Näiteid jätkude rakendustest: erindite töötlus

- Leida arvude listi elementide korrutis:

```
(define allprod
  (lambda (lst)
    (if (null? lst)
        1
        (* (car lst) (allprod (cdr lst))))))
```

- Kui argumentlist sisaldab nulli, siis on tulemus alati null!

```
> (allprod '(1 2 3 0 1 2 3 4 5 6 7))
0
```

- Toodud definitsiooni korral läbitakse kogu list ja korrutatakse kõik elemendid

Näiteid jätkude rakendustest: erindite töötlus

- Akumulaatoriga versioon:

```
(define allprod-acc
  (lambda (lst acc)
    (if (null? lst) acc
        (if (zero? (car lst)) 0
            (allprod-acc (cdr lst)
                         (* (car lst) acc))))))

> (allprod-acc '(1 2 3 0 1 2 3 4 5 6 7) 1)
0
```

- List läbitakse ainult kuni (esimese) nullini!
- Tehakse nii mitu korrutamist, kui mitu elementi läbiti!

Näiteid jätkude rakendustest: erindite töötlus

- Jätkudega versioon:

```
(define allprod-k
  (lambda (lst k)
    (if (null? lst) (k 1)
        (if (zero? (car lst)) 0
            (allprod-k (cdr lst)
                       (lambda (z) (k (* (car lst) z))))))))

> (allprod-k '(1 2 3 0 1 2 3 4 5 6 7) (lambda (v) v))
0
```

- List läbitakse ainult kuni (esimese) nullini!
- Ei tehta ühtegi korrutamist!!

Näiteid jätkude rakendustest: mitu resutaati

- Leida listi pikkus ja elementide summa:

```
(define sum-and-length
  (lambda (lst acc-sum acc-len)
    (if (null? lst)
        (cons acc-sum acc-len)
        (sum-and-length (cdr lst)
            (+ acc-sum (car lst)) (+ acc-len 1)))))
```

- Ehitab paari, mille järgmine protseduur, mis tahab tulemust kasutada, peab kohe “tükeldama”!

```
(define average
  (lambda (lst)
    (let ((pair (sum-and-length lst 0 0)))
      (/ (car pair) (cdr pair)))))
```

Näiteid jätkude rakendustest: mitu resutaati

- CPS versioon:

```
(define sum-and-length-k
  (lambda (lst acc-sum acc-len k)
    (if (null? lst)
        (k acc-sum acc-len)
        (sum-and-length-k (cdr lst)
            (+ acc-sum (car lst)) (+ acc-len 1) k))))

(define average-k
  (lambda (lst)
    (sum-and-length-k lst 0 0
         (lambda (sum len) (/ sum len)))))
```

- Resultaadid antakse otse edasi!

Jätkude esitamine andmestruktuuridena

CPS-kujul `remove`

```
(define remove
  (lambda (s los)
    (remove-cps s los (lambda (v) v))))

(define remove-cps
  (lambda (s los k)
    (cond
      ((null? los) (k '()))
      ((eq? s (car los)) (remove-cps s (cdr los) k))
      (else (remove-cps s (cdr los)
                        (lambda (v) (k (cons (car los) v))))))))
```

Jätkude esitamine andmestruktuuridena

Jätkude esitusest sõltumatu CPS-kujul `remove`

```
(define remove
  (lambda (s los)
    (remove-cps s los (make-final-valcont))))

(define remove-cps
  (lambda (s los k)
    (cond
      ((null? los) (apply-continuation k '()))
      ((eq? s (car los)) (remove-cps s (cdr los) k))
      (else (remove-cps s (cdr los)
                        (make-rem1 los k))))))
```

Jätkude esitamine andmestruktuuridena

Jätkude esitus protseduuride abil

```
(define make-final-valcont
  (lambda ()
    (lambda (v) v)))

(define make-rem1
  (lambda (los k)
    (lambda (v)
      (apply-continuation k (cons (car los) v)))))

(define apply-continuation
  (lambda (k v)
    (k v)))
```

Jätkude esitamine andmestruktuuridena

Jätkude esitus andmestruktuuride abil

```
(define-record final-valcont ())
(define-record rem1 (los k))

(define apply-continuation
  (lambda (k v)
    (variant-case k
      (final-valcont () v)
      (rem1 (los k)
        (apply-continuation k (cons (car los) v))))))
```

Lähte interpretaator

```
(define eval-exp
  (lambda (exp env)
    (variant-case exp
      (lit (datum) datum)
      (varref (var) (apply-env env var))
      (app (rator rands)
        (let ((proc (eval-exp rator env))
              (args (eval-rands rands env)))
          (apply-proc proc args)))
      ...
```

Lähte interpretaator

```
      (if (test-exp then-exp else-exp)
        (if (true-value? (eval-exp test-exp env))
            (eval-exp then-exp env)
            (eval-exp else-exp env)))
      (proc (formals body) (make-closure formals body env))
      (else (error "Invalid abstract syntax:" exp)))))

(define eval-rands
  (lambda (rands env)
    (if (null? rands)
        '()
        (cons (eval-exp (car rands) env)
              (eval-rands (cdr rands) env)))))
```

Lähte interpretaator

```
(define apply-proc
  (lambda (proc args)
    (variant-case proc
      (prim-proc (prim-op)
        (apply-prim-op prim-op args))
      (closure (formals body env)
        (eval-exp body (extend-env formals args env)))
      (else (error "Invalid procedure:" proc)))))
```

CPS-kujul interpretaator

```
(define eval-exp
  (lambda (exp env k)
    (variant-case exp
      (lit (datum) (k datum))
      (varref (var) (k (apply-env env var)))
      (app (rator rands)
        (eval-exp rator env
          (lambda (proc)
            (eval-rands rands env
              (lambda (all)
                (apply-proc proc all k))))))
      ...
```

CPS-kujul interpretaator

```
...
(if (test-exp then-exp else-exp)
  (eval-exp test-exp env
    (lambda (test)
      (if (true-value? test)
          (eval-exp then-exp env k)
          (eval-exp else-exp env k)))))
(proc (formals body)
  (k (make-closure formals body env)))
(else (error "Invalid abstract syntax:" exp)))))
```

CPS-kujul interpretaator

```
(define eval-rands
  (lambda (rands env k)
    (if (null? rands)
        (k '())
        (eval-exp (car rands) env
          (lambda (first)
            (eval-rands (cdr rands) env
              (lambda (rest)
                (k (cons first rest)))))))))
```

CPS-kujul interpretaator

```
(define apply-proc
  (lambda (proc args k)
    (variant-case proc
      (prim-proc (prim-op)
        (k (apply-prim-op prim-op args)))
      (closure (formals body env)
        (eval-exp body (extend-env formals args env) k))
      (else (error "Invalid procedure:" proc)))))
```

Esitusest sõltumatu CPS-kujul interpretaator

```
(define eval-exp
  (lambda (exp env k)
    (variant-case exp
      (lit (datum)
        (apply-continuation k datum))
      (varref (var)
        (apply-continuation k (apply-env env var)))
      (app (rator rands)
        (eval-exp rator env
          (make-proc-valcont rands env k)))
      ...
```

Esitusest sõltumatu CPS-kujul interpretaator

```
...
(if (test-exp then-exp else-exp)
  (eval-exp test-exp env
    (make-test-valcont then-exp else-exp env k)))
(proc (formals body)
  (apply-continuation k
    (make-closure formals body env)))
(else (error "Invalid abstract syntax:" exp)))))
```

Esitusest sõltumatu CPS-kujul interpretaator

```
(define eval-rands
  (lambda (rands env k)
    (if (null? rands)
        (apply-continuation k '())
        (eval-exp (car rands) env
          (make-first-valcont rands env k)))))
```

Esitusest sõltumatu CPS-kujul interpretaator

```
(define apply-proc
  (lambda (proc args k)
    (variant-case proc
      (prim-proc (prim-op)
        (apply-continuation k
          (apply-prim-op prim-op args)))
      (closure (formals body env)
        (eval-exp body (extend-env formals args env) k))
      (else (error "Invalid procedure:" proc)))))
```

Jätkude protseduurne esitus

```
(define apply-continuation
  (lambda (k val) (k val)))

(define make-proc-valcont
  (lambda (rands env k)
    (lambda (proc)
      (eval-rands rands env
         (make-all-argcont proc k)))))

(define make-all-argcont
  (lambda (proc k)
    (lambda (all)
      (apply-proc proc all k))))
```

Jätkude protseduurne esitus

```
(define make-first-valcont
  (lambda (rands env k)
    (lambda (first)
      (eval-rands (cdr rands) env
        (make-rest-argcont first k)))))

(define make-rest-argcont
  (lambda (first k)
    (lambda (rest)
      (apply-continuation k (cons first rest)))))
```

Jätkude protseduurne esitus

```
(define make-test-valcont
  (lambda (then-exp else-exp env k)
    (lambda (test)
      (if (true-value? test)
          (eval-exp then-exp env k)
          (eval-exp else-exp env k)))))

(define make-final-valcont
  (lambda ()
    (lambda (final) final)))
```

Jätkude esitus andmestruktuuridena

```
(define-record final-valcont ())
(define-record proc-valcont (rands env k))
(define-record all-argcont (proc k))
(define-record test-valcont (then-exp else-exp env k))
(define-record first-valcont (rands env k))
(define-record rest-argcont (first k))
```

Jätkude esitus andmestruktuuridena

```
(define apply-continuation
  (lambda (k val)
    (variant-case k
      (final-valcont ()
        (let ((final val))
          final))
      (proc-valcont (rands env k)
        (let ((proc val))
          (eval-rands rands env
            (make-all-argcont proc k))))
      (all-argcont (proc k)
        (let ((all val))
          (apply-proc proc all k)))
```

Jätkude esitus andmestruktuuridena

```
...
(test-valcont (then-exp else-exp env k)
  (let ((test val))
    (if (true-value? test)
        (eval-exp then-exp env k)
        (eval-exp else-exp env k))))
(first-valcont (rands env k)
  (let ((first val))
    (eval-rands (cdr rands) env
      (make-rest-argcont first k))))
(rest-argcont (first k)
  (let ((rest val))
    (apply-continuation k (cons first rest)))))))
```

Järgmiseks korraks

- Lugeda läbi EOPL ptk. 9.1, 9.2
- “Mängida” interpretaatoriga:
  - `loeng20-1.ss` (CPS-kujul interpretaator)
  - `loeng20-2.ss` (protseduurse esitusega jätkud)
  - `loeng20-3.ss` (kirjete abil esitatud jätkud)