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"
```

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.

- 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.
- 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
```

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`

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

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]
```

- 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)]
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
```

- 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

```haskell
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

```
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)
```

```
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)
```

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
```

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
```

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)
```

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)
```

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)
```

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
```

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…
- … 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

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

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

p
p

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

Kui pikad on listid?

Mis täpselt olid argumendid?

```
Main> quickCheck (\ xs -> collect xs (p xs))
+++ OK, passed 100 tests:
 8% []
 1% [97723,95805,-104521,45943,-73844,6249,64936]
...
 4% 4
```

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

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

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]
```

Probleem!

Pool väärtustest on `Nothing`!!

Paren

```
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)
```

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
```

Probleem!

Termineerimine pole kindlustatud!

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

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.