L19 & L20. Mutable Data Structure

유형	강의
복습 여부	☑
Status	In progress

We must force the interpreter to return not only the value of each expression but also an updated store that reflects mutations made in the process of computing that value.

cf. Racket에 존재하는 Box operator를 사용하면 State을 사용하는 것임

Implementing Boxes without State

```
;interp: BFAE DefrdSub Store -> Value*Store

(define-type BFAE-Value
   [numV (n number?)]
   [closureV (param symbol?) (body BFAE?) (ds DefrdSub?)]
   [boxV (address integer?)])

(define-type Value*Store
   [v*s (value BFAE-Value?) (store Store?)])
```

Q. Why boxV has only address for its field?

In the box there must be value. However, to deal with all the values in the box, we are going to use address to point to the value.

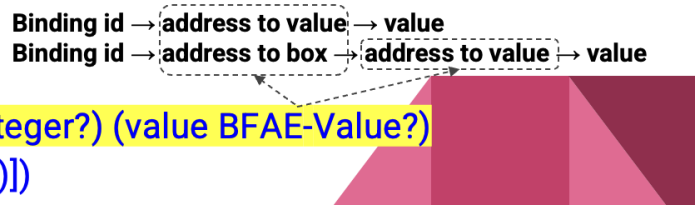
- We need two repositories (caches)
  - One for keeping a memory address value of a box for static scope

    (define-type DefrdSub
     [mtSub]
     [aSub (name symbol?) (address integer?) (ds DefrdSub?)])

  - Another for maintaining dynamic changes of boxes.

    ⇒ Let's call this cache as 'store'

    (define-type Store
     [mtSto]
     [aSto (address integer?) (value BFAE-Value?)
      (rest Store?)])



Ex1.

{newbox {+ 2 3}}
⇒ (v*s (boxV 1) (aSto 1 (numV 5) (mtSto)))

Ex2.

7 ⇒ (v*s (numV 7) (mtSto))
{+ 7 6} ⇒ (v*s (numV 13) (mtSto))
{with {b {newbox 7}}
 {seqn {setbox b 10}
  {openbox b}}}

(1)
Address assigned to 'b'

⇒ (v*s (numV 10) (aSto 1 (numV 10) (aSto 2 (boxV 1) (aSto 1 (numV 7) (mtSto)))))

BFAE

```
; BFAE
(define-type BFAE
  [num (n number?)]
  [add (lhs BFAE?) (rhs BFAE?)]
  [sub (lhs BFAE?) (rhs BFAE?)]
  [id (name symbol?)]
```

```
  [fun (param symbol?) (body BFAE?)]
  [newbox (v BFAE?)]
  [setbox (bn BFAE?) (v BFAE?)]
  [openbox (v BFAE?)]
  [seqn (ex1 BFAE?) (ex2 BFAE?)]
  [app (ftn BFAE?) (arg BFAE?)])
```

```
; DefrdSub
(define-type DefrdSub
  [mtSub]
  [aSub (name symbol?) (address integer?) (ds DefrdSub?)])
```

```
; BFAE-Value
; keep a memory address value of a box for static scope
(define-type BFAE-Value
  [numV (n number?)]
  [closureV (param symbol?) (body BFAE?) (ds DefrdSub?)]
  [boxV (address integer?)])

; Store
; track dynamic changes of boxes
(define-type Store
  [mtSto]
  [aSto (address integer?) (value BFAE-Value?) (rest Store?)]

; Value*Store
; return type holds both the value and the storage informatio
(define-type Value*Store
  [v*s (value BFAE-Value?) (store Store?)])
```

Parser

```
; parse : sexp -> BFAE
(define (parse sexp)
  (match sexp
    [(? number?) (num sexp)]
```

```
    [(list '+ l r) (add (parse l) (parse r))]
    [(list '- l r) (sub (parse l) (parse r))]
    [(list 'with (list i v) e) (app (fun i (parse e)) (parse
    [(? symbol?) (id sexp)]
    [(list 'newbox v) (newbox (parse v))]
    [(list 'setbox i v) (setbox (parse i) (parse v))]
    [(list 'openbox i) (openbox (parse i))]
    [(list 'seqn ex1 ex2) (seqn (parse ex1) (parse ex2))]
    [(list 'fun (list p) b) (fun p (parse b))]
    [(list f a) (app (parse f) (parse a))]
    [else (error 'parse "bad syntax: ~a" sexp)]))
```

Interpreter

```
; Interpreter
(define (interp expr ds st)
  (type-case BFAE expr
    [num (n) (v*s (numV n) st)]
    [add (l r) (type-case Value*Store (interp l ds st)
                 [v*s (l-value l-store)
                      (type-case Value*Store (interp r ds l-s
                        [v*s (r-value r-store)
                             (v*s (num+ l-value r-value) r-st
    [sub (l r) (type-case Value*Store (interp l ds st)
                 [v*s (l-value l-store)
                      (type-case Value*Store (interp r ds l-s
                      [v*s (r-value r-store)
                           (v*s (num- l-value r-value) r-stor
    [id (s) (v*s (store-lookup (lookup s ds) st) st)]
    [fun (p b) (v*s (closureV p b ds) st)]
    [app (f a) (type-case Value*Store (interp f ds st)
                 [v*s (f-value f-store)
                      (type-case Value*Store (interp a ds f-s
                        [v*s (a-value a-store)
                             (local ([define new-address (mal
                               (interp (closureV-body f-value
                                       (aSub (closureV-param
```

```
                                              new-address
                                              (closureV-ds f-v
                                         (aSto new-address
                                              a-value
                                              a-store)))])])]
    [newbox (val)
            (type-case Value*Store (interp val ds st)
              [v*s (vl st1)
                   (local [(define a (malloc st1))]
                     (v*s (boxV a)
                          (aSto a vl st1)))])]

    [openbox (bx-expr)
             (type-case Value*Store (interp bx-expr ds st)
               [v*s (bx-val st1)
                    (v*s (store-lookup (boxV-address bx-val)

    [setbox (bx-expr val-expr)
            (type-case Value*Store (interp bx-expr ds st)
              [v*s (bx-val st2)
                   (type-case Value*Store (interp val-expr ds
                     [v*s (val st3)
                          (v*s val
                               (aSto (boxV-address bx-val)
                                     val
                                     st3))])])]

    [seqn (a b) (type-case Value*Store (interp a ds st)
                  [v*s (a-value a-store)
                       (interp b ds a-store)])]
    ))
```

Supporting functions

```
; num-op: (number number ->number) -> (FWAE FWAE -> FWAE)
(define (num-op op)
  (lambda (x y)
    (numV (op (numV-n x) (numV-n y)))))
```

```
(define num+ (num-op +))
(define num- (num-op -))

; lookup: symbol DefrdSub -> address
(define (lookup name ds)
  (type-case DefrdSub ds
    [mtSub () (error 'lookup "free identifier")]
    [aSub (i adr saved) (if (symbol=? i name)
                            adr
                            (lookup name saved))]))

; store-lookup: address store -> BFAE-Value
(define (store-lookup address sto)
  (type-case Store sto
    [mtSto () (error 'store-lookup "No value at address")]
    [aSto (location value rest-store)
          (if (= location address)
              value
              (store-lookup address rest-store))]))

; malloc: Store -> Integer
; to allocate memory for a new box
(define (malloc st)
  (+ 1 (max-address st)))

; max-address: Store -> Integer
(define (max-address st)
  (type-case Store st
    [mtSto () 0]
    [aSto (n v st) (max n (max-address st))]))

; run
(define (run sexp ds st)
     (interp (parse sexp) ds st))
```

```
(run '7 (mtSub) (mtSto))
(run '{+ 7 6} (mtSub) (mtSto))
```

```
(run '{newbox {+ 2 3}} (mtSub) (mtSto))
(run '{with {b {newbox {+ 2 3}}} {openbox b}} (mtSub) (mtSto)
(run '{with {b {newbox 7}} {seqn {setbox b 10} {openbox b}}}
(run '{+ {with {b {newbox 10}}
                {seqn {setbox b 7}
                      {openbox b}}}
         {with {b {newbox 10}}
                {seqn {setbox b 5}
                      {openbox b}}}}
     (mtSub) (mtSto))
```

```
(v*s (numV 7) (mtSto))
(v*s (numV 13) (mtSto))
(v*s (boxV 1) (aSto 1 (numV 5) (mtSto)))
(v*s (numV 5) (aSto 2 (boxV 1) (aSto 1 (numV 5) (mtSto))))
(v*s (numV 10) (aSto 1 (numV 10) (aSto 2 (boxV 1) (aSto 1 (numV 7) (mtSto)))))
(v*s (numV 12) (aSto 3 (numV 5) (aSto 4 (boxV 3) (aSto 3 (numV 10) (aSto 1 (numV 7) (aSto 2 (boxV 1) (aSto 1 (numV 10) (mtSto))))))))
```

add, sub, app, setbox, seqn need the same sort of sequencing.

We can avoid the repetitive operation by implementing interp-two function.

```
; interp-two
(define (interp-two expr1 expr2 ds st handle)
  (type-case Value*Store (interp expr1 ds st)
    [v*s (val1 st2)
         [type-case Value*Store (interp expr2 ds st2)
           [v*s (val2 st3)
                (handle val1 val2 st3)]]]))
```

```
; Interpreter
(define (interp expr ds st)
  (type-case BFAE expr
    [num (n) (v*s (numV n) st)]
    [add (l r) (interp-two l r ds st (lambda (v1 v2 st1) (v*s
    [sub (l r) (interp-two l r ds st (lambda (v1 v2 st1) (v*s
    [id (s) (v*s (store-lookup (lookup s ds) st) st)]
    [fun (p b) (v*s (closureV p b ds) st)]
    [app (f a) (interp-two f a ds st
                           (lambda (f-value a-value st1)
```

```
                              (local ([define new-address (mal
                                (interp (closureV-body f-value
                                        (aSub (closureV-param
                                              new-address
                                              (closureV-ds f-v
                                        (aSto new-address
                                              a-value
                                              st1)))))]
    [newbox (val)
            (type-case Value*Store (interp val ds st)
              [v*s (vl st1)
                   (local [(define a (malloc st1))]
                     (v*s (boxV a)
                          (aSto a vl st1)))])]

    [openbox (bx-expr)
             (type-case Value*Store (interp bx-expr ds st)
               [v*s (bx-val st1)
                    (v*s (store-lookup (boxV-address bx-val)

    [setbox (bx-expr val-expr)
            (interp-two bx-expr val-expr ds st
                        (lambda (bx-val val st1)
                          (v*s val
                               (aSto (boxV-address bx-val)
                                     val
                                     st1))))]
    [seqn (a b) (interp-two a b ds st
                            (lambda (v1 v2 st1) (v*s v2 st1))
    ; In original code, only one type-case is used. But we ca
    ))
```