Reduktsiooni järjekorrad

Aplikatiivne järjekord — väärtustatakse seest välja

$$\begin{array}{rcl}\texttt{double}(\underline{2+3}) & \Longrightarrow & \underline{\texttt{double}\ 5} \\ & \Longrightarrow & \underline{2\times 5} \\ & \Longrightarrow & 10\end{array}$$

Normaaljärjekord — väärtustatakse väljast sisse

$$\begin{array}{rcl}\underline{\texttt{double}(2+3)} & \Longrightarrow & 2\times(\underline{2+3}) \\ & \Longrightarrow & \underline{2\times 5} \\ & \Longrightarrow & 10\end{array}$$

Church-Rosser'i teoreem: Avaldise normaalkuju ei sõltu reduktsioonijadast.

Reduktsiooni järjekorrad

Normaalkuju leidumine sõltub reduktsioonijärjekorrast!

$$\begin{array}{lcl}\texttt{const}\ x\ y & = & x \\ \texttt{loop} & = & \texttt{loop}\end{array}$$

$$\begin{array}{lcl}\texttt{const}\ 5\ \underline{\texttt{loop}} & \Longrightarrow & \texttt{const}\ 5\ \underline{\texttt{loop}} \\ & \Longrightarrow & \texttt{const}\ 5\ \underline{\texttt{loop}} \\ & \Longrightarrow & \ldots\end{array}$$

$$\underline{\texttt{const}\ 5\ \texttt{loop}} \Longrightarrow 5$$

Normaliseerimisteoreem: Kui avaldisel leidub normaalkuju, siis normaaljärjekorras reduktsioonijada on lõplik.

Reduktsiooni järjekorrad

Normaaljärjekord võib olla ebaefektiivne!

$$
\begin{array}{ll}
\mathtt{square}(\underline{2+3}) & \underline{\mathtt{square}(2+3)} \\
\quad \Longrightarrow \underline{\mathtt{square}\ 5} & \quad \Longrightarrow (\underline{2+3}) \times (2+3) \\
\quad \Longrightarrow \underline{5 \times 5} & \quad \Longrightarrow 5 \times (\underline{2+3}) \\
\quad \Longrightarrow 25 & \quad \Longrightarrow 5 \times 5 \\
 & \quad \Longrightarrow 25
\end{array}
$$

Laisk väärtustamine = normaaljärjekord + graafireduktsioon

$$
\begin{array}{llll}
\underline{\mathtt{square}(2+3)} & \Longrightarrow & x \times x & \mathtt{where}\ x = \underline{2+3} \\
 & \Longrightarrow & \underline{x \times x} & \mathtt{where}\ x = 5 \\
 & \Longrightarrow & 25 &
\end{array}
$$

Laisk väärtustamine

- Lõpmatud listid

```
take 5 [1..]  ===> [1,2,3,4,5]

multiples :: [[Int]]
multiples = [ [ m*n | m <- [1..]] | n <- [1..]]

multiples ===> [[1, 2, 3, 4, 5, ...],
                [2, 4, 6, 8,10, ...],
                [3, 6, 9,12,14, ...],
                .....................]

take 4 (multiples !! 3) ===> [4,8,12,16]
```

Laisk väärtustamine

- Funktsioon `iterate` (eeldefineeritud)

$$iterate\, f\, x = [x,\ f\, x,\ f^2 x,\ f^3 x,\ \ldots]$$

- Definitsioon

```
iterate     :: (a -> a) -> a -> [a]
iterate f x = x : iterate f (f x)
```

- Näited

```
powertables :: [[Int]]
powertables = [iterate (*n) 1 | n <- [2..]]

digits :: Int -> [Int]
digits = reverse . map (`mod` 10) . takeWhile (/= 0)
                                  . iterate (`div` 10)
```


Laisk väärtustamine

- Ruutjuur Newton–Raphson'i meetodil

$$a_{i+1} = \frac{a_i + \frac{n}{a_i}}{2}$$

```
next n x = (x + n/x)/2.0

within eps (x1:x2:xs)
           | abs(x1-x2) <= eps  = x2
           | otherwise          = within eps (x2:xs)

sqrt1 n = within eps (iterate (next n) a0)
          where eps = 0.00001
                a0  = 1.0
```

Laisk väärtustamine

- Erasthostenese sõel
  1. Kirjuta üles kõik positiivsed täisarvud alates arvust 2;
  2. Tähista jada esimene element *p* kui algarv;
  3. Kustuta jadast kõik arvu *p* kordsed;
  4. Mine tagasi sammule 2.

```
primes       = map head (iterate sieve [2..])
sieve (p:xs) = [x | x<-xs, x `mod` p /= 0]
```


Laisk väärtustamine

- Kaheksa lippu

```
queens 0     = [[]]
queens (m+1) = [p++[n] | p<-queens m, n<-[1..8]
                                    , safe p n]

safe p n = and [not(check (i,j) (m,n))
                | (i,j) <- zip [1..] p]
           where m = 1 + length p

check (i,j) (m,n) = j==n || (i+j==m+n)
                         || (i-j==m-n)
```

Laisk väärtustamine

- Fibonacci jada

$$1, 1, 2, 3, 5, 8, 13, 21, 34, 55, \ldots$$

- n-inda fibonacci arvu leidmine

```
fib :: Integer -> Integer
fib 0 = 1
fib 1 = 1
fib n = fib (n-1) + fib (n-2)
```


Laisk väärtustamine

Toodud definitsioon on väga ebaefektiivne (eksponentsiaalse keerukusega)

```
fib 8 ===> fib 7 + fib 6
      ===> (fib 6 + fib 5) + (fib 5 + fib 4)
      ===> ((fib 5 + fib 4) + (fib 4 + fib 3))
           + ((fib 4 + fib 3) + (fib 3 + fib 2))
      ===> (((fib 4 + fib 3) + (fib 3 + fib 2))
            + ((fib 3 + fib 2) + (fib 2 + fib 1)))
           + (((fib 3 + fib 2) + (fib 2 + fib 1))
             + ((fib 2 + fib 1) + (fib 1 + fib 0)))
      ...
```

Laisk väärtustamine

- Fibonacci jada

	1	1	2	3	5	8	13	...
	1	2	3	5	8	13	21	...
+	2	3	5	8	13	21	34	...

```
fibs :: [Integer]
fibs = 1 : 1 : zipWith (+) fibs (tail fibs)
```

- Toodud definitsioon on palju efektiivsem (lineaarse keerukusega) ja kasutab palju vähem ressursse!

Laisk väärtustamine

- Klient-server näide:

```
type Request  = Integer
type Response = Integer

client :: [Response] -> [Request]
client ys = 1 : ys

server :: [Request]  -> [Response]
server xs = map (+1) xs

reqs  = client resps
resps = server reqs
```

Laisk väärtustamine

- Klient, mis soovib testida serveri “õigsust”:

```
reqs  = client resps
resps = server reqs

client (y:ys) = if ok y then 1 : (y:ys)
                else error "faulty server"
server xs     = map (+1) xs
```

- Ei väljasta midagi !!

```
reqs ==> client resps
     ==> client (server reqs)
     ==> client (server (client resps))
     ==> client (server (client (server reqs)))
     ...
```


Laisk väärtustamine

- Fibonacci veelkord:

```
fibsFn :: () -> [Integer]
fibsFn () = 1 : 1 : zipWith (+) (fibsFn ())
                                (tail (fibsFn ()))
```

- On jälle eksponentsiaalse keerukusega!!

```
fibsFn ()
  ===> 1 : 1 : add (fibsFn ()) (tail (fibsFn ()))
  ===> 1 : 1 : add (1 : 1 : add (fibsFn ())
                                (tail (fibsFn ())))
                   (1 : add (fibsFn ())
                            (tail (fibsFn ())))
  ===> ...
```

Laisk väärtustamine

- Vajame nn. memoiseerivat funktsiooni, mis salvestab varasemad väljakutsed tabelisse:

```
memo :: (a->b) -> (a->b)
-- memo f x = f x
```

- Funktsioon `memo1` ehitab täpselt ühe elemendilise tabeli
- Memoiseeriv fibonacci:

```
mfibsFn :: () -> [Integer]
mfibsFn x = let mfibs = memo1 mfibsFn
            in 1 : 1 : zipWith (+) (mfibs ())
                                   (tail (mfibs ()))
```


Laisk väärtustamine

- Liita listi kõigile elementidele listi minimaalne element

```
incmin xs = map  (minv+) xs
          where minv = minimum xs
```

- Listi ühekordse läbimisega versioon

```
incmin [] = []
incmin xs = newlist
  where (minv,newlist) = onepass xs
        onepass [a]    = (a, [a+minv])
        onepass (a:as) = (a `min` b, (a+minv):bs)
                         where (b,bs) = onepass as
```