

# 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
- Real programming languages do not behave this way

# Non-Functional Procedure

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





# Non-Functional Procedure

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

# Non-Functional Procedure, w/boxes

```
(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>}
| {newbox <BCFAE>} 
| {setbox <BCFAE> <BCFAE>} 
| {openbox <BCFAE>} 
| {seqn <BCFAE> <BCFAE>} 
```

```
{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?)
            (ds DefrdSub?)]
  [boxV (container (box-of BCFAE?))])
```

# Implementing Boxes with Boxes

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

# Implementing Boxes with Boxes

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

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	

*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 DefrdSub Store -> Value*Store
```

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

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

# Implementing Boxes without State

```
; interp : BCFAE DefrdSub Store -> Value*Store
(define (interp expr ds st)
  ...
  [newbox (expr)
    ... (interp expr ds st) ...]
  ...)
```

# Implementing Boxes without State

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

# Implementing Boxes without State

```
; interp : BCFAE DefrdSub Store -> Value*Store
(define (interp expr ds st)
  ...
  [newbox (expr)
    (type-case Value*Store (interp expr ds st)
      [v*s (val st)
        ... (malloc st)
        ...])]
  ...)

; malloc : Store -> integer
```

# Implementing Boxes without State

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

; malloc : Store -> integer
```



# Implementing Boxes without State

```
; interp : BCFAE DefrdSub Store -> Value*Store
(define (interp expr ds st)
  ...
  [newbox (expr)
    (type-case Value*Store (interp expr ds 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 DefrdSub Store -> Value*Store
(define (interp expr ds st)
  ...
  [openbox (bx-expr)
    (type-case Value*Store (interp bx-expr ds st)
      [v*s (bx-val st)
        ...]])]
  ...)
```

# Implementing Boxes without State

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

# Implementing Boxes without State

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

# Implementing Boxes without State

```
; interp : BCFAE DefrdSub Store -> Value*Store
(define (interp expr ds st)
  ...
  [setbox (bx-expr val-expr)
          ...]
  ...)
```

# Implementing Boxes without State

```
; interp : BCFAE DefrdSub Store -> Value*Store
(define (interp expr ds st)
  ...
  [setbox (bx-expr val-expr)
    ... (interp bx-expr ds st) ...
    ... (interp val-expr ds st)
    ...]
  ...)
```

# Implementing Boxes without State

```
; interp : BCFAE DefrdSub Store -> Value*Store
(define (interp expr ds st)
  ...
  [setbox (bx-expr val-expr)
    ... (interp bx-expr ds st) ...
    ... (interp val-expr ds st)
    ...]
  ...)
```

```
{with {q {newbox 10}}
  {setbox {seqn {setbox q 12} b}
    {openbox q}}}
```

should put 12 in b, not 10



# Implementing Boxes without State

```
; interp : BCFAE DefrdSub Store -> Value*Store
(define (interp expr ds st)
  ...
  [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 st2)
          [v*s (val st3)
            ...])])]
  ...)
```

# Implementing Boxes without State

```
; interp : BCFAE DefrdSub Store -> Value*Store
(define (interp expr ds st)
  ...
  [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 st2)
          [v*s (val st3)
            (v*s val
              (aSto (boxV-address bx-val)
                    val
                    st3))]]))]
  ...)
```

# Implementing Boxes without State

```
; interp : BCFAE DefrdSub Store -> Value*Store
(define (interp expr ds st)
  ...
  [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 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 DefrdSub Store
;             (Value Value Store -> Value*Store)
;             -> Value*Store)
(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)]))]))
```

# Implementing Boxes without State

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