

Rekursioon lambda-arvutuses

- Rekursiivne protseduur on defineeritud “iseenda kaudu”:

```
(define fact
  (lambda (n)
    (if (zero? n)
        1
        (* n (fact (- n 1))))))
```

- Selleks, et mingit objekti (protseduuri) kasutada, tuleb anda talle nimi
- ..., aga lambda-arvutuses on kõik protseduurid “anonüümsed”

Rekursioon lambda-arvutuses

- Lambda-arvutuses saab “nimesid anda” ainult lambda-abstraktsiooni kasutades:

```
((lambda (id) (id 3))  
  (lambda (x) x))
```

- ..., ehk kasutades let-avaldisi “süntaktilise suhkruna”:

```
(let ((var1 exp1)  
      ...  
      (varn expn))  
  body) ⇒ ((lambda (var1 ... varn) body)  
            exp1 ... expn)
```

Rekursioon lambda-arvutuses

- Selline “nimede andmine” töötab hästi mitte-rekursiivsete protseduuride korral, aga kuidas “anda nimesid” rekursiivsetele protseduuridele?

```

(lambda (g)
  (lambda (n)
    (if (zero? n)
        1
        (* n (g (- n 1))))))
( ??? )

```

- Oleks vaja nn. püsipunktioperaatorit *fix*, mille korral $(fix E) = (E (fix E))$

Rekursioon lambda-arvutuses

- Y-kombinaator:

```
(lambda (f)
  ((lambda (x) (f (x x)))
   (lambda (x) (f (x x)))))
```

- Y-kombinaator on püsipunktioperaator!!

```
((lambda (f)
  ((lambda (x) (f (x x)))
   (lambda (x) (f (x x)))))
E)
```

Rekursioon lambda-arvutuses

- Y-kombinaator:

```
(lambda (f)
  ((lambda (x) (f (x x)))
   (lambda (x) (f (x x)))))
```

- Y-kombinaator on püsipunktioperaator!!

```
((lambda (f)
  ((lambda (x) (f (x x)))
   (lambda (x) (f (x x)))))
E)
```

Rekursioon lambda-arvutuses

- Y-kombinaator:

```
(lambda (f)
  ((lambda (x) (f (x x)))
   (lambda (x) (f (x x)))))
```

- Y-kombinaator on püsipunktioperaator!!

```
((lambda (x) (E (x x)))
 (lambda (x) (E (x x))))
```

Rekursioon lambda-arvutuses

- Y-kombinaator:

```
(lambda (f)
  ((lambda (x) (f (x x)))
   (lambda (x) (f (x x)))))
```

- Y-kombinaator on püsipunktioperaator!!

```
((lambda (x) (E (x x)))
 (lambda (x) (E (x x))))
```

Rekursioon lambda-arvutuses

- Y-kombinaator:

```
(lambda (f)
  ((lambda (x) (f (x x)))
   (lambda (x) (f (x x)))))
```

- Y-kombinaator on püsipunktioperaator!!

```
(E ((lambda (x) (E (x x)))
   (lambda (x) (E (x x)))))
```

Rekursioon lambda-arvutuses

- Y-kombinaator:

```
(lambda (f)
  ((lambda (x) (f (x x)))
   (lambda (x) (f (x x)))))
```

- Y-kombinaator on püsipunktioperaator!!

```
(E ((lambda (x) (E (x x)))
   (lambda (x) (E (x x)))))
```

Rekursioon lambda-arvutuses

- Y-kombinaator:

```
(lambda (f)
  ((lambda (x) (f (x x)))
   (lambda (x) (f (x x)))))
```

- Y-kombinaator on püsipunktioperaator!!

```
(E ((lambda (f)
      ((lambda (x) (f (x x)))
       (lambda (x) (f (x x)))))
  E))
```

Püsipunktioperaatorid Scheme's

- Toodud Y-kombinaator “töötab” ainult normaaljärjekorra korral ja pole seetõttu Scheme's kasutatav
- Kasutades η -ekspansiooni, saame Scheme's defineerida aplikatiivjärjekorra püsipunktioperaatori:

```
(define Y1
  (lambda (f)
    ((lambda (x) (f (lambda (y) ((x x) y))))
     (lambda (x) (f (lambda (y) ((x x) y)))))))
```

Püsipunktioperaatorid Scheme's

- Püsipunktioperaatori abil defineeritud faktoriaal:

```
(define fact
  (let ((f (lambda (g)
            (lambda (n)
              (if (zero? n)
                  1
                  (* n (g (- n 1))))))))
    (Y1 f)))
```

```
> (fact 10)
3628800
```

Laisk-väärtustamine Scheme's

- Normaaljärjekorda kasutavad keeled võimaldavad opereerida “lõpmatute objektidega”

```
(define nats-from  
  (lambda (n)  
    (cons n (nats-from (+ 1 n)))))
```

```
(define nats (nats-from 0))
```

- Kuna Scheme kasutab aplikatiivset järjekorda, siis läheb lõpmatusse tsüklisse!!

Laisk-väärtustamine Scheme's

- Lõpmatute objektide modelleerimiseks saame kasutada parameetriteta protseduure (ingl. *thunk*)
- Näide — striimid:

```
(define make-stream  
  (lambda (value thunk)  
    (cons value thunk)))
```

```
(define the-null-stream  
  (make-stream "end-of-stream"  
              (lambda () the-null-stream)))
```

Laisk-väärtustamine Scheme's

- Striimid (järg):

```
(define stream-car car)
```

```
(define stream-cdr  
  (lambda (stream)  
    ((cdr stream))))
```

```
(define stream-null?  
  (lambda (stream)  
    (eq? stream the-null-stream)))
```

Laisk-väärtustamine Scheme's

- Striimid (järg):

```
(define show-stream
  (lambda (str n)
    (if (zero? n)
        (list '...)
        (cons (stream-car str)
              (show-stream (stream-cdr str)
                           (- n 1))))))
```

Laisk-väärtustamine Scheme's

```
(define nats-from  
  (lambda (n)  
    (make-stream n (lambda ()  
                    (nats-from (+ 1 n))))))
```

```
(define nats (nats-from 0))
```

```
> (show-stream nats 10)  
(0 1 2 3 4 5 6 7 8 9 ...)
```

Laisk-väärtustamine Scheme's

- Fibonacci jada:

```
(define fibonacci
  (letrec ((fib01 (make-stream 0
                              (lambda () fib02)))
           (fib02 (make-stream 1
                              (lambda () fib03)))
           (fib03 (add-streams fib01 fib02)))
    fib01))
```

Laisk-väärtustamine Scheme's

- Fibonacci jada (järg):

```
(define add-streams
  (lambda (str1 str2)
    (make-stream (+ (stream-car str1)
                    (stream-car str2))
                 (lambda ()
                   (add-streams
                    (stream-cdr str1)
                    (stream-cdr str2))))))
```

```
> (show-stream fibonacci 10)
(0 1 1 2 3 5 8 13 21 34 ...)
```

Laisk-väärtustamine Scheme's

- Memo-striimid:

```
(define stream-cdr
  (lambda (stream)
    (if (pair? (cdr stream))
        (cdr stream)
        (let ((s ((cdr stream))))
            (set-cdr! stream s)
            s))))
```

Järgmiseks korraks

- Lugeda läbi EOPL ptk. 4.4, 4.7