Lokaalsed definitsioonid

- Lokaalsete definitsioonide BNF ja abstraktne süntaks:

⟨*exp*⟩	::=	...	
	|	`let` ⟨*decls*⟩ `in` ⟨*exp*⟩	`let (decls body)`
⟨*decls*⟩	::=	⟨*decl*⟩ {`;` ⟨*decl*⟩}*	
⟨*decl*⟩	::=	⟨*var*⟩ = ⟨*exp*⟩	`decl (var exp)`

- Näide:

```
let x = 5;
    y = 6
in +(x, y)
```

Keskonnaga 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)))
      ...
```

Keskonnaga 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)))
      (else (error "Invalid abstract syntax:" exp)))))

(define eval-rands
  (lambda (rands env)
    (map (eval-rand env) rands)))
```

Keskonnaga interpretaator

```
(define eval-rand
  (lambda (env)
   (lambda (exp)
    (eval-exp exp env))))

(define run
  (lambda (x)
    (eval-exp (parse x) init-env)))
```

Lokaalsed definitsioonid

- Let-avaldiste väärtustamine:

```
(define eval-exp (lambda (exp env)
    (variant-case exp
      ...
      (let (decls body)
        (let ((vars (map decl->var decls))
              (exps (map decl->exp decls)))
          (let ((new-env (extend-env vars
                                     (eval-rands exps env)
                                     env)))
            (eval-exp body new-env))))
      ...
```

Lokaalsed definitsioonid

- Näited:

```
--> "let x = 1
     in let y = +(x,3);
            z = 4
     in add1(-(y,z))"
1
--> "let x = 1
     in let x = +(x,2)
        in add1(x)"
4
```

Protseduuride defineerimine

- Protseduuri definitsioonide BNF ja abstraktne süntaks:

$$\begin{array}{rcll} \langle exp \rangle & ::= & \ldots & \\ & | & \texttt{proc}\ \langle varlist \rangle\ \langle exp \rangle & \texttt{proc (formals body)} \\ \langle varlist \rangle & ::= & \texttt{()} \mid \texttt{(}\langle vars \rangle\texttt{)} & \\ \langle vars \rangle & ::= & \langle var \rangle\ \{\texttt{,}\ \langle var \rangle\}^* & \end{array}$$

- Näiteid:

```
let f = proc(y,z) *(y, -(z, 5))
in f(2, 8)
```

```
(proc (y,z) *(y, -(z, 5)))(2, 8)
```

Protseduurid

- Protseduuris olevad vabad muutujad viitavad väärtustele, mis neil olid protseduuri defineerimise hetkel!

```
let x = 5 in
   let f = proc(y,z) *(y, -(z, x)) ;
       x = 28 in
     f(2, 8)
```

- Protseduuride rakendamisel võib keskond olla muutunud; seetõttu "pakitakse" protseduurid koos defineerimisaegse keskonnaga nn. sulundiks (ingl. *closure*)

```
(define-record closure (formals body env))
```

Protseduurid

- Protseduurid defineerimisel “pakime” protseduuri koos hetkel kehtiva keskonnaga:

```
(define eval-exp
  (lambda (exp env)
    (variant-case exp
      ...
       (proc (formals body)
         (make-closure formals body env))
      ...
```

Protseduurid

Protseduuri rakendamisel lisame sulundis olevale keskonnale protseduuri parameetritele vastavad seosed ning seejärel väärtustame keha kasutades seda keskonda:

```
(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)))))
```

Protseduurid

- Näited:

```
--> "(proc (y,z) *(y, -(z, 5)))(2, 8)"
6
--> "let x = 5 in
       let f = proc(y,z) *(y, -(z, x)) ;
           x = 28 in
       f(2, 8)"
6
```

Omistamine

- Muutujate omistamise BNF ja abstraktne süntaks:

$\langle exp \rangle$ ::= ...

| $\langle var \rangle := \langle exp \rangle$ `varassign (var exp)`

- Seni olid muutujad seotud otse väärtustega
- Omistamise modelleerimiseks seome muutjaga selle "mälupesa" aadressi kuhu väärtus on salvestatud

$$\begin{aligned} \textit{Expressed Value} &= \textit{Number} + \textit{Procedure} \\ \textit{Denoted Value} &= \textit{Cell}(\textit{Expressed Value}) \end{aligned}$$

Omistamine

- Pesade esitamiseks kasutame järgmiste operatsioonidega abstraktset andmetüüpi:
  - `(make-cell v)` — loob uue pesa ja salvestab sinna väärtuse `v`
  - `(cell-ref c)` — loeb pesast `c` sinna salvestatud väärtuse
  - `(cell-set! c v)` — salvestab pesasse `c` väärtuse `v`

Omistamine

- Interpretaatoris peame modifitseerima kõik keskonna poole pöördumised:

```
(define eval-exp
  (lambda (exp env)
    (variant-case exp
      (varref (var) (cell-ref (apply-env env var)))
      ...

(define eval-rand
  (lambda (env)
    (lambda (exp)
      (make-cell (eval-exp exp env)))))
```

Omistamine

```
(define apply-proc
  (lambda (proc args)
    (variant-case proc
      (prim-proc (prim-op)
         (apply-prim-op prim-op (map cell-ref args)))
      ...

(define init-env
  (extend-env
    prim-op-names
    (map make-cell (map make-prim-proc prim-op-names))
    the-empty-env))
```

Omistamine

- Omistamise korral asendame pesas oleva väärtuse uue väärtusega:

```
(define eval-exp
  (lambda (exp env)
    (variant-case exp
      ...
      (varassign (var exp)
        (cell-set! (apply-env env var)
                   (eval-exp exp env)))
      ...
```

Järjekorras väärtustamine

- Begin-avaldiste BNF ja abstraktne süntaks:

  ⟨*exp*⟩ ::= ...

  | `begin` ⟨*exp*⟩ ⟨*compound*⟩

  `begin (exp1 exp2)`

  ⟨*compound*⟩ ::= {`;` ⟨*exp*⟩}* `end`

- Näide:

```
let x = 3 in
begin x := add1(x);
      x := +(x, x);
      +(x, 2)
end
```

Järjekorras väärtustamine

```
(define eval-exp
  (lambda (exp env)
    (variant-case exp
      ...
      (begin (exp1 exp2)
        (eval-exp exp1 env)
        (eval-exp exp2 env))
      ...
```

Järgmiseks korraks

- Lugeda läbi EOPL ptk. 5.3 – 5.5
- “Mängida” interpretaatoritega
  - failis `loeng11-1.ss` olev interpretaator “tunneb” lokaalseid definitsioone ja protseduuride definitsioone;
  - failis `loeng11-2.ss` olev interpretaator “tunneb” lisaks veel omistamist ja järjestikku väärtustamist