ITP20005
Variables

Lecture21
JC

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.

It's a store-passing interpreter.

```
(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)))))

Variables

Boxes don't explain this example:

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

In a program like this, an identifier no longer stands for a value; instead, an identifier stands for a variable.

Implementing Variables

Option1:

```
(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))
```

Implementing Variables

Option1:

```
(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))
```

Option 2:

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

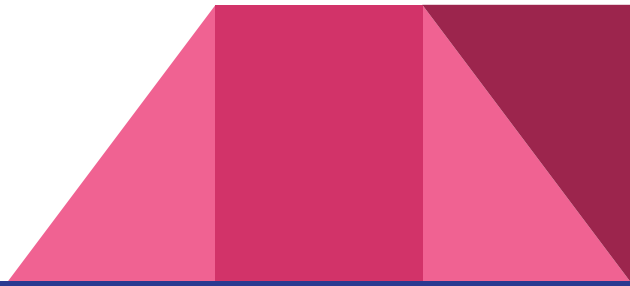
BMFAE = BFAE + Variables

<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>}

Examples:
{with {a 3} {setvar a 5}}
{with {a 3} {seqn {{fun {x} {setvar x 5}} a} a}}

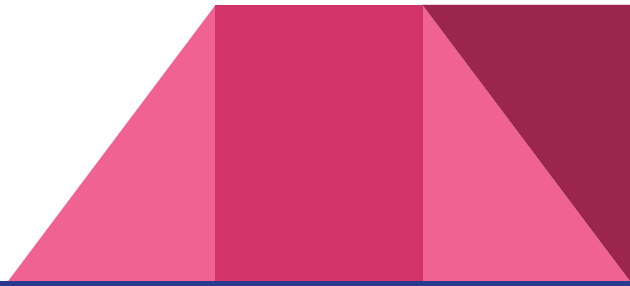
Implementing Variables

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



Implementing Variables

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



Implementing Variables

```
; interp : BMFAE DefrdSub Store -> Value*Store
(define (interp expr ds st)
   (type-case BMFAE expr
   ...
   [id (name) (v*s (...)
                   st)]
   ...)
```

Implementing Variables

```
; interp : BMFAE DefrdSub Store -> Value*Store
(define (interp expr ds st)
   (type-case BMFAE expr
   ...
   [id (name) (v*s (store-lookup (lookup name ds) st)
                   st)]
   ...)
```

Implementing Variables

```
;interp: BMFAE DefrdSub Store -> Value*Store
(define (interp expr 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-store)
                          [v*s (a-value a-store)
                               (local ([define new-address (malloc a-store)])
                                  (interp (closureV-body f-value)
                                          (aSub (closureV-param f-value)
                                                new-address
                                                (closureV-ds f-value))
                                          (aSto new-address
                                                a-value
                                                a-store)))])])]
   … )
```

Implementing Variables

```
; interp : BMFAE DefrdSub Store -> Value*Store
(define (interp expr ds st)
        ...
        [setvar (id val-expr)
                ... ]
 ...)
```

Implementing Variables

```
; interp : BMFAE DefrdSub Store -> Value*Store
(define (interp expr ds st)
        ...
        [setvar (id val-expr)
                … (type-case Value*Store (interp val-expr ds st)
                        [v*s (val st)
                                … ])]
 ...)
```

Implementing Variables

```
; interp : BMFAE DefrdSub Store -> Value*Store
(define (interp expr ds st)
        ...
        [setvar (id val-expr)
                … (type-case Value*Store (interp val-expr ds st)
                        [v*s (val st)
                                (v*s val
                                        (aSto …))])]
 ...)
```

Implementing Variables

```
; interp : BMFAE DefrdSub Store -> Value*Store
(define (interp expr ds st)
        ...
        [setvar (id val-expr)
                (local [(define a (lookup id ds))]
                    (type-case Value*Store (interp val-expr ds st)
                        [v*s (val st)
                                (v*s val
                                        (aSto …))]))]
 ...)
```

Implementing Variables

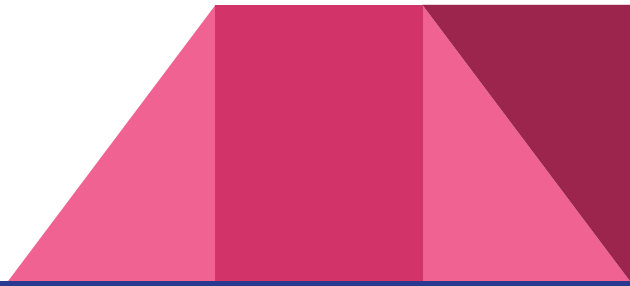
```
; interp : BMFAE DefrdSub Store -> Value*Store
(define (interp expr ds st)
        ...
        [setvar (id val-expr)
                (local [(define a (lookup id ds))]
                    (type-case Value*Store (interp val-expr ds st)
                        [v*s (val st)
                       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

address bounded with id

Run the following example code

(run '{with {a 3} {setvar a 5}} (mtSub) (mtSto))
⇒ ??

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



Run the following example code

(run '{with {a 3} {setvar a 5}} (mtSub) (mtSto))
⇒ (v*s (numV 5) (aSto 1 (numV 5) (aSto 1 (numV 3) (mtSto))))

(run '{with {a 3} {seqn {{fun {x} {setvar x 5}} a} a}} (mtSub) (mtSto))
⇒ (v*s **(numV 3)** (aSto 2 (numV 5) (aSto 2 (numV 3) **(aSto 1 (numV 3)** (mtSto)))))

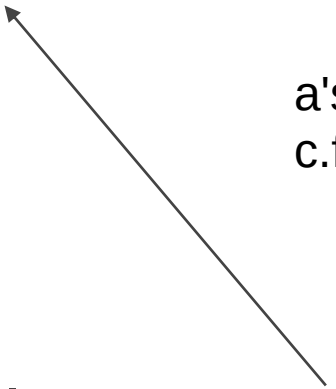
Implementing Variables

```
;interp: BMFAE DefrdSub Store -> Value*Store
(define (interp expr 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-store)
                          [v*s (a-value a-store)
                               (local ([define new-address (malloc a-store)])
                                  (interp (closureV-body f-value)
                                          (aSub (closureV-param f-value)
                                                new-address
                                                (closureV-ds f-value))
                                          (aSto new-address
                                                a-value
                                                a-store)))])])]
   … )
```

Run the following example code

(run '{with {a 3} {setvar a 5}} (mtSub) (mtSto))
⇒ (v*s (numV 5) (aSto 1 (numV 5) (aSto 1 (numV 3) (mtSto))))

(run '{with {a 3} {seqn {{fun {x} {setvar x 5}} a} a}} (mtSub) (mtSto))
⇒ (v*s **(numV 3)** (aSto 2 (numV 5) (aSto 2 (numV 3) **(aSto 1 (numV 3)** (mtSto)))))



a's address is still 1.
c.f. box's address?

Call-by-value

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

ITP20005
Variables (2)

Lecture22
JC

Run the following example code

(run '{with {a 3} {setvar a 5}} (mtSub) (mtSto))
⇒ (v*s (numV 5) (aSto 1 (numV 5) (aSto 1 (numV 3) (mtSto))))

(run '{with {a 3} {seqn {{fun {x} {setvar x 5}} a} a}} (mtSub) (mtSto))
⇒ (v*s **(numV 3)** (aSto 2 (numV 5) (aSto 2 (numV 3) **(aSto 1 (numV 3)** (mtSto)))))

Implementing Variables

```
;interp: BMFAE DefrdSub Store -> Value*Store
(define (interp expr 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-store)
                          [v*s (a-value a-store)
                               (local ([define new-address (malloc a-store)])
                                  (interp (closureV-body f-value)
                                          (aSub (closureV-param f-value)
                                                new-address
                                                (closureV-ds f-value))
                                          (aSto new-address
                                                a-value
                                                a-store)))])])]
   … )
```

Implementing Variables

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

Run the following example code

(run '{with {a 3} {setvar a 5}} (mtSub) (mtSto))
⇒ (v*s (numV 5) (aSto 1 (numV 5) (aSto 1 (numV 3) (mtSto))))

(run '{with {a 3} {seqn {{fun {x} {setvar x 5}} a} a}} (mtSub) (mtSto))
⇒ (v*s **(numV 3)** (aSto 2 (numV 5) (aSto 2 (numV 3) **(aSto 1 (numV 3)** (mtSto)))))

How about this??

(run '{with {a {newbox 3}} {seqn {{fun {x} {setbox x 5}} a} (openbox a)}} (mtSub) (mtSto))
⇒ ?

Run the following example code (2)

(run '{with {a 3} {setvar a 5}} (mtSub) (mtSto))
⇒ (v*s (numV 5) (aSto 1 (numV 5) (aSto 1 (numV 3) (mtSto))))

(run '{with {a 3} {seqn {{fun {x} {setvar x 5}} a} a}} (mtSub) (mtSto))
⇒ (v*s **(numV 3)** (aSto 2 (numV 5) (aSto 2 (numV 3) **(aSto 1 (numV 3)** (mtSto)))))

How about this??

(run '{with {a {newbox 3}} {seqn {{fun {x} {setbox x 5}} a} {openbox a}}}
(mtSub) (mtSto))
⇒ (v*s (numV 5) (aSto 1 (numV 5) (aSto 3 (boxV 1) (aSto 2 (boxV 1)
(aSto 1 (numV 3) (mtSto))))))

Run the following example code (3)

```
{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}}}}
```

⇒ a is 10. (Why? our interpreter is based on call-by-value)

Run the following example code (4)

```
(v*s
 (numV 10)  (aSto  5  (numV 10)  (aSto   4   (numV 20)
                  (aSto   6 (numV 10) (aSto 5 (numV 20)
                    (aSto 4 (numV 10) (aSto 3 (numV 20)
                      (aSto 2 (numV 10)
                        (aSto 1 (closureV 'x (fun 'y (app (fun 'z (seqn (setvar 'x
(id 'y)) (setvar 'y (id 'z)))) (id 'x))) (mtSub)) (mtSto)))))))))
```

Run the following example code (5)

```
{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}}}}
```

⇒ b is 20.

Run the following example code (6)

```
(v*s
 (numV 20)  (aSto  5  (numV 10)  (aSto   4   (numV 20)
                  (aSto   6 (numV 10) (aSto 5 (numV 20)
                    (aSto 4 (numV 10) (aSto 3 (numV 20)
                      (aSto 2 (numV 10)
                        (aSto 1 (closureV 'x (fun 'y (app (fun 'z (seqn (setvar 'x
(id 'y)) (setvar 'y (id 'z)))) (id 'x))) (mtSub)) (mtSto)))))))))
```

Variables and Function Calls (Racket example)

```
(define (swap x y)
   (local [(define z y)]
      (set! y x)
      (set! x z)))
(local [(define a 10)
        (define b 20)]
   (begin
      (swap a b)
      a))
```

Variables and Function Calls (Racket example)

```
(define (swap x y)
   (local [(define z y)]
      (set! y x)
      (set! x z)))
(local [(define a 10)
        (define b 20)]
   (begin
      (swap a b)
      a))
```

Result is 10; assignment in swap cannot affect a.

Variables and Function Calls (Racket example)

What if we wanted swap to change a?

```
(define (swap x y)
   (local [(define z y)]
      (set! y x)
      (set! x z)))
```

```
(local [(define a 10)
        (define b 20)]
   (begin
      (swap a b)
      a))
```

⇒

```
(local [(define a (box 10))
        (define b (box 20))]
   (begin
      (swap a b)
      (unbox a)))
```

This is called call-by-reference, as opposed to call-by-value

Variables and Function Calls (Racket example)

What if we wanted swap to change a?

```
(define (swap x y)
   (local [(define z y)]
      (set! y x)
      (set! x z)))
(local [(define a 10)
        (define b 20)]
   (begin
      (swap a b)
      a))
```

⇒

```
(define  (swap x y)
   (local [(define z (box (unbox y)))]
      (set-box! y (unbox x))
      (set-box! x (unbox z))))
(local [(define a (box 10))
        (define b (box 20))]
   (begin
      (swap a b)
      (unbox a)))
```

This is called call-by-reference, as opposed to call-by-value

Variables and Function Calls (current BFAE)

What if we wanted swap to change a?

```
{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}}}}
```

⇒

```
{with {swap {fun {x}
          {fun {y}
            {with {z x}
              {seqn {setbox x (openbox y)}
                    {setbox y (openbox z)}}}}}}
       {with {a (newbox 10)}
             {with {b (newbox 20)}
                   {seqn {{swap a} b}
                         (openbox a)}}}}
```

This is called call-by-reference, as opposed to call-by-value

Run the following example code

(run '{with {a 3} {setvar a 5}} (mtSub) (mtSto))
⇒ (v*s (numV 5) (aSto 1 (numV 5) (aSto 1 (numV 3) (mtSto))))

(run '{with {a 3} {seqn {{fun {x} {setvar x 5}} a} a}} (mtSub) (mtSto))
⇒ (v*s **(numV 5)** (aSto 1 (numV 5) (aSto 1 (numV 3) (mtSto))))

Call-by-reference

When a function is called, the value of the existing address of 'a' is stored with 5, which was mutated in the function.

Implementing Call-By-Reference

```
; interp : BMFAE DefrdSub Store -> Value*Store
(define (interp expr ds st)
        ...
        [app (fun-expr arg-expr)
             (if (id? arg-expr)
                 ; call-by-ref handling for an 'id' argument as a value:
                 ...
                 ; as before:
                 (type-case Value*Store (interp f ds st)
                   [v*s (f-value f-store)
                   (type-case Value*Store (interp a ds f-store)
                       [v*s (a-value a-store)
                           (local ([define new-address (malloc a-store)])
                            (interp (closureV-body f-value)
                                    (aSub (closureV-param f-value)
                                          new-address
                                          (closureV-ds f-value))
                                    (aSto new-address
                                          a-value
                                          a-store)))])]))] ...)
```

Implementing Call-By-Reference

```
; interp : BMFAE DefrdSub Store -> Value*Store
(define (interp expr ds st)
        ...
        [app (fun-expr arg-expr)
             (if (id? arg-expr)
                 ; call-by-ref handling for id arg:
                 (type-case Value*Store (interp fun-expr ds st)
                      [v*s (fun-val st1)
                           (local [(define a (lookup (id-name arg-expr) ds))]
                                (interp (closureV-body fun-val)
                                        (aSub (closureV-param fun-val)
                                              a
                                              (closureV-ds fun-val))
                                        st1))])
                 ; as before:
                 … )]
 ...)
```

Run the following example code

(run '{with {a 3} {setvar a 5}} (mtSub) (mtSto))
⇒ (v*s (numV 5) (aSto 1 (numV 5) (aSto 1 (numV 3) (mtSto))))

(run '{with {a 3} {seqn {{fun {x} {setvar x 5}} a} a}} (mtSub) (mtSto))
⇒ (v*s **(numV 5)** (aSto 1 (numV 5) (aSto 1 (numV 3) (mtSto))))

Call-by-reference

When a function is called, the value of the existing address of 'a' is stored with 5, which is mutated in the function.

Run the following example code

```
{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}}}}
```

⇒ a is 20.

Run the following example code

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

Run the following example code

```
{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}}}}
```

⇒ b is 20. What's wrong with this??????

Run the following example code

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

As we swap 'a' and 'b', 'a' must be 20 and 'b' must be 10. However, 'b' is not updated. Why?

Run the following example code

```
{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.

⇒ b is still 20.

Run the following example code

```
{with {swap {fun {x}
              {fun {y}
```

In our interpreter, 'with' expression is replaced into fun expression.

of x. So y, z will f y.

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.

⇒ b is still 20.

RBMFAE = BFAE + Variables + call-by-Reference

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

Adding 'refun' syntax

```
(define-type RBMFAE
    …
    [fun      (param symbol?) (body RBMFAE?)]
    [refun     (param symbol?) (body RBMFAE?)]
    …
  )
; parse : sexp -> RBMFAE
(define (parse sexp)
   (match sexp
        …
        [(list 'with (list i v) e)  (app (fun i (parse e)) (parse v))]
       …
        [(list 'fun (list p) b)     (fun p (parse b))]
        [(list 'refun (list p) b)   (refun p (parse b))]
        …
```

Adding 'refun' syntax

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

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

app based on call-by-value

```
;interp: RBMFAE DefrdSub Store -> Value*Store
(define (interp expr 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-store)
                        [v*s (a-value a-store)
                             (local ([define new-address (malloc a-store)])
                               (interp (closureV-body f-value)
                                       (aSub (closureV-param f-value)
                                             new-address
                                             (closureV-ds f-value))
                                       (aSto new-address
                                             a-value
                                             a-store)))])])]
   … )
```

Adding 'refun' syntax

```
;interp: RBMFAE DefrdSub Store -> Value*Store
(define (interp expr 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)
                                  ...
                                  (local ([define new-address (malloc a-store)])
                                                                          ... ]
                            [refclosV (rc-param rc-body rc-ds)
                                  ...
                                  (local ([define address (lookup (id-name a) ds)])
                                                                          ... ]
                            [else (error interp "trying to apply a number")]
                       )])]
   ...)
```

Topics we cover and schedule (tentative)

- Racket tutorials (L2,3, HW)
- Modeling languages (L4,5, HW
- Interpreting arithmetic (L5)
- Language principles
  - **Substitution** (L6, HW)
  - **Function** (L7)
  - **Deferring Substitution** (L8,L9)
  - **First-class Functions** (L10-12)
  - **Laziness** (L13, L14)
  - **Recursion** (L15, L16)
  - **Mutable data structures** (L17,18,19)
  - **Variables** (L20, L21)
  - Continuations
  - Garbage collection
  - Semantics
  - Type
- Guest Video Lecture

No class: October 2 (Fri, Chuseok), October 9 (Fri, Hangul day)
Online only class can be provided.

TODO

Read Chapter 17~20: Continuations

JC

jcnam@handong.edu

https://lifove.github.io

* Slides are from Prof. Sukyoung Ryu's PL class in 2018 Spring or created by JC based on the main text book.