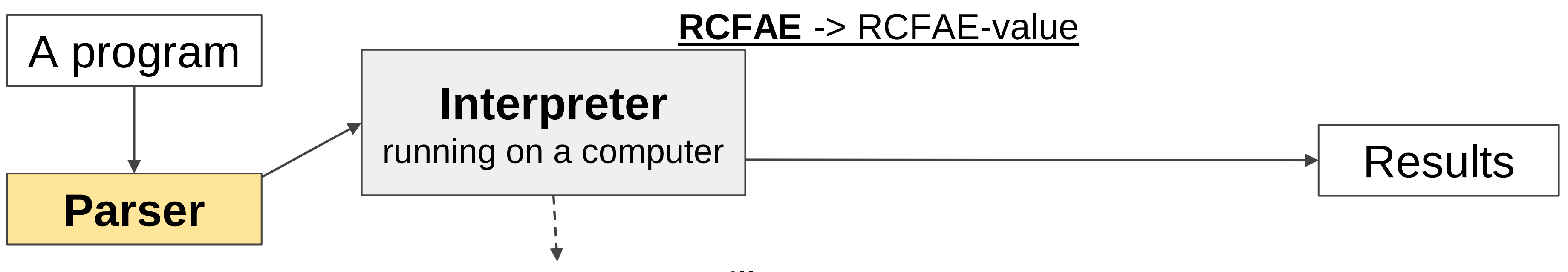
ITP20005
Mutable Data Structure

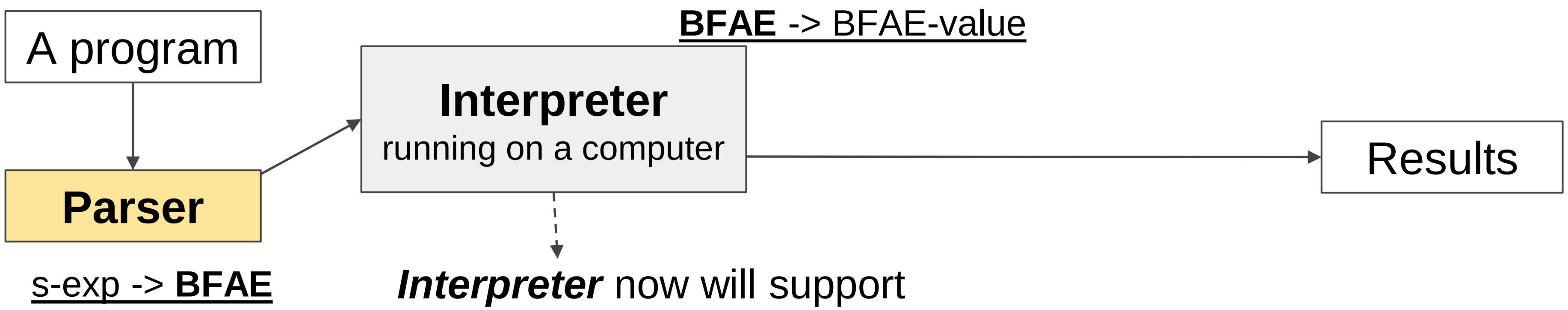
Lecture17
JC

Big Picture (modeling languages: substitution)



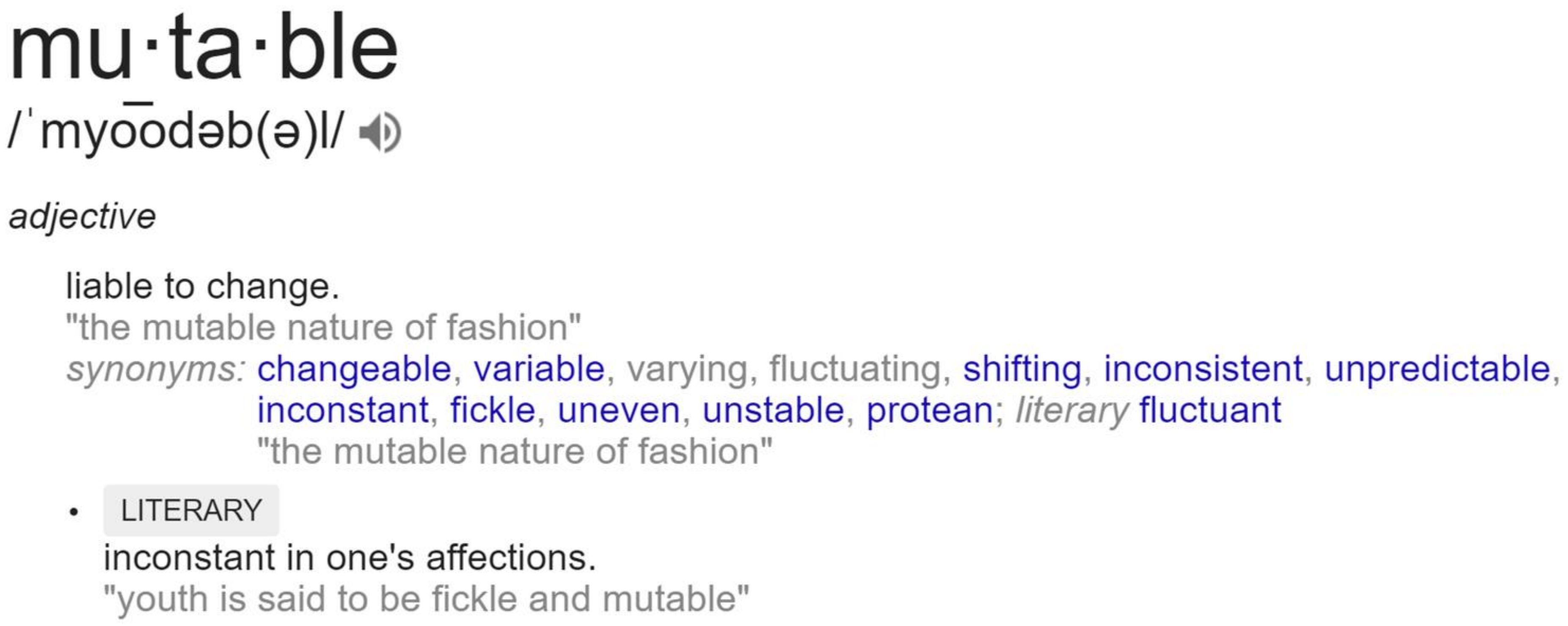
Interpreter now will support
(1) Substitution
(2) Function
(3) Deferring Substitution
(4) First-class Functions
(5) Laziness
(6) Recursion and its Implementation

Big Picture (modeling languages: substitution)



Interpreter now will support
(1) Substitution
(2) Function
(3) Deferring Substitution
(4) First-class Functions
(5) Laziness
(6) Recursion and its Implementation
(7) Mutable Data Structure

Why is the concept of 'mutation' important in PL??



mu·ta·ble

/ˈmyo͞odəb(ə)l/

adjective

liable to change.

"the mutable nature of fashion"

synonyms: changeable, variable, varying, fluctuating, shifting, inconsistent, unpredictable, inconstant, fickle, uneven, unstable, protean; *literary* fluctuant

"the mutable nature of fashion"

- LITERARY

  inconstant in one's affections.

  "youth is said to be fickle and mutable"

The mutable nature of data!

Any examples?

The mutable nature of data!

$\Rightarrow$ State

The mutable nature of data!

vs.

State makes it harder to reason about programs.

Functional Programs

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

- A function produces the same results every time for the same arguments.

However, "real" programming languages usually do not behave this way.

Non-Functional Procedure

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

(f 5)

(define g
  (local [(define b (box 0))]
    (lambda (x)
      (begin
        (set-box! b (+ x (unbox b)))
        (unbox b)))))
(g 5)
(g 5)


(define counter 0)
(define (h x)
  (begin
    (set! counter (+ x counter))
    counter))
(h 2)
(h 2)
```

Functional Programs

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

- A function produces the same results every time for the same arguments.

However, "real" programming languages usually do not behave this way.

$\Rightarrow$ Something can be changed!!!
Mutable!

Box

- A data structure that can hold any type of a single value.
- It is also mutable!!
    - In DrRacket, we could use the 'set-box!' operator to change a value in the box!

Let's improve our language to support mutable data structure!

But…
Why do we start from mutable data structure?
What about variables which seem mutable too…

But…
Why do we start from mutable data structure?
What about variables which seem mutable too…

(Variable is the next topic. To support variables in our language, we need to provide 'mutation' in our language. Mutable data structure is *a preliminary step toward supporting variables* in our language. We implement a simple mutable data structure, box in this lecture!)

ITP20005

Mutable Data Structure

Lecture18

JC

BFAE = FAE + Boxes

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

BFAE = FAE + Boxes

```
<BFAE> ::= <num>
         | {+ <BFAE> <BFAE>}
         | {- <BFAE> <BFAE>}
         | <id>
         | {fun {<id>} <BFAE>}
         | {<BFAE> <BFAE>}
         | {newbox <BFAE>}             ⇒ Define/initialize a box
         | {setbox <BFAE> <BFAE>}      ⇒ Update a box (Do mutation)
         | {openbox <BFAE>}            ⇒ Extract a value from a box
         | {seqn <BFAE> <BFAE>}
```

BFAE = FAE + Boxes

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

⇒ <id> is bound with values including number, function, or box.

Implementing Boxes with Boxes

```
(define-type BFAE-Value
  [numV       (n number?)]
  [closureV   (param symbol?)
              (body BFAE?)
              (ds DefrdSub?)]
  [boxV (container (box/c BFAE-Value?))]])
```

Implementing Boxes with Boxes

```
; interp : BFAE DefrdSub -> BFAE-Value
(define (interp bfae ds)
    (type-case BFAE bfae
    ...
    [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!

We want to implement box operations from scratch!

BFAE = FAE + Boxes

```
<BFAE> ::= <num>
         | {+ <BFAE> <BFAE>}
         | {- <BFAE> <BFAE>}
         | <id>
         | {fun {<id>} <BFAE>}
         | {<BFAE> <BFAE>}
         | {newbox <BFAE>}              ⇒ Define/initialize a box
         | {setbox <BFAE> <BFAE>}       ⇒ Update a box (Make something mutable)
         | {openbox <BFAE>}             ⇒ Extract a value from a box
         | {seqn <BFAE> <BFAE>}         ⇒ Run two expressions sequentially
```

BFAE = FAE + Boxes

```
<BFAE> ::= <num>
         | {+ <BFAE> <BFAE>}
         | {- <BFAE> <BFAE>}
         | <id>
         | {fun {<id>} <BFAE>}
         | {<BFAE> <BFAE>}
         | {newbox <BFAE>}              ⇒ Define/initialize a box
         | {setbox <BFAE> <BFAE>}       ⇒ Update a box (Make something mutable)
         | {openbox <BFAE>}             ⇒ Extract a value from a box
         | {seqn <BFAE> <BFAE>}         ⇒ Run two expressions sequentially
```

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

BFAE = FAE + Boxes

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

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

Abstract Syntax for 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?)]
 )
```

Examples

```
{with {b {newbox 0}}
   {seqn {setbox b {+ 1 {openbox b}}}
            {openbox b}}}
```

$\Rightarrow$??

Examples

```
{with {b {newbox 0}}
   {seqn {setbox b {+ 1 {openbox b}}} ; mutation on b by setbox
            {openbox b}}}
```

$\Rightarrow$ 1

(Mutation in the first operation in the sequence has *an effect* in the output of the second)

Abstract Syntax for 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?)]
 )
```

Example

```
{with {b {newbox 0}}
   {seqn {setbox b {+ 1 {openbox b}}} ; mutation on b by setbox
            {openbox b}}}
```

What if we implement our interpreter like this for seqn?

```
; interp : BFAE DefrdSub -> BFAE-Value
(define (interp bfae ds)
   (type-case BFAE bfae
      …
      [seqn (e1 e2)
         (interp e1 ds)
         (interp e2 ds)
      …
```

Any effect on e2 from e1????

Example

```
{with {b {newbox 0}}
   {seqn {setbox b {+ 1 {openbox b}}} ; mutation on b by setbox
            {openbox b}}}
```

What if we implement our interpreter like this for seqn?

```
; interp : BFAE DefrdSub -> BFAE-Value
(define (interp bfae ds)
   (type-case BFAE bfae
      …
      [seqn (e1 e2)
         (interp e1 ds)
         (interp e2 ds)
      …
```

Any effect on e2 from e1????

NO!

Example

```
{with {b {newbox 0}}
   {seqn {setbox b {+ 1 {openbox b}}} ; mutation on b by setbox
            {openbox b}}}
```

What if we implement our interpreter like this for seqn?

```
; interp : BFAE DefrdSub -> BFAE-Value
(define (interp bfae ds)
   (type-case BFAE bfae
      …
      [seqn (e1 e2)
         (interp e1 ds)
         (interp e2 ds)
      …
```

Any effect on e2 from e1????
NO!

We need more complex logic to support this sequence.

How can we make e1 affect e2?????

Example

```
{with {b {newbox 0}}
   {seqn {setbox b {+ 1 {openbox b}}} ; mutation on b by setbox
            {openbox b}}}
```

What if we implement our interpreter like this for seqn?

```
; interp : BFAE DefrdSub -> BFAE-Value
(define (interp bfae ds)
   (type-case BFAE bfae
      …
      [seqn (e1 e2)                    ???
         (interp e1 ds)
         (interp e2 ds)
      …
```

Example

```
{with {b {newbox 0}}
   {seqn {setbox b {+ 1 {openbox b}}} ; mutation on b by setbox
          {openbox b}}}
```

What if we implement our interpreter like this for seqn?

```
; interp : BFAE DefrdSub -> BFAE-Value
(define (interp bfae ds)
   (type-case BFAE bfae
      …
      [seqn (e1 e2)
         (interp e1 ds)
         (interp e2 ds)
      …
```

Our interpreter needs to return both the value of e1 and the updated ds??

Think about this example!

```
{with {a {newbox 1}}
  {with {f {fun {x} {+ x {openbox a}}}}
    {seqn
      {setbox a 2} ; update 'a' like an assumption that 'ds' could be updated.
      {f 5}}}}
```

- 'a' is bound to 1 (static scope)
- Contradiction
  - Even though 'a' becomes '2' by setbox, {f 5} will be 6 as our language is based on static scope! But it must be 7 as we mutated 'a' in 'seqn'
  - So the box value has been changed, but this looks like dynamic scope. Compare the code below...

    c.f.
    ```
    {with {x 3}
          {with {f {fun {y} {+ x y}}}
            {with {x 5}
              {f 10}}}}
    ```

Example

```
{with {b {newbox 0}}
   {seqn {setbox b {+ 1 {openbox b}}} ; mutation on b by setbox
         {openbox b}}}
```

What if we implement our interpreter like this for seqn?

```
; interp : BFAE DefrdSub -> BFAE-Value
(define (interp bfae ds)
   (type-case BFAE bfae
      …
      [seqn (e1 e2)
         (interp e1 ds)
         (interp e2 ds)
      …
```

Our interpreter needs to return both the value of e1 and the updated ds??

This idea is not enough.
How can we return both??

How can we make our language to support mutation for seqn?????

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

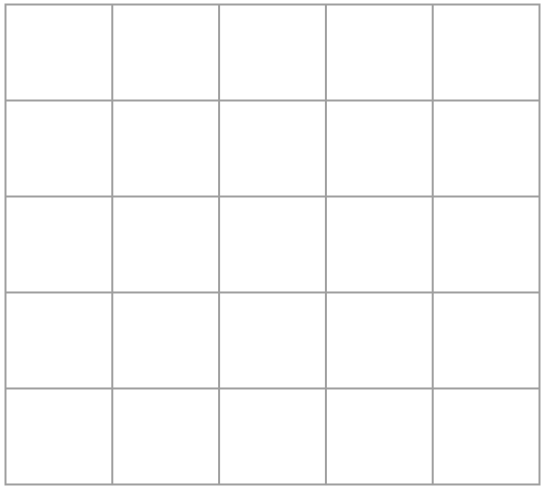
Idea

- We need two repositories (caches)
  - One for keeping a memory address value of a box for static scope
  - Another for tracking dynamic changes of boxes.
    ⇒ Let's call this cache as 'store'

Boxes and Memory

```
{with {b {newbox 7}}
   … }
```

Memory:



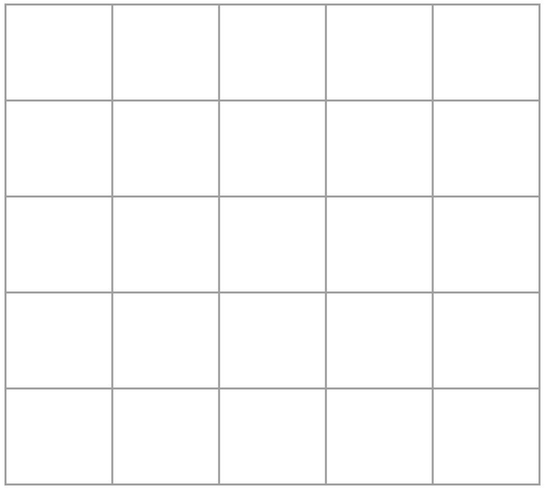
Boxes and Memory

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

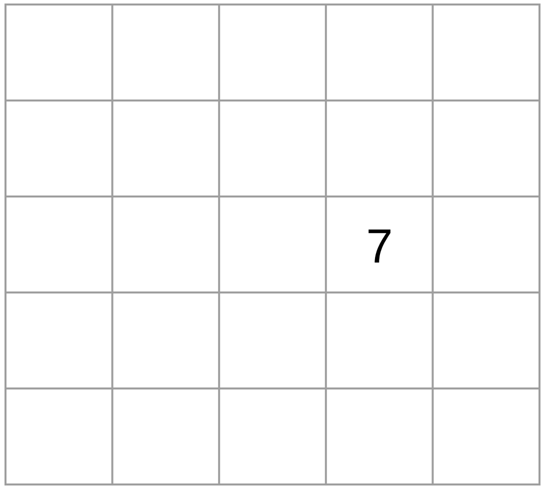
$\Rightarrow$

...

Memory:



Memory:



Boxes and Memory

```
… {setbox b 10}
…
```

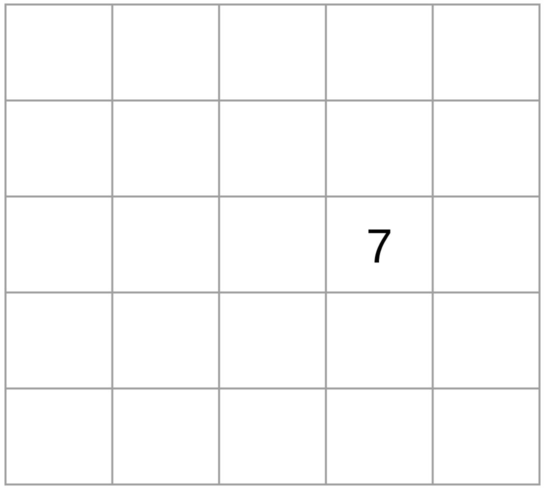
Memory:

			7	

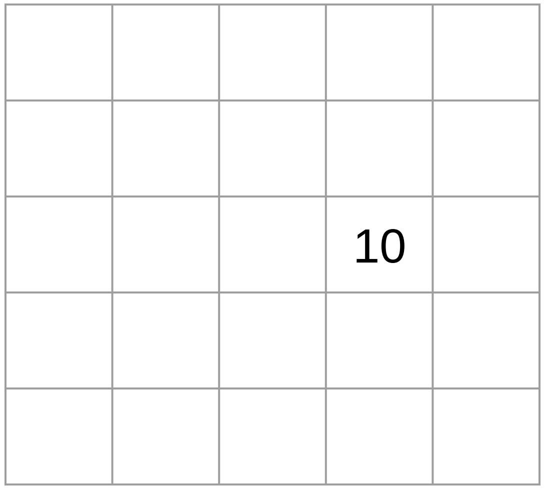
Boxes and Memory

```
… {setbox b 10}        ⇒        …
…
```

Memory:



Memory:

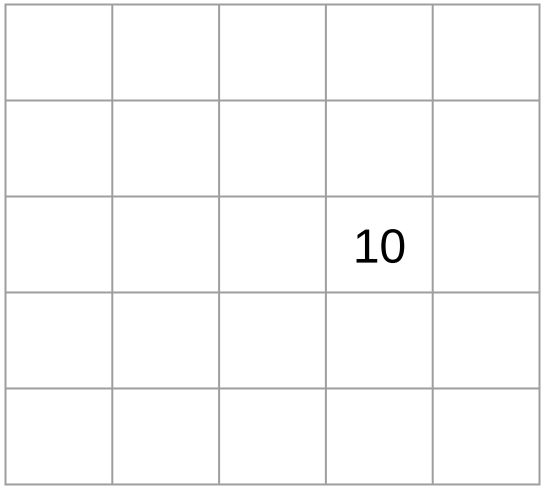


Boxes and Memory

We present memory with a store:

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

Memory



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

Idea

- 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?)])
```

Idea

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

- 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?)])
```

Idea

- We need two repositories (caches)
  - One for keeping a memory address value of a box for static scope
  - Another for maintaining dynamic changes of boxes.
    ⇒ Let's call this cache as 'store'
- Return both

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

⇒ (v*s (boxV 13) (aSto 13 (numV 10) … (mtSto)))
⇒ (v*s (numV 10) (aSto 13 (numV 10) … (mtSto)))

Idea

- We need two repositories (caches)
  - One for keeping a memory address as a value of a box for static scope
  - Another for maintaining dynamic changes of boxes.
    ⇒ Let's call this cache as 'store'
- Return both

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

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

⇒ (v*s (boxV 13) (aSto 13 (numV 10) … (mtSto)))
⇒ (v*s (numV 10) (aSto 13 (numV 10) … (mtSto)))

Implementing Boxes

```
;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?)])
```

ITP20005

Mutable Data Structure

Lecture19
JC

BFAE = FAE + Boxes

```
<BFAE> ::= <num>
         | {+ <BFAE> <BFAE>}
         | {- <BFAE> <BFAE>}
         | <id>
         | {fun {<id>} <BFAE>}
         | {<BFAE> <BFAE>}
         | {newbox <BFAE>}                ⇒ Define/initialize a box
         | {setbox <BFAE> <BFAE>}         ⇒ Update a box (Make something mutable)
         | {openbox <BFAE>}               ⇒ Extract a value from a box
         | {seqn <BFAE> <BFAE>}           ⇒ Run two expressions sequentially
```

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

How can we make our language to support mutation for seqn?????

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

Idea

- We need two repositories (caches)
    - One for keeping a memory address as a value of a box for static scope
    - Another for maintaining dynamic changes of boxes.
      ⇒ Let's call this cache as 'store'
- Return both

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

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

⇒ (v*s (boxV 13) (aSto 13 (numV 10) … (mtSto)))
⇒ (v*s (numV10) (aSto 13 (numV 10) … (mtSto)))

Idea

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

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.

- F

{with

10}

{openbox b}}}

{openbox b} }}

{newbox 7}}

⇒ (v*s (boxV 13) (aSto 13 (numV 10) … (mtSto)))

⇒ (v*s (numV10) (aSto 13 (numV 10) … (mtSto)))

Implementing Boxes for our BFAE language without State.

(What does this mean?)

C.f) Implementing Boxes with Boxes (State)

```
; interp : BFAE DefrdSub -> BFAE-Value
(define (interp bfae ds)
   (type-case BFAE bfae
   ...
   [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!

We want to implement box operations from scratch!

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?)])
```

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?)])
```

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

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?)])
```

```
  7
  {+ 7 6}
  {with {b {newbox 7}}
          {seqn {setbox b 10}
                  {openbox b}}}
```

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?)])
```

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

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?)])

  7                  ⇒ (v*s (numV 7) (mtSto))
  {+ 7 6}            ⇒ (v*s (numV 13) (mtSto))
  {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)))))
```

Idea

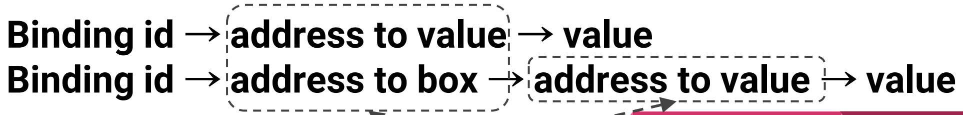
- We need two repositories (caches)
  - One for keeping a memory address value of a box for static scope

```
(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?)])
```


Binding id → address to value → value
Binding id → address to box → address to value → value

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?)])
```

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

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

(1)

Address assigned to 'b'

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

Implementing Boxes without State

```
;interp: BFAE DefrdSub Store -> Value*Store
(define (interp expr ds st) ...
   [num (n)     (v*s (numV n) st)]
   ...
   ... )
```

```
7                    ⇒ (v*s (numV 7) (mtSto))
{+ 7 6}              ⇒ (v*s (numV 13) (mtSto))
{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)))))
```

Implementing Boxes without State

```
;interp: BFAE DefrdSub Store -> Value*Store
(define (interp expr ds st) ...
   [num (n)     (v*s (numV n) st)]
   [add  (l r)    … (num+ (interp l ds st) (interp r ds st)...)]
   … )
```

```
  {+ 7 6}          ⇒ (v*s (numV 13) (mtSto))
```

Implementing Boxes without State

```
;interp: BFAE DefrdSub Store -> Value*Store
(define (interp expr ds st) ...
   [num (n)     (v*s (numV n) st)]
   [add  (l r)   (v*s (num+ (interp l ds st) (interp r ds st)))]
   ... )
```

$$\underbrace{\texttt{(interp l ds st)}}_{\texttt{(v*s }\textbf{(numV 7)}\texttt{ (mtSto))}} \quad \underbrace{\texttt{(interp r ds st)}}_{\texttt{(v*s }\textbf{(numV 6)}\texttt{ (mtSto))}}$$

{+ 7 6}   ⇒ (v*s (numV 13) (mtSto))

Implementing Boxes without State

```
;interp: BFAE DefrdSub Store -> Value*Store
(define (interp expr ds st) ...
   [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-store)
                              [v*s (r-value r-store) (num+ l-value r-value)
                              ... ])])]
                                                   (numV 7)  (numV 6)
   ... )
```

```
  {+ 7 6}          ⇒ (v*s (numV 13) (mtSto))
```

Implementing Boxes without State

```
;interp: BFAE DefrdSub Store -> Value*Store
(define (interp expr ds st) ...
   [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-store)
                             [v*s (r-value r-store) (num+ l-value r-value)
                             ... )])])]
   ... )
```

(numV 7) (numV 6)

```
{+ 7 6}          ⇒ (v*s (numV 13) (mtSto))
```

* The sub branch can be updated in a similar way.

Implementing Boxes without State

```
;interp: BFAE DefrdSub Store -> Value*Store
(define (interp expr ds st) ...
    [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-store)
                                  [v*s (r-value r-store)
                                         (v*s        (num+ l-value r-value)
                                                      r-store)])])]
                                                                (numV 7)  (numV 6)
    ... )
  {+ 7 6}          ⇒ (v*s (numV 13) (mtSto))
```

* The sub branch can be updated in a similar way.

Q&A

- Can we use l-store rather than r-store for the store in the 'add' branch?? NO

```
;interp: BFAE DefrdSub Store -> Value*Store
(define (interp expr ds st)
   [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-store)
                              [v*s (r-value r-store)
                                     (v*s         (num+ l-value r-value)
                                                   r-store)])])]
```

```
(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 12) (aSto 1 (numV 7) (aSto 2 (boxV 1) (aSto 1 (numV 10) (mtSto))))) ; when using l-store
(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))))))))
```

ITP20005
Mutable Data Structure

Lecture20
JC

Implementing Boxes without State

```
;interp: BFAE DefrdSub Store -> Value*Store
(define (interp expr ds st)
   [num (n)     (v*s (numV n) st)]
   [id     (s)     ...(lookup s ds)...]
   ... )
```

```
(run '{with {a 7} a} (mtSub) (mtSto))
⇒ (v*s (numV 7) (aSto 1 (numV 7) (mtSto)))
```

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?)])
```

```
                                                  {{fun {b}
  7                  ⇒ (v*s (numV 7) (mtSto))            {seqn {setbox b 10}
  {+ 7 6}            ⇒ (v*s (numV 13) (mtSto))                 {openbox b} }}
  {with {b {newbox 7}}                        (1)         {newbox 7}}
        {seqn {setbox b 10}
                  {openbox b}}}  (2)
                                         Address assigned to 'b'
  ⇒ (v*s (numV 10) (aSto 1 (numV 10) (aSto 2 (boxV 1) (aSto 1 (numV 7) (mtSto)))))
```

Recall the idea

- We need two repositories (caches)
  - One for keeping a memory address as a 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?)])
```

Implementing Boxes without State

```
;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))]))
```

```
(run '{with {a 7} a} (mtSub) (mtSto))
⇒ (v*s (numV 7) (aSto 1 (numV 7) (mtSto)))
```

Implementing Boxes without State

```
;interp: BFAE DefrdSub Store -> Value*Store
(define (interp expr ds st)
   [num (n)     (v*s (numV n) st)]
   [id      (s)     (v*s … (lookup s ds) … st)]
   … )
```

```
(run '{with {a 7} a} (mtSub) (mtSto))
⇒ (v*s (numV 7) (aSto 1 (numV 7) (mtSto)))
```

Implementing Boxes without State

$$\texttt{(v*s (store-lookup } \underbrace{\texttt{(lookup s ds)}}_{\text{Return the address of the binding id, a.}} \texttt{ st) st)]}$$

```
;interp: BFAE DefrdSub Store -> Value*Store
(define (interp expr ds st)
   [num (n)     (v*s (numV n) st)]
   [id      (s)     (v*s (store-lookup (lookup s ds) st) st)]
   ... )
```

Return the address of the binding id, a.

```
(run '{with {a 7} a} (mtSub) (mtSto))
⇒ (v*s (numV 7) (aSto 1 (numV 7) (mtSto)))
```

Implementing Boxes without State

```
;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))]))
```

Deferred substitution cache:
(aSub 'a **1** (mtSub))

```
(run '{with {a 7} a} (mtSub) (mtSto))
⇒ (v*s (numV 7) (aSto 1 (numV 7) (mtSto)))
```

Implementing Boxes without State

```
;interp: BFAE DefrdSub Store -> Value*Store
(define (interp expr ds st)
   [num (n)     (v*s (numV n) st)]
   [id     (s)     (v*s (store-lookup (lookup s ds) st) st)]
   … )
```

```
(run '{with {a 7} a} (mtSub) (mtSto))
⇒ (v*s (numV 7) (aSto 1 (numV 7) (mtSto)))
```

FAE Interpreter with Deferred Substitution

```
; interp: FAE DefrdSub -> FAE-Value
(define (interp fae ds)
   (type-case FAE fae
      …
      [fun     (p b)  (closureV p b ds)]
      …
```

Implementing Boxes without State

```
;interp: BFAE DefrdSub Store -> Value*Store
(define (interp expr ds st)
    …
    [fun (p b)   (v*s (closureV p b ds) st)]
    [app (f a)  ...
    … )
```

FAE Interpreter with Deferred Substitution

```
; interp: FAE DefrdSub -> FAE-Value
(define (interp fae ds)
   (type-case FAE fae
      …
      [fun     (p b)  (closureV p b ds)]
      [app     (f a)   (local [(define f-val (interp f ds))
                                (define a-val (interp a ds))]
                             (interp (closureV-body f-val)
                                     (aSub (closureV-param f-val)
                                           a-val
                                           (closureV-ds f-val))))]))
```

Implementing Boxes without State

```
;interp: BFAE DefrdSub Store -> Value*Store
(define (interp expr ds st)
   …
   [fun (p b)   (v*s (closureV p b ds) st)]
   [app (f a)  (... (interp f ds st)
                    ... (interp a ds f-store)...
                           ...
                                (interp (closureV-body f-value)
                                        (aSub (closureV-param f-value)
                                              new-address
                                              (closureV-ds f-value))
                                        (aSto ...]
   … )
```

Recall: how we change 'add' branch

```
;interp: BFAE DefrdSub Store -> Value*Store
(define (interp expr ds st)
   [num (n)     (v*s (numV n) st)]
   [add  (l r)     … (num+ (interp l ds st) (interp r ds st)...)]
   … )
```

```
{+ 7 6}          ⇒ (v*s (numV 13) (mtSto))
```

Recall: how we change 'add' branch

```
;interp: BFAE DefrdSub Store -> Value*Store
(define (interp expr ds st)
   [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-store)
                                 [v*s (r-value r-store)
                                     (v*s       (num+ l-value r-value)
                                                 r-store)])])]
                                                          (numV 7)  (numV 6)
   ... )
  {+ 7 6}          ⇒ (v*s (numV 13) (mtSto))
```

* The sub branch can be updated in a similar way.

FAE Interpreter with Deferred Substitution

```
; interp: FAE DefrdSub -> FAE-Value
(define (interp fae ds)
   (type-case FAE fae
      …
      [fun     (p b)  (closureV p b ds)]
      [app     (f a)   (local [(define f-val (interp f ds))
                               (define a-val (interp a ds))]
                            (interp (closureV-body f-val)
                                     (aSub (closureV-param f-val)
                                           a-val
                                           (closureV-ds f-val))))]))
```

Implementing Boxes without State

```
;interp: BFAE DefrdSub Store -> Value*Store
(define (interp expr ds 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-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 Boxes without State

```
;interp: BFAE DefrdSub Store -> Value*Store
(define (interp expr ds st)
    …
    [newbox (val)
                    … (interp val ds st) …]
    … )
```

Implementing Boxes without State

```
;interp: BFAE DefrdSub Store -> Value*Store
(define (interp expr ds st)
    …
    [newbox (val)
                (type-case Value*Store (interp val ds st)
                    [v*s   (vl st1)
                            … ])]
    … )
```

```
{newbox {+ 2 5}}
⇒
(v*s (boxV 1) (aSto 1 (numV 7) (mtSto)))   ; note that this is not the final result.
```

$\uparrow$

address

Implementing Boxes without State

```
;interp: BFAE DefrdSub Store -> Value*Store
(define (interp expr ds st)
   …
   [newbox (val)
             (type-case Value*Store (interp val ds st)
                [v*s   (vl st1)  <- - - - - - - - - - -  The updated store after interpreting val
                       … (malloc st1)
                       … ])]
   … )

;malloc : Store -> Integer
```

Implementing Boxes without State

```
;interp: BFAE DefrdSub Store -> Value*Store
(define (interp expr ds st)
   …
   [newbox (val)
               (type-case Value*Store (interp val ds st)
                  [v*s  (vl st1)
                        (local [(define a (malloc st1))]
                        … (boxV a)
                        … (aSto a vl st1) … )])]
   … )

;malloc : Store -> Integer
```

Implementing Boxes without State

```
;interp: BFAE DefrdSub Store -> Value*Store
(define (interp expr ds st)
   …
   [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)))])]
   … )
;malloc : Store -> Integer
;purpose: to allocate memory for a new box.
```

{newbox {+ 2 5}}

$\Rightarrow$

(v*s (boxV 1) (aSto 1 (numV 7) (mtSto)))   ; note that this is not a final evaluation result.

Implementing Boxes without State

```
; malloc : Store -> Integer
(define (malloc st)
   (+ 1 (max-address st)))     ; what will be the first address?

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

* We are allocating memory sequentially….

C.f. Implementing Boxes with State

```
; malloc : Store -> Integer
(define malloc
   (local ([define max-address (box -1)])
            (lambda (store)
               (begin
                  (set-box! max-address (+ 1 (unbox max-address)))
                  (unbox max-address)))))
```

Implementing Boxes without State

```
;interp: BFAE 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 st1)
                         …]))
   … )
```

```
{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)))))

Implementing Boxes without State

```
;interp: BFAE 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 st1)
                                … (boxV-address bx-val) … ]))
    … )
```

```
{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)))))

Implementing Boxes without State

```
;interp: BFAE 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 st1)
                          (v*s (store-lookup (boxV-address bx-val)
                                                        st1)
                                  st1)])]
   … )
```

```
{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)))))

Implementing Boxes without State

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

Implementing Boxes without State

```
;interp: BFAE 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} q}
                                        {openbox q}}}
```

q is 10 or 12?

Implementing Boxes without State

```
;interp: BFAE 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} q}
                                          {openbox q}}}
```

should put 12 in q, not 10

Implementing Boxes without State

```
;interp: BFAE 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: BFAE 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 need ***the same sort of sequencing***. two type-cases!

Implementing Boxes without State

```
; interp-two : BFAE BFAE 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: BFAE DefrdSub Store -> Value*Store
(define (interp expr ds st)
   [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-store)
                                  [v*s (r-value r-store)
                                                            (v*s (num+ l-value r-value)
                                                                   r-store)])])]
  …

⇒ [add    (l r)  (interp-two l r ds st
                               (lambda (v1 v2 st1) (v*s (num+ v1 v2) st1)))]
```

Implementing Boxes without State

```
;interp: BFAE DefrdSub Store -> Value*Store
(define (interp expr ds st)
    …
    [seqn (a b)  (type-case Value*Store (interp a ds st)
                          [v*s (a-value a-store)
                                 (interp b ds a-store)])]
   …
```

⇒ [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 can still apply interp-two for the seqn branch. We are just not using v1 but only v2.

Implementing Boxes without State

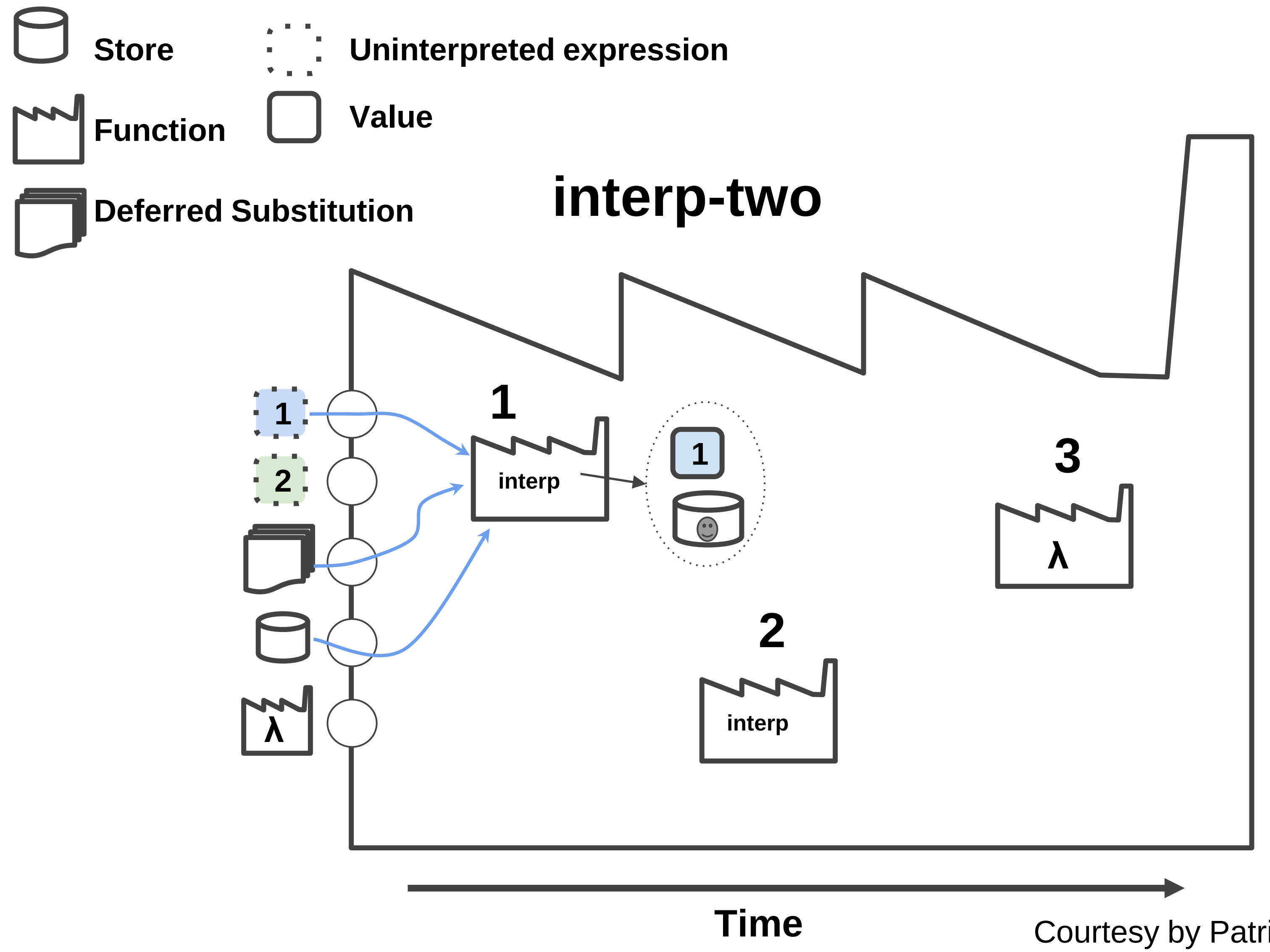
```
; interp-two : BFAE BFAE DefrdSub Store
;                (Value Value Store -> Value*Store)
;                -> Value*Store
(define (interp-two expr1 expr2 ds st handle)
  (type-case Value*Store (interp expr1 ds st)
    [v*s (val1 st2)
      (type-case Value*Store (interp expr2 ds st2)
        [v*s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

```
(lambda (v1 v2 st1) (v*s v2 st1))
```

Implementing Boxes without State

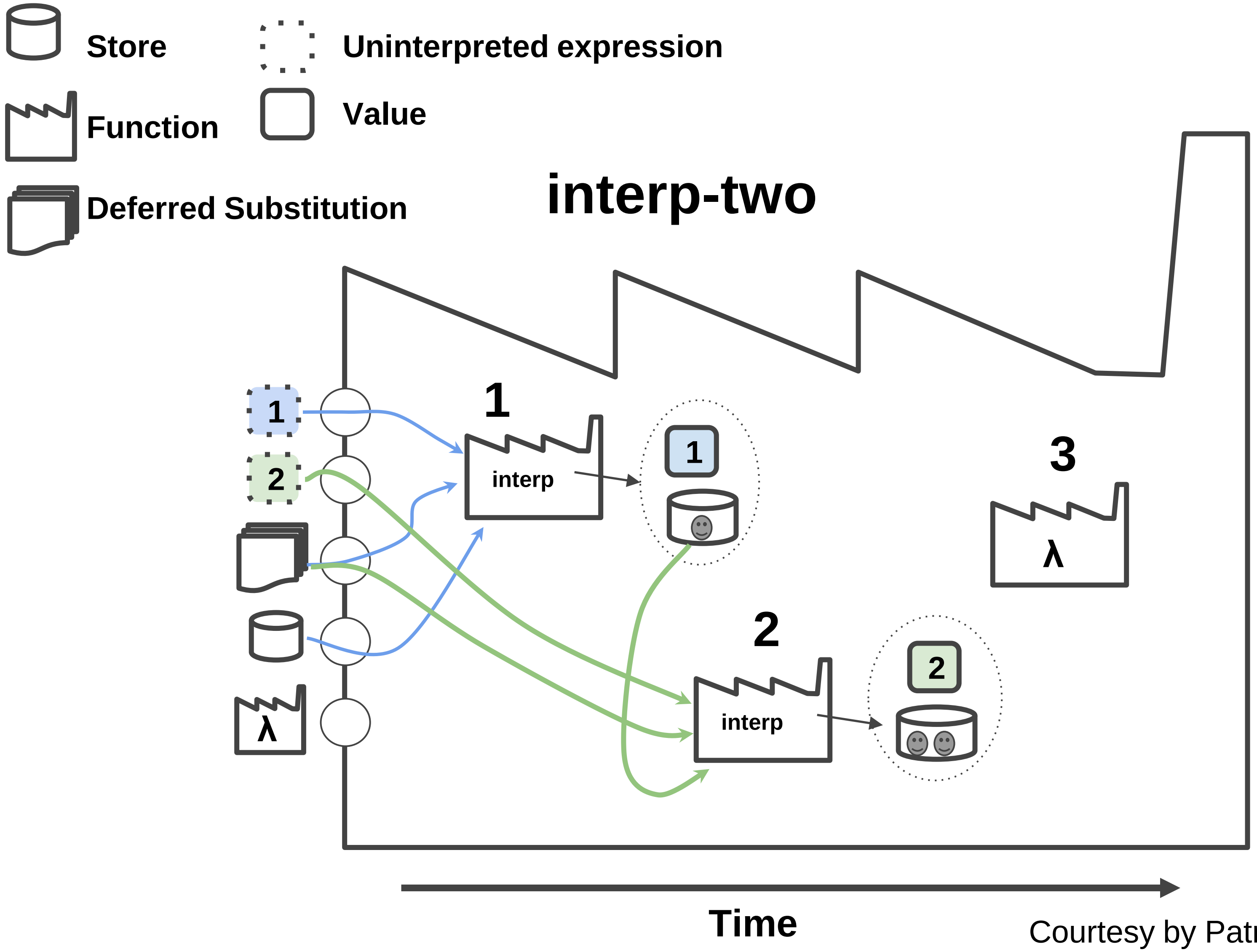
```
;interp : BFAE DefrdSub Store -> Value*Store
(define (interp expr ds st)
    [add (l r) (interp-two l r ds st (lambda (v1 v2 st1) (v*s (num+ v1 v2) st1)))]
    [seqn (a b) (interp-two a b ds st (lambda (v1 v2 st1) (v*s v2 st1)))]
    [setbox (bx-expr val-expr)
                        (interp-two bx-expr val-expr ds st
                              (lambda (bx-val val st1)
                                    (v*s val
                                          (aSto (boxV-address bx-val)
                                                  val
                                                  st1))))]
...)
```

interp-two?



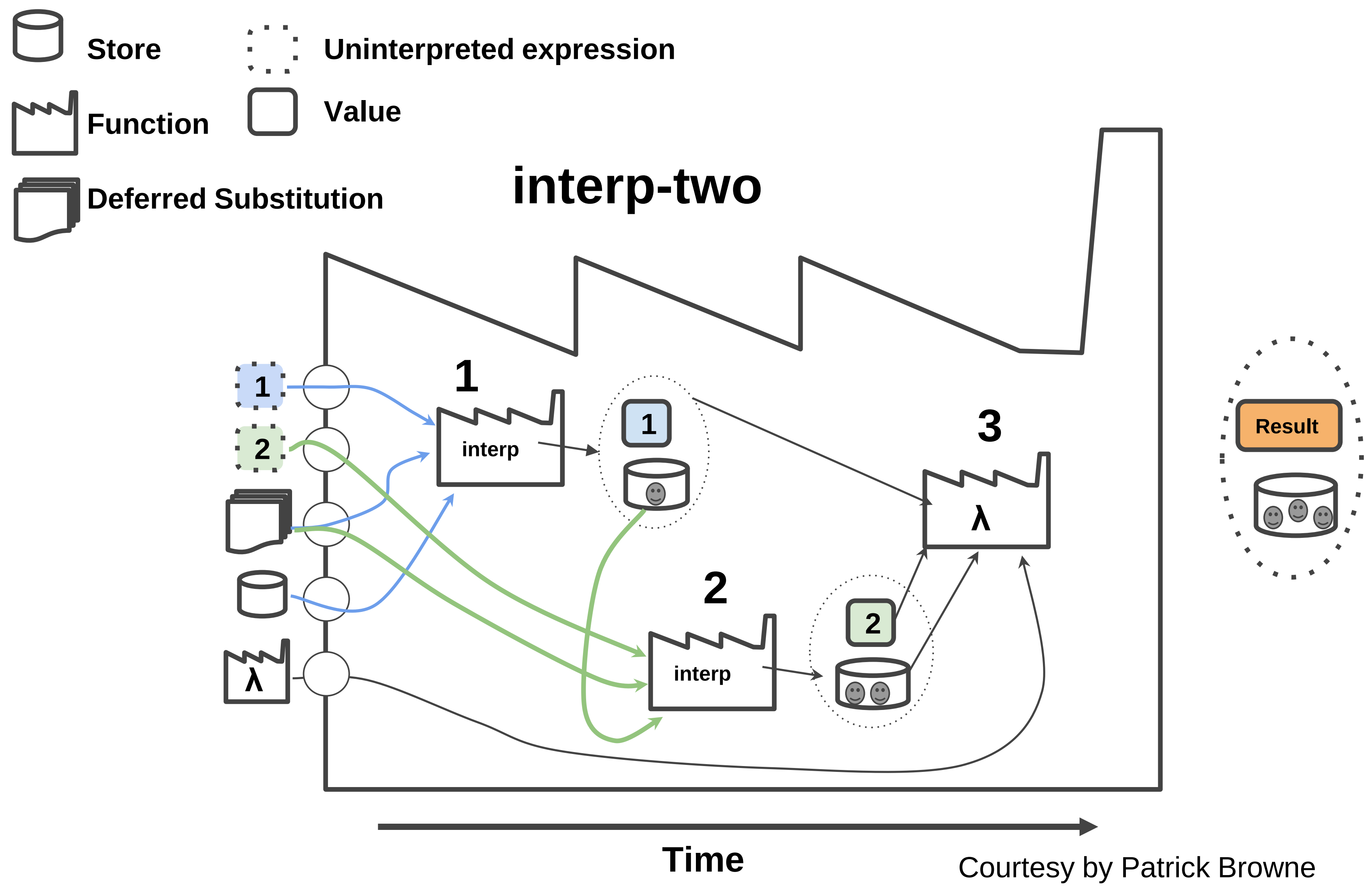
Courtesy by Patrick Browne

interp-two?



Courtesy by Patrick Browne

interp-two?



Courtesy by Patrick Browne

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 (L17,18,19,20)
  - Variables (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 14. Variables

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.