :*:  Expr
           |  Expr  :/:  Expr
           |  Expr  :^:  Expr
```

Aritmeetilised avaldised

Parseri ver. 0

```
expr0 =       do  {  e0  <-  expr0; keyw "+";
                     e1  <-  expr0; return(e0 :+: e1)}
        <|>   do  {  e0  <-  expr0; keyw "-";
                     e1  <-  expr0; return(e0 :-: e1)}
        <|>               ...
        <|>   do  {  e0  <-  expr0; keyw "^";
                     e1  <-  expr0; return(e0 :^: e1)}
        <|>   do  {i <- int; return (Num i)}
        <|>   paren expr0
```

Aritmeetilised avaldised

Parseri ver. 0

```
expr0 =       do  {  e0  <-  expr0; keyw "+";
                     e1  <-  expr0; return(e0 :+: e1)}
                     e1  <-  expr0; return(e0 :^: e1)}
          <|>  do  {i <- int; return (Num i)}
          <|>  paren expr0
```

NB!

Ei tööta, kuna parser on vasak-rekursiivne!!

Aritmeetilised avaldised

Parseri ver. 1

```
expr1  =  do   a   <- atom1
               op  <- oper1
               e   <- expr1
               return (a `op` e)
          <|>  atom1

oper1  =       (keyw  "+"  >>  return (:+:))
          <|>  (keyw  "-"  >>  return (:-:))
          <|>  (keyw  "*"  >>  return (:*:))
          <|>  (keyw  "/"  >>  return (:/:))
          <|>  (keyw  "^"  >>  return (:^:))

atom1  =  do   {i <- int; return (Num i)}
          <|>  paren expr1
```

Aritmeetilised avaldised

Parseri ver. 1

```
expr1  =  do   a   <- atom1
               op  <- oper1
               e   <- expr1
               return (a `op` e)

oper1  =  do   {reservedOp "*"; return (:*:)}
          <|>  (keyw  "*"  >>  return (:*:))
          <|>  (keyw  "/"  >>  return (:/:))
          <|>  (keyw  "^"  >>  return (:^:))

atom1  =  do   {i <- int; return (Num i)}
          <|>  paren expr1
```

NB!

”Töötab” kuid ei arvesta
prioriteete ja assotsiatiivsust!

Aritmeetilised avaldised

Parseri ver. 2

```
expr2  ::  Parser Expr
expr2  =   chainl term2  (    (keyw "+"  >>  return (:+:))
                         <|>  (keyw "-"  >>  return (:-:)))

term2  ::  Parser Expr
term2  =   chainl fact2  (    (keyw "*"  >>  return (:*:))
                         <|>  (keyw "/"  >>  return (:/:)))

fact2  ::  Parser Expr
fact2  =   chainr atom2  (keyw "^" >> return (:^:))

atom2  ::  Parser Expr
atom2  =   do   {i <- int; return (Num i)}
           <|>  paren expr2
```

Arithmetic expressions

Evaluaator

```
expr3  ::  Parser Int
expr3  =   chainl term3  (    (keyw "+"  >>  return (+))
                         <|>  (keyw "-"  >>  return (-)))

term3  ::  Parser Int
term3  =   chainl fact3  (    (keyw "*"  >>  return (*))
                         <|>  (keyw "/"  >>  return div))

fact3  ::  Parser Int
fact3  =   chainr atom3 (keyw "^" >> return (^))

atom3  ::  Parser Int
atom3  =   int <|> paren expr3
```

Parsec

```
data ParsecT s u m a
```

on parser milles

- sisendvoo tüüp `s`,
- kasitaja defineeritud seisund `u`,
- kasutaja monaad `m`, ja
- tagastustüüp `a`.

Type

```
data ParsecT s u m a
  = ParsecT {unParser :: forall b .
                 State s u
              -> (a -> State s u -> ParseError -> m b) -- consumed ok
              -> (ParseError -> m b)                   -- consumed err
              -> (a -> State s u -> ParseError -> m b) -- empty ok
              -> (ParseError -> m b)                   -- empty err
              -> m b
             }
```

Parsec

- Text.Parsec

```
runParser :: Stream s Identity t => Parsec s u a -> u -> SourceName -> s
                                                 -> Either ParseError a

(<|>) :: ParsecT s u m a -> ParsecT s u m a -> ParsecT s u m a
try   :: ParsecT s u m a -> ParsecT s u m a
many  :: Stream s m t => ParsecT s u m a -> ParsecT s u m [a]
many1 :: Stream s m t => ParsecT s u m a -> ParsecT s u m [a]
sepBy :: Stream s m t => ParsecT s u m a -> ParsecT s u m sep -> ParsecT s u m [a]
```

- Text.Parsec.Char

```
letter  :: Stream s m Char => ParsecT s u m Char
string  :: Stream s m Char => String -> ParsecT s u m String
anyChar :: Stream s m Char => ParsecT s u m Char
oneOf   :: Stream s m Char => [Char] -> ParsecT s u m Char
satisfy :: Stream s m Char => (Char -> Bool) -> ParsecT s u m Char
```

Monad Transformer

```
runParserT :: Stream s m t => ParsecT s u m a -> u -> SourceName
              -> s -> m (Either ParseError a)

class MonadTrans (t :: (* -> *) -> * -> *) where
  lift :: Monad m => m a -> t m a

instance MonadTrans (ParsecT s u)
```

Monad Transformer

```
runParserT :: Stream s m t => ParsecT s u m a -> u -> SourceName
              -> s -> m (Either ParseError a)

class MonadTrans (t :: (* -> *) -> * -> *) where
  lift :: Monad m => m a -> t m a

instance MonadTrans (ParsecT s u)
```

Example

```
okP = do
  string "ok"
  lift $ putStrLn "read ok!"

parseOk = runParserT okP () "test source" "ok"
```

Monad Transformer

```
runParserT :: Stream s m t => ParsecT s u m a -> u -> SourceName
              -> s -> m (Either ParseError a)

class MonadTrans (t :: (* -> *) -> * -> *) where
  lift :: Monad m => m a -> t m a

instance MonadTrans (ParsecT s u)
```

Example

```
okP = do
  string "ok"
  lift $ putStrLn "read ok!"

parseOk = runParserT okP () "test source" "ok"
```

```
*Hw4_2> parseOk
read ok!
Right ()
```

State

```
getState    :: Monad m => ParsecT s u m u
putState    :: Monad m => u -> ParsecT s u m ()
modifyState :: Monad m => (u -> u) -> ParsecT s u m ()
```

State

```
getState    :: Monad m => ParsecT s u m u
putState    :: Monad m => u -> ParsecT s u m ()
modifyState :: Monad m => (u -> u) -> ParsecT s u m ()
```

Example

```
stateP :: ParsecT [Char] Int Identity Int
stateP = do
  x <- getState
  putState 10
  return x

parseSt = runParser stateP 5 "test source" "ok"
```

State

```
getState    :: Monad m => ParsecT s u m u
putState    :: Monad m => u -> ParsecT s u m ()
modifyState :: Monad m => (u -> u) -> ParsecT s u m ()
```

Example

```
stateP :: ParsecT [Char] Int Identity Int
stateP = do
  x <- getState
  putState 10
  return x

parseSt = runParser stateP 5 "test source" "ok"
```

```
*Hw4_2> parseSt
Right 5
```