Imperatiivsed konstruktsioonid

- Seni vaadeldud Scheme alamhulk on puhtalt funktsionaalne; so. programmi täitmise ainsaks tulemuseks on tema väärtus
- Enamik keeli on imperatiivsed, kus programmi täidetakse mitte konkreetse väärtuse leidmiseks, vaid mingi kõrvalefekti tekitamiseks
  - sisend/väljund
  - omistamine, “muteeritavad” muutujad
  - efektide järjekorras täitmine

Efektide järjekorras täitmine

- Begin-avaldiste süntaks:

  $\langle expr \rangle$ ::= ...

  | (begin $\langle expr \rangle \ldots \langle expr \rangle$)

- Avaldises (`begin` $E_1 \ldots E_n$)
  - väärtustatakse avaldised $E_1 \ldots E_n$ üksteise järel vasakult paremale
  - viimasena väärtustatava avaldise E_n väärtus on kogu avaldise väärtuseks

Efektide järjekorras täitmine

```
> (begin
    (display "Here is my answer: ")
    (display (+ 2 4 (* 4 5))))
Here is my answer: 26

> (begin
    (display "Here is my answer: ")
    (newline)
    (display (+ 2 4 (* 4 5))))
Here is my answer:
26
```

Efektide järjekorras täitmine

```
(define for-each
  (lambda (proc lst)
    (if (null? lst)
        'done
        (begin
          (proc (car lst))
          (for-each proc (cdr lst))))))
```

Efektide järjekorras täitmine

```
(define displayln
  (lambda lst
    (begin
      (for-each display lst)
      (newline))))

>  (displayln "The answer is: " (+ 3 4 5))
The answer is: 12
```

Efektide järjekorras täitmine

- Avaldiste järjekorrad võivad esineda mõnedes teistes keelekonstruktsioonides (s.h. `lambda`, `let`, `letrec` kehad ning `cond`, `case`, `variant-case` alternatiivid)
- Selliseid avaldiste järjekordi käsitletakse begin-avaldise süntaktilise suhkruna

```
(define displayln
  (lambda lst
    (for-each display lst)
    (newline)))
```

Sisend/väljund operatsioone

- `write` trükib objekti esituse
- `read-char` loeb konsoolilt ühe sümboli
- `read` loeb terve objekti

```
(define read-eval-print
  (lambda ()
    (display "-->")
    (write (eval (read)))
    (newline)
    (read-eval-print)))
```

Sisend/väljund operatsioone

```
> (read-eval-print)
-->3
3
-->(+ 1 2 3)
6
-->"Hello"
"Hello"
-->(car 3)
car: expects argument of type <pair>; given 3
>
```

Andmestruktuuride “muteerimine”

```
> (define lst (list 1 2))
> (set-car! lst 3)
> lst
(3 2)
> (set-cdr! lst (list 4 5))
> lst
(3 4 5)
> (set-car! (cdr lst) 6)
> lst
(3 6 5)
```

Andmestruktuuride “muteerimine”

```
> (define p (cons 1 '()))
> (begin (set-cdr! p p) 'done)
done
> (car p)
1
> (cadr p)
1
> (caddr p)
1
> p
#0=(1 . #0#)
```

Andmestruktuuride “muteerimine”

```
(define reverse!
  (letrec ((loop
            (lambda (last ls)
              (let ((next (cdr ls)))
                (set-cdr! ls last)
                (if (null? next)
                    ls
                    (loop ls next))))))
    (lambda (ls)
      (if (null? ls) ls (loop '() ls)))))
```

Andmestruktuuride “muteerimine”

```
> (define c (cons 3 '()))
> (define b (cons 2 c))
> (define a (cons 1 b))
> (reverse! a)
(3 2 1)
> a
(1)
> b
(2 1)
> c
(3 2 1)
```

Omistamine

- `set!` spetsiaalvorm

  $\langle expr \rangle$ ::= ...
  | (set! $\langle variable \rangle$ $\langle expr \rangle$)

- Avaldis (`set!` *x exp*) omitsab muutujale *x* avaldise *exp* väärtuse
- Muutuja *x* tähistab aadressit (ja mitte muutuja väärtust)
- Selleks et muutujale omistada, peab muutuja olema enne defineeritud

Omistamine

```
> (define x 1)
> (set! x 2)
> x
2
> (set! x (+ x 2))
> x
4
> (let ((y 3))
    (set! y 4)
    (+ y 1))
5
```

Magasin (ver. 1)

```
(define empty? '*)
(define push! '*)
(define pop! '*)
(define top '*)

(let ((stk '()))
  (set! empty?
        (lambda () (null? stk)))
  (set! push!
        (lambda (x) (set! stk (cons x stk))))
  ...
```

Magasin (ver. 1)

```
...
(set! pop!
      (lambda ()
        (if (empty?)
            (error "Stack empty")
            (set! stk (cdr stk)))))
(set! top
      (lambda ()
        (if (empty?)
            (error "Stack empty")
            (car stk)))))
```

Magasin (ver. 1)

```
> (push! 1)
> (push! 2)
> (top)
2
> (pop!)
> (top)
1
> (pop!)
> (empty?)
#t
```

Magasin (ver. 2)

```
(define stack
  (let ((stk '()))
    (lambda (message)
      (case message
        ((empty?) (lambda ()
                    (null? stk)))
        ((push!) (lambda (x)
                   (set! stk (cons x stk))))
        ...
```

Magasin (ver. 2)

```
...
((pop!) (lambda ()
          (if (null? stk)
              (error "Stack empty")
              (set! stk (cdr stk)))))
((top) (lambda ()
         (if (null? stk)
             (error "Stack empty")
             (car stk))))
(else (error "stack: Invalid message" message))))))
```

Magasin (ver. 2)

```
> ((stack 'push!) 1)
> ((stack 'push!) 2)
> ((stack 'top))
2
> ((stack 'pop!))
> ((stack 'top))
1
> ((stack 'pop!))
> ((stack 'empty?))
#t
```

Magasin (ver. 3)

```
(define make-stack
  (lambda ()
    (let ((stk '()))
      (lambda (message)
        (case message
          ((empty?) (lambda ()
                      (null? stk)))
          ((push!) (lambda (x)
                     (set! stk (cons x stk))))
          ...
```

Magasin (ver. 3)

```
...
((pop!) (lambda ()
          (if (null? stk)
              (error "Stack empty")
              (set! stk (cdr stk)))))
((top) (lambda ()
          (if (null? stk)
              (error "Stack empty")
              (car stk))))
(else (error "stack: Invalid message" message))))))
```

Magasin (ver. 3)

```
> (define s1 (make-stack))
> (define s2 (make-stack))
> ((s1 'push!) 1)
> ((s2 'push!) 2)
> ((s1 'top))
1
> ((s2 'top))
2
```

Järgmiseks korraks

- Lugeda läbi EOPL ptk. 4.5, 4.6
- Saata 1. kodutöö lahendused
  - NB! Lahendustes mitte kasutada imperatiivseid konstruktsioone