Functional Programs

So far, the language that we've implemented is purely ***functional***

- A function produces the same result every time for the same arguments
- Also, lazy and eager results are the same

... except that eager evaluation might loop forever or raise an exception where the lazy version produces a result

Non-Functional Procedures

```
(define (f x)
  (+ x (read)))

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

(define f
  (local [(define b (box 0))]
    (lambda (x)
      (begin
        (set-box! b (+ x (unbox b)))
        (unbox b)))))
```

BCFAE = FAE + Boxes

```
<BCFAE>  ::=  <num>
           |  {+ <BCFAE> <BCFAE>}
           |  {- <BCFAE> <BCFAE>}
           |  <id>
           |  {fun {<id>} <BCFAE>}
           |  {<BCFAE> <BCFAE>}
           |  {if0 <BCFAE> <BCFAE> <BCFAE>}
           |  {newbox <BCFAE>}                   NEW
           |  {setbox <BCFAE> <BCFAE>}           NEW
           |  {openbox <BCFAE>}                  NEW
           |  {seqn <BCFAE> <BCFAE>}             NEW
```

```
{with {b {newbox 0}}
  {seqn
   {setbox b 10}
   {openbox b}}}      ⇒  10
```

Implementing Boxes with Boxes

```
(define-type BCFAE-Value
  [numV (n number?)]
  [closureV (param symbol?)
            (body BCFAE?)
            (sc SubCache?)]
  [boxV (container (box-of BCFAE?))])
```

Implementing Boxes with Boxes

```
; interp : BCFAE SubCache -> BCFAE-Value
(define (interp a-bcfae sc)
  (type-case RCFAE a-bcfae
    ...
    [newbox (val-expr)
            (boxV (box (interp val-expr sc)))]
    [setbox (box-expr val-expr)
            (set-box! (boxV-container
                       (interp box-expr sc))
                      (interp val-expr sc))]
    [openbox (box-expr)
             (unbox (boxV-container
                     (interp box-expr sc)))]))
```

But this doesn't explain anything about boxes!

Boxes and Memory

```
{with {b {newbox 7}}    ⇒    ...
   ...}
```

Memory:

Memory:

			7	

Boxes and Memory

```
... {setbox b 10}
...
```

Memory:

			7	

⇒

```
... {openbox b}
...
```

Memory:

			10	

The Store

We represent memory with a ***store***:

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

Memory:

			10	

```
(aSto 13 (numV 10)
      (mtSto))
```

Implementing Boxes without State

```
; interp : BCFAE SubCache Store -> Value*Store

(define-type BCFAE-Value
  [numV (n number?)]
  [closureV (param symbol?)
            (body BCFAE?)
            (sc SubCache?)]
  [boxV (address integer?)])

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

Implementing Boxes without State

```
; interp : BCFAE SubCache Store -> Value*Store
(define (interp expr sc st)
  ...
  [newbox (expr)
          (type-case Value*Store (interp expr sc st)
            [v*s (val st)
                 (local [(define a (malloc st))]
                   (v*s (boxV a)
                        (aSto a val st)))])]
  ...)

; malloc : Store -> integer
```

Implementing Boxes without State

```
; malloc : Store -> integer
(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))]))
```

Implementing Boxes without State

```
; interp : BCFAE SubCache Store -> Value*Store
(define (interp expr sc st)
  ...
  [openbox (bx-expr)
           (type-case Value*Store (interp bx-expr sc st)
             [v*s (bx-val st)
                  (v*s (store-lookup (boxV-address bx-val)
                                     st)
                       st)])]
  ...)
```

Implementing Boxes without State

```
; interp : BCFAE SubCache Store -> Value*Store
(define (interp expr sc st)
  ...
  [setbox (bx-expr val-expr)
          (type-case Value*Store (interp bx-expr sc st)
            [v*s (bx-val st2)
                 (type-case Value*Store (interp val-expr sc st2)
                   [v*s (val st3)
                        (v*s val
                             (aSto (boxV-address bx-val)
                                   val
                                   st3))])])]
  ...)
```

seqn, **add**, **sub**, and **app** will need the same sort of sequencing

Implementing Boxes without State

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

Implementing Boxes without State

```
; interp : BCFAE SubCache Store -> Value*Store
(define (interp expr sc st)
  ...
  [add (r l) (interp-two r l sc st
                         (lambda (v1 v2 st)
                           (v*s (num+ v1 v2) st)))]
  ...
  [seqn (a b) (interp-two a b sc st
                          (lambda (v1 v2 st)
                            (v*s v2 st)))]
  ...
  [setbox (bx-expr val-expr)
          (interp-two bx-expr val-expr sc st
                      (lambda (bx-val val st3)
                        (v*s val
                             (aSto (boxV-address bx-val)
                                   val
                                   st3))))]
 ...)
```

Variables

Boxes don't explain one of our earlier Scheme examples:

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

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

Implementing Variables

Option 1:

```
(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

BMCFAE = BCFAE + variables

```
<BMCFAE>  ::=  <num>
            |  {+ <BMCFAE> <BMCFAE>}
            |  {- <BMCFAE> <BMCFAE>}
            |  <id>
            |  {fun {<id>} <BMCFAE>}
            |  {<BMCFAE> <BMCFAE>}
            |  {if0 <BMCFAE> <BMCFAE> <BMCFAE>}
            |  {newbox <BMCFAE>}
            |  {setbox <BMCFAE> <BMCFAE>}
            |  {openbox <BMCFAE>}
            |  {seqn <BMCFAE> <BMCFAE>}
            |  {set <id> <BMCFAE>}                    NEW
```

Implementing Variables

```
(define-type SubCache
  [mtSub]
  [aSub (name symbol?)
        (address integer?)
        (sc SubCache?)])
```

Implementing Variables

```
; interp : BCFAE SubCache Store -> Value*Store
(define (interp expr sc st)
  ...
  [id (name) (v*s (store-lookup (lookup name sc) st)
                  st)]
  ...)
```

Implementing Variables

```
; interp : BCFAE SubCache Store -> Value*Store
(define (interp expr sc st)
  ...
  [app (fun-expr arg-expr)
       (interp-two fun-expr arg-expr sc st
                   (lambda (fun-val arg-val st)
                     (local [(define a (malloc st))]
                       (interp (closureV-body fun-val)
                               (aSub name
                                     a
                                     (closureV-sc fun-val))
                               (aSto a
                                     arg-val
                                     st)))))]
  ...)
```

Implementing Variables

```
; interp : BCFAE SubCache Store -> Value*Store
(define (interp expr sc st)
  ...
  [set (id val-expr)
       (local [(define a (lookup id sc))]
         (type-case Store*Value (interp val-expr sc st)
           [v*s (val st)
                (v*s val
                     (aSto a
                           val
                           st))]))]
  ...)
```

Variables and Function Calls

```
(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`**

Call-by-Reference

What if we wanted **swap** to change **a**?

```
(define (swap x y)      ⇒   (define (swap x y)
  (local [(define z y)]       (local [(define z (box (unbox y)))]
    (set! y x)                  (set-box! y (unbox x))
    (set! x z)))                (set-box! x (unbox z))))

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

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

Terminology alert: this “call-by-value” is orthogonal to the use in “call-by-value” vs. “call-by-name”

Implementing Call-by-Reference

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