L21 & L22. Variables

유형	강의
복습 여부	✓
Status	In progress

Store-Passing Interpreters

Our BFAE interpreter explains state by representing the store as a value.

- every step in computation produces a new store
- the interpreter itself is purely functional

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

```
{with {b {newbox 7}}
        {seqn {setbox b 10}
                 {openbox b}}}
```

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

Instance of v*s type that contains both resulting value and current state of data changes.

We can check how our data changed.

Benefits

We can continue additional operations in a certain point.

Variables

```
(define counter 0)
(define (f x)
    (begin
        (set! counter (+ x counter))  ; set! assigns a value into counter
         counter))
```

an identifier no longer stands for a value; instead, an it stands for a variable.

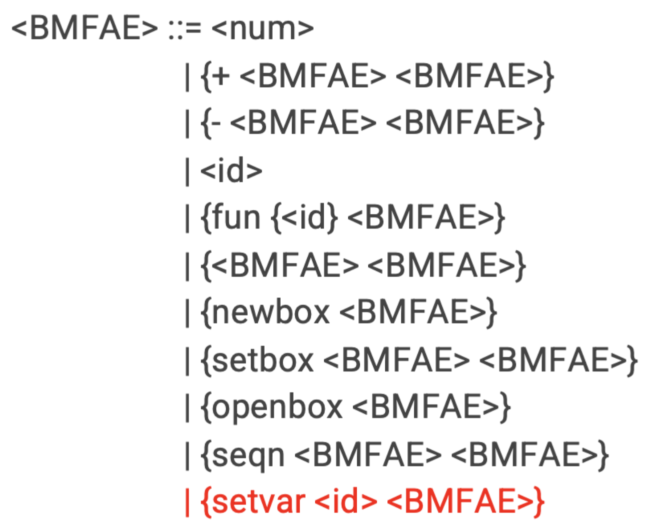
```
(define counter 0)
(define (f x)
    (begin
        (set! counter
                (+ x counter))

         counter))
(f 10)
```

```
(define counter (box 0))
(define (f x)
    (begin
        (set-box! counter
                    (+ (unbox x)
                        (unbox counter)))
         (unbox counter)))
(f (box 10))
```

an identifier no longer stands for a value; instead, it stands for a variable.

essentially the same, but hide the boxes in the interpreter.

BMFAE = BFAE + variable



```
<BMFAE> ::= <num>
          | {+ <BMFAE> <BMFAE>}
          | {- <BMFAE> <BMFAE>}
          | <id>
          | {fun {<id>} <BMFAE>}
          | {<BMFAE> <BMFAE>}
          | {newbox <BMFAE>}
          | {setbox <BMFAE> <BMFAE>}
          | {openbox <BMFAE>}
          | {seqn <BMFAE> <BMFAE>}
          | {setvar <id> <BMFAE>}
```

- Box
  - an example of how our language can support mutable data structure
  - we learned the fundamentals of how we can implement variables
- Variable
  - based on the concept of address
    - It’s possible to represent variable concept using box.
    - By adding new syntax, (setvar <id><BMFAE>), we can simply make our language support variables.
  - memorizing is essential part in variable
    - In BMFAE, we are saving the value bounded with the identifier in our memory. Then, by updating the value for the address of the identifier, we can maintain its changes.

```
{with {a 3} {setvar a 5}}
; a : identifier bound with value 3
{with {a 3} {seqn {{fun {x} {setvar x 5}} a} a}}
```

By adopting the concept of box, we can update our deferred substitution cache in our store to support address concept.

Call-by-value

When a function is called, malloc generates a new address for the function parameter.

When we make a new function call, the parameter gets new address.

BMFAE

```
(define-type BMFAE
  [num (n number?)]
  [add (lhs BMFAE?) (rhs BMFAE?)]
  [sub (lhs BMFAE?) (rhs BMFAE?)]
  [id (name symbol?)]
  [fun (param symbol?) (body BMFAE?)]
  [app (ftn BMFAE?) (arg BMFAE?)]
  [newbox (v BMFAE?)]
  [setbox (bn BMFAE?) (v BMFAE?)]
  [openbox (v BMFAE?)]
  [seqn (ex1 BMFAE?) (ex2 BMFAE?)]
  [setvar (name symbol?) (v BMFAE?)] ; variable
  )
```

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

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

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

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

Parser

```
; parse : sexp -> BMFAE
(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 bn v) (setbox (parse bn) (parse v))]
    [(list 'openbox v) (openbox (parse v))]
    [(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))]
    [(list 'setvar n v) (setvar n (parse v))] ; variable
    [else (error 'parse "bad syntax: ~a" sexp)])
  )
```

Interpreter

```
; Interpreter
(define (interp expr ds st)
  (type-case BMFAE 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
    [setvar (id val-expr) (local [(define a (lookup id ds))]
                            (type-case Value*Store (interp va
                              [v*s (val st)
                                   (v*s val
                                        (aSto a val st))]))])
    )
```

Supporting Functions

```
; num-op: (number number ->number) -> (BMFAE BMFAE -> BMFAE)
(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))]))
```

```
; 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)]]]))

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

```
(run '{with {a 3} {setvar a 5}} (mtSub) (mtSto))
(run '{with {a 3} {seqn {{fun {x} {setvar x 5}} a} a}} (mtSub
```

```
(v*s (numV 5) (aSto 1 (numV 5) (aSto 1 (numV 3) (mtSto))))
(v*s (numV 3) (aSto 2 (numV 5) (aSto 2 (numV 3) (aSto 1 (numV 3) (mtSto)))))
```

```
; we want a to be 20 and b to be 10
(run '{with {swap {fun {x}
                       {fun {y}
                            {with {z x}
                                  {seqn {setvar x y}
                                        {setvar y z}}}}}}}
            {with {a 10}
                  {with {b 20}
                        {seqn {{swap a} b}
                              a}}}} (mtSub) (mtSto))
```

Result

: 10 (incorrect)

```
(v*s (numV 10)
; the right part of seqn
; looks for the value bounded with identifier 'a', which is 1
(aSto 5 (numV 10)
; {setvar y z}
; 'y' identifier's address was 5
; 'z' identifier's value was 10
; update value of 'y' to 10
(aSto 4 (numV 20)
; {setvar x y}
; 'x' identifier's address was 4
; 'y' identifier's value was 20
; update value of 'x' to 20
(aSto 6 (numV 10)
; with {z x}
; 'z' identifier gets new address
(aSto 5 (numV 20)
; 'y' identifier in swap function recieves the value of 'b'
; fun {y}
(aSto 4 (numV 10)
; the left part of seqn, {swap a} b is called
; 'x' identifier in swap function recieves the value of 'a'
; fun {x}
(aSto 3 (numV 20)
; with {b 20}
; 'b' identifier
(aSto 2 (numV 10)
; with {a 10}
; 'a' identifier
(aSto 1 (closureV 'x (fun 'y (app (fun 'z (seqn (setvar 'x (i
; swap function
```

Call-by-reference

We can use box to implement call-by-reference.

```
(run '{with {a {newbox 3}} {seqn {{fun {x} {setbox x 5}} a} {
; for identifier 'a', the address is still 2 and the value is
; however, because the value in address has changed from 3 to
```

▼ Swap function in racket using box

```
; call-by-value
(define (swap_val x y) ; x = 10 y = 20
  (local [(define z y)] ; z = 20
    (set! y x) ; y = 10
    (set! x z))) ; x = 20

(local [(define a 10)
        (define b 20)]
  (begin
    (swap_val a b)
    a)) ; a = 10
```

```
; call-by-reference
(define (swap_ref x y) ; x = (box 10) y = (box 20)
  (local [(define z (box (unbox y)))] ; z = (box 20)
    (set-box! y (unbox x)) ; y = (box 10)
    (set-box! x (unbox z)))) ; x = (box 20)

(local [(define a (box 10))
        (define b (box 20))]
  (begin
    (swap_ref a b)
    (unbox a))) ; a = 20
```

However, we want to avoid the complicated logic that uses box. Instead we want our variables to directly support call-by-reference by updating our interpreter. We don't create new address for our function parameter, but just pass address of the provided identifier.

▼ cf.

When we pass identifier as argument in our function call, then rather creating new address for new parameter of our function, how about if we pass the address of the 'x identifier for the y' identifier?

When we pass identifier as argument in our function call, then rather creating new address for new parameter of our function, how about if we pass the address of the 'x identifier for the y' identifier?

We do not create new memory for the ‘y identifier, but just provide existing address for our argument identifier. So, we just provide 1 for y’ bounded address.

시도 1

```
code for page 38 PPT
```

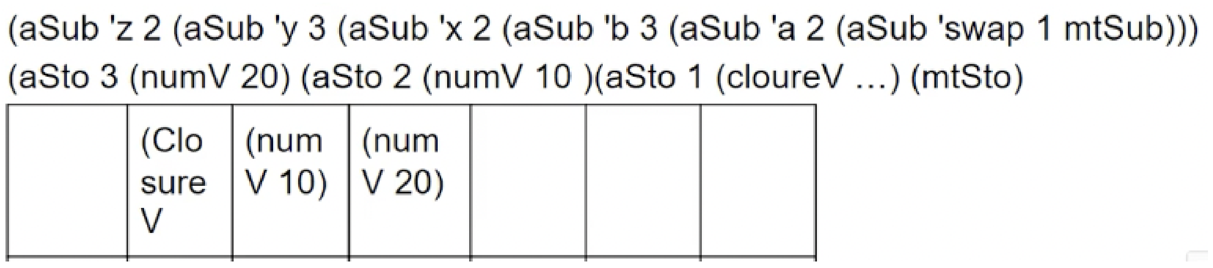
```
; we want 'a' to be 20 and 'b' to be 10
(run '{with {swap {fun {x}
                       {fun {y}
                            {with {z x}
                                  {seqn {setvar x y}
                                        {setvar y z}}}}}}}
            {with {a 10}
                  {with {b 20}
                        {seqn {{swap a} b}
                              b}}}}} (mtSub) (mtSto))
```

Result

: 20 (incorrect)

```
(v*s (numV 20)
(aSto 3 (numV 20)
; {setvar y z}
; We have a problem here!!
; in the address of 'b', the value should be updated to the v
(aSto 2 (numV 20)
; the left part of seqn, {swap a} b is called
; in the address of 'a', update the value to the value of 'b'
; {setvar x y}
(aSto 3 (numV 20)
; with {b 20}
; 'b' identifier
(aSto 2 (numV 10)
; with {a 10}
; 'a' identifier
```

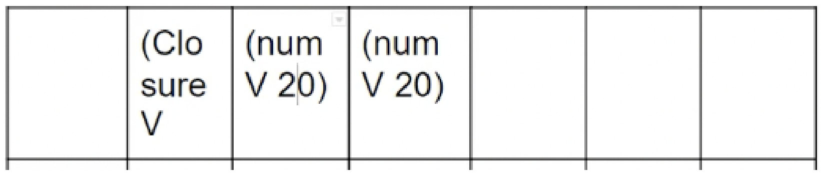
```
(aSto 1 (closureV 'x (fun 'y (app (fun 'z (seqn (setvar 'x (i
; swap function
```



(aSub 'z 2 (aSub 'y 3 (aSub 'x 2 (aSub 'b 3 (aSub 'a 2 (aSub 'swap 1 mtSub)))
(aSto 3 (numV 20) (aSto 2 (numV 10)(aSto 1 (cloureV …) (mtSto)

	(Closure V	(numV 10)	(numV 20)			

{setvar x y}



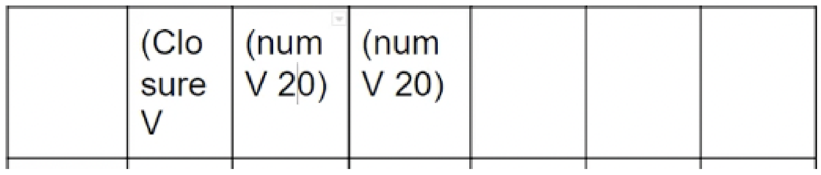
	(Closure V	(numV 20)	(numV 20)			

{setvar y z}

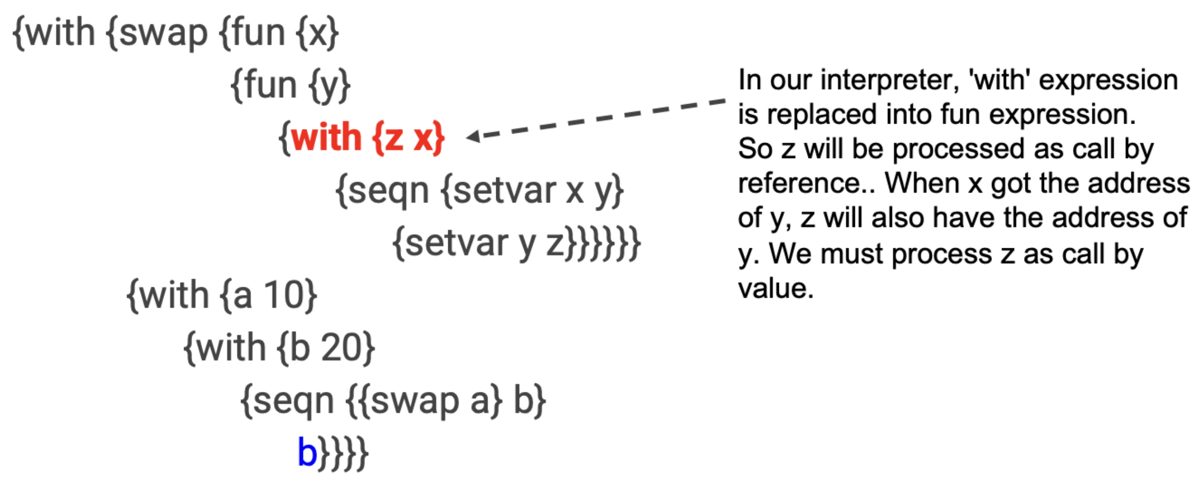
We have to update the value of ‘y’ to the value of ‘z’.

‘z’ has the address of ‘x’.

Then the value of ‘y’ will be updated to the value of the address of ‘x’ which is (numV 20).



	(Closure V	(numV 20)	(numV 20)			



```
{with {swap {fun {x}
              {fun {y}
                 {with {z x}
                    {seqn {setvar x y}
                       {setvar y z}}}}}}
      {with {a 10}
         {with {b 20}
            {seqn {{swap a} b}
               b}}}}
```

In our interpreter, 'with' expression is replaced into fun expression. So z will be processed as call by reference.. When x got the address of y, z will also have the address of y. We must process z as call by value.

We need to separate logics for call-by-value and call-by-reference.

⇒ Add new syntax for function definition that supports call-by-ref for a function call for a given argument as an id.

RBMFAE

```
(define-type RBMFAE
  [num (n number?)]
  [add (lhs RBMFAE?) (rhs RBMFAE?)]
  [sub (lhs RBMFAE?) (rhs RBMFAE?)]
  [id (name symbol?)]
  [fun (param symbol?) (body RBMFAE?)] ; call-by-value
  [refun (param symbol?) (body RBMFAE?)] ; call-by-reference
  [app (ftn RBMFAE?) (arg RBMFAE?)]
  [newbox (v RBMFAE?)]
  [setbox (bn RBMFAE?) (v RBMFAE?)]
  [openbox (v RBMFAE?)]
  [seqn (ex1 RBMFAE?) (ex2 RBMFAE?)]
  [setvar (name symbol?) (v RBMFAE?)] ; variable
  )
```

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

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

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

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

Parser

```
; parse : sexp -> RBMFAE
(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 v
    [(? symbol?) (id sexp)]
    [(list 'newbox v) (newbox (parse v))]
    [(list 'setbox bn v) (setbox (parse bn) (parse v))]
    [(list 'openbox v) (openbox (parse v))]
    [(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))]
    [(list 'setvar n v) (setvar n (parse v))] ; variable
    [(list 'refun (list p) b) (refun p (parse b))] ; call-by-
    [else (error 'parse "bad syntax: ~a" sexp)])
  )
```

Interpreter

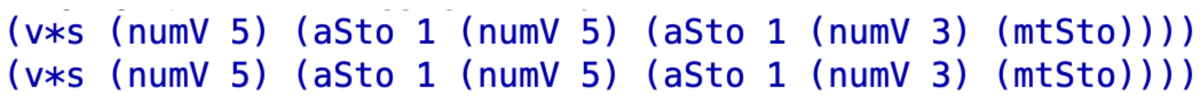
```
; Interpreter
(define (interp expr ds st)
  (type-case RBMFAE expr
    [num (n) (v*s (numV n) st)]
    [add (l r) (interp-two l r ds st (lambda (v1 v2 st1) (v*s
    [sub (l r) (interp-two l r ds st (lambda (v1 v2 st1) (v*s
    [id (s) (v*s (store-lookup (lookup s ds) st) st)]
    [fun (p b) (v*s (closureV p b ds) st)]
    [refun (p b) (v*s (refclosV p b ds) st)]
    [app (f a) (type-case Value*Store (interp f ds st)
                 [v*s (f-value f-store)
                      (type-case RBMFAE-Value f-value
                        [closureV (c-param c-body c-ds)
                                  (type-case Value*Store (int
                                    [v*s (a-value a-store)
                                         (local ([define new-
                                           (interp c-body
                                                   (aSub c-pa
                                                         new-
                                                         c-ds
                                                   (aSto new-
                                                         a-va
                                                         a-st
                        [refclosV (rc-param rc-body rc-ds)
                                  (local [(define address (lo
                                    (interp rc-body
                                            (aSub rc-param
                                                  address
                                                  rc-ds)
                                            f-store))]
                        [else (error interp "trying to apply
    [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
     [setvar (id val-expr) (local [(define a (lookup id ds))]
                             (type-case Value*Store (interp va
                               [v*s (val st)
                                    (v*s val
                                         (aSto a val st))]))])
     )
```

Supporting functions

```
; num-op: (number number ->number) -> (RBMFAE RBMFAE -> RBMFAE
(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 -> RBMFAE-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
; call-by-value
(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))]))

; 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)]]]))
```

```
; call by reference
(run '{with {a 3} {setvar a 5}} (mtSub) (mtSto))
(run '{with {a 3} {seqn {{refun {x} {setvar x 5}} a} a}} (mtS
```



```
(v*s (numV 5) (aSto 1 (numV 5) (aSto 1 (numV 3) (mtSto))))
(v*s (numV 5) (aSto 1 (numV 5) (aSto 1 (numV 3) (mtSto))))
```

```
; we want a to be 20 and b to be 10
(run '{with {swap {refun {x}
                       {refun {y}
                            {with {z x}
                                  {seqn {setvar x y}
```

```
                                                {setvar y z}}}}}}
             {with {a 10}
                   {with {b 20}
                         {seqn {{swap a} b}
                               b}}}} (mtSub) (mtSto))
```

```
(v*s
 (numV 10)
 (aSto
  3
  (numV 10)
  (aSto 2 (numV 20) (aSto 4 (numV 10) (aSto 3 (numV 20) (aSto 2 (numV 10) (aSto 1 (refclosV 'x (refun 'y
(app (fun 'z (seqn (setvar 'x (id 'y)) (setvar 'y (id 'z)))) (id 'x))) (mtSub)) (mtSto))))))))
```

```
(v*s (numV 10)
(aSto 3 (numV 10)
; {setvar y z}
; in the address of 'b', update the value to the value of 'a'
(aSto 2 (numV 20)
; {setvar x y}
; in the address of 'a', update the value to the value of 'b'
(aSto 4 (numV 10)
; {swap a}
(aSto 3 (numV 20)
; with {b 20}
; 'b' identifier
(aSto 2 (numV 10)
; with {a 10}
; 'a' identifier
(aSto 1 (refclosV 'x (refun 'y (app (fun 'z (seqn (setvar 'x
; swap function
```