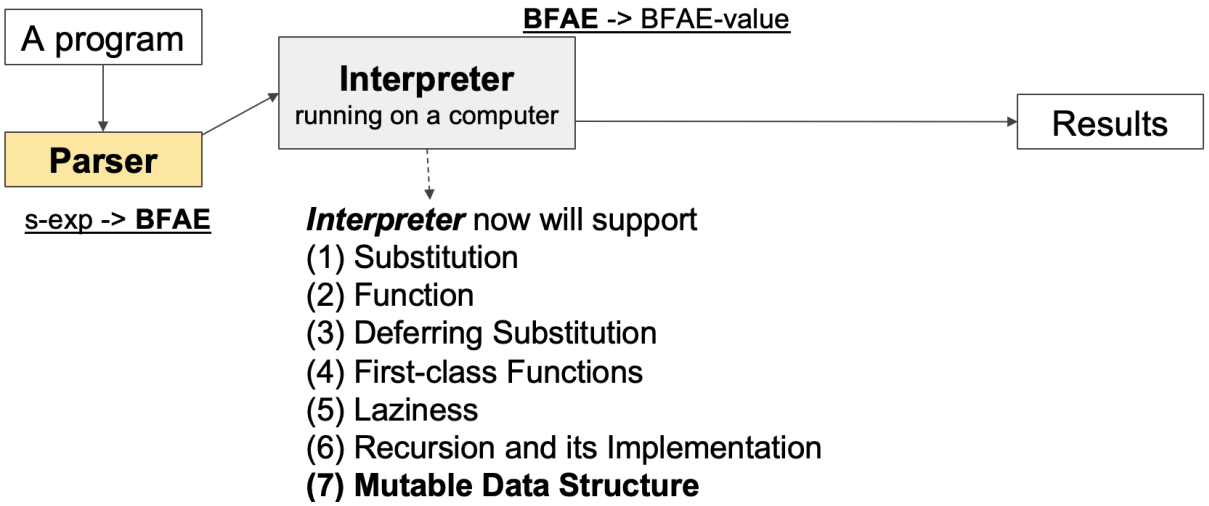
L17 & L18. Mutable Data Structure

유형	강의
복습 여부	☑
Status	In progress

Learn how to support data structure for our language.



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.

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

→ something can be changed = mutable!

Non-functional Procedure

```
(define (f x)
  (+ x (read))) # read : receives input from user

(f 5)
```

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

(g 5)

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

(h 2)
```

input이 4라 가정 했을 때,

```
4
9
5
2
```

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!

cf. Mutable data structure is *a preliminary step toward supporting variables* in our language.

BFAE = FAE + Boxes

```
<BFAE> ::= <num>
         | {+ <BFAE> <BFAE>}
         | {- <BFAE> <BFAE>}
         | <id>                    ⇒ bound with values including number, function, or box
         | {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}}}
```

The example above will produce value 10 as a result.

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

We also want to return box itself as a value.

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

The example above will produce value 1 as a result.

Mutation in the first operation in the sequence has an effect in the output of the second.

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 effect e2?

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

Think about this example!

We need to support both static scope and mutation for the box.

```
{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}}}}
```

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?

Q. How about this code below?

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

→ Because we adopt static scope, the result is 13.

Idea

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

    Ex. {with {b {newbox 0}}}

    'b' is not binding with value 0 but with address of the box. In deferred substitution cache, value of b will not change but always keep its box address.

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

  - **Another for tracking dynamic changes of boxes.**

    → this cache is called 'Store' to distinguish with deferred substitution cache.

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

- Return both

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

  define a new data type that holds both the value and the storage information.

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