We have previously seen how pairs can be represented purely in terms of procedures.

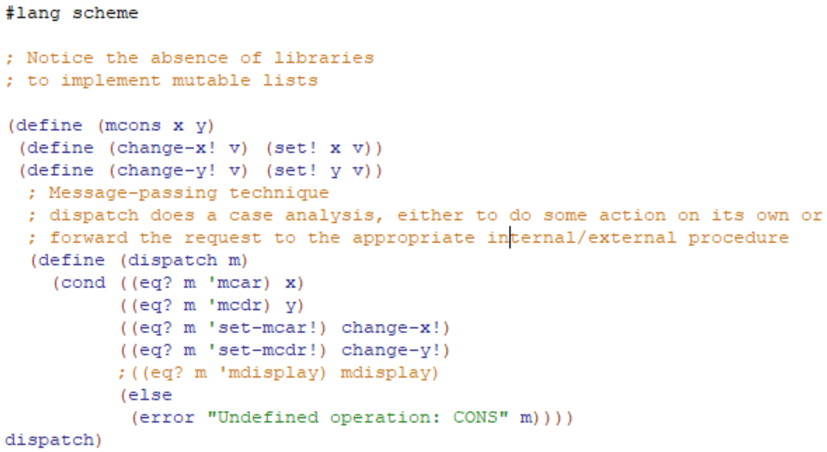
We saw how one could implement cons, car, and cdr without using any data structures at all but only using procedures.

We have previously seen how pairs can be represented purely in terms of procedures.

We saw how one could implement cons, car, and cdr without using any data structures at all but only using procedures.

Similarly, we can implement mutable data objects as procedures using assignment and local state.

In simple terms, **set! is enough**.
set-mcar!, set-mcdr! can be implemented using just set!.



```
#lang scheme

; Notice the absence of libraries
; to implement mutable lists

(define (mcons x y)
 (define (change-x! v) (set! x v))
 (define (change-y! v) (set! y v))
  ; Message-passing technique
  ; dispatch does a case analysis, either to do some action on its own or
  ; forward the request to the appropriate internal/external procedure
   (define (dispatch m)
     (cond ((eq? m 'mcar) x)
           ((eq? m 'mcdr) y)
           ((eq? m 'set-mcar!) change-x!)
           ((eq? m 'set-mcdr!) change-y!)
           ;((eq? m 'mdisplay) mdisplay)
           (else
            (error "Undefined operation: CONS" m))))
dispatch)
```

```
; defining each of the procedures one
; intends to allow the object to make use of

(define (mcar z) (z 'mcar))
(define (mcdr z) (z 'mcdr))

(define (set-mcar! z new-value)
  ((z 'set-mcar!) new-value) z)
(define (set-mcdr! z new-value)
  ((z 'set-mcdr!) new-value) z)

(define (mdisplay z)
  (begin
   (display "{") (display (mcar z)) (display " ")
   (display (mcdr z)) (display "}"))
  (void)
)
; mdisplay is expected to return some object.
; (void) allows us to specify that the
; procedure returns nothing.
```

These external procedures mcar, mcdr, set-mcar! are defined so that they require their first operand to be a procedure. This procedure is the dispatch procedure.
So (mcar! z) is defined to use the procedure z (aka dispatch) and provide 'mcar as its parameter.
The dispatch procedure then returns the locally defined parameter x.

Similarly in case of (set-mcar! z new-value) takes the procedure z (aka dispatch) and uses 'dispatching on type' strategy to direct us towards the appropriate procedure which in this case is the change-x! procedure internal to the object z.

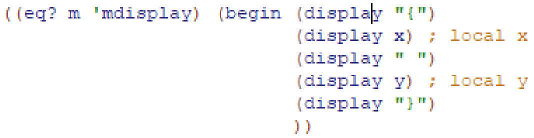
This change-x! procedure changes the value of x to be new-value.
Thus (set-mcar! z new-value)
$\rightarrow$((z 'set-mcar!) new-value) z)
$\rightarrow$(change-x! new-value) internal to z and then return z.

If (set-mcar! z new-value) is defined to be
((z 'set-mcar!) new-value))
instead of
((z 'set-mcar!) new-value) z)

then the first term of the pair would be changed, but nothing would be returned.

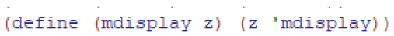
The first way of defining set-mcar! could be more suitable, if one intends to allow nested operations to reuse the same object.

Notice that in the previous example, mdisplay procedure is not present as a type in the dispatch procedure.



```
((eq? m 'mdisplay) (begin (display "{")
                          (display x) ; local x
                          (display " ")
                          (display y) ; local y
                          (display "}")
                         ))
```

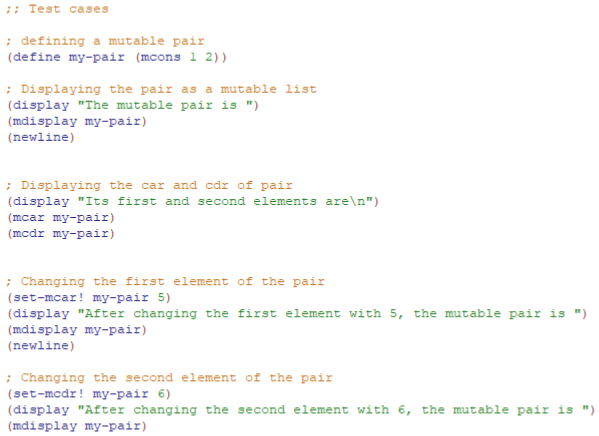
One can make mdisplay procedure also a part of dispatch procedure.



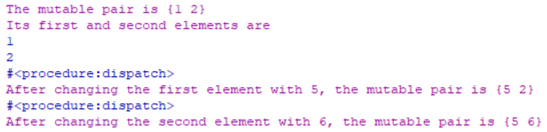
```
(define (mdisplay z) (z 'mdisplay))
```

The external procedure would just be a providing the dispatch with 'mdisplay as an argument.

Since 'mdisplay is now internal, it can directly access x and y without using (mcar z) and (mcdr z).



```
;; Test cases

; defining a mutable pair
(define my-pair (mcons 1 2))

; Displaying the pair as a mutable list
(display "The mutable pair is ")
(mdisplay my-pair)
(newline)


; Displaying the car and cdr of pair
(display "Its first and second elements are\n")
(mcar my-pair)
(mcdr my-pair)


; Changing the first element of the pair
(set-mcar! my-pair 5)
(display "After changing the first element with 5, the mutable pair is ")
(mdisplay my-pair)
(newline)

; Changing the second element of the pair
(set-mcdr! my-pair 6)
(display "After changing the second element with 6, the mutable pair is ")
(mdisplay my-pair)
```



```
The mutable pair is {1 2}
Its first and second elements are
1
2
#<procedure:dispatch>
After changing the first element with 5, the mutable pair is {5 2}
#<procedure:dispatch>
After changing the second element with 6, the mutable pair is {5 6}
```

Thus, assignment is all that is needed, theoretically, to account for the behavior of mutable data.

As soon as we admit set! to our language, we raise all the issues, not only of assignment, but of mutable data in general.

On the other hand, from the viewpoint of implementation, assignment requires us to
modify the environment, which is itself a mutable data structure.

Thus, **assignment and mutation are equipotent**: Each can be implemented in terms of the other.

set! allowed us to change a particular variable in its entirety, aka (set! x new) would change the value of x to new.

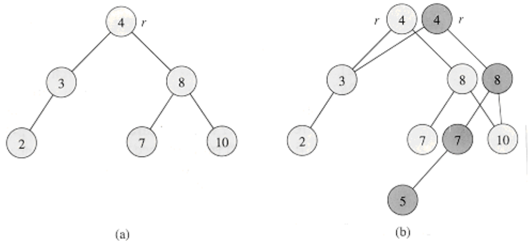
set! allowed us to change a particular variable in its entirety, aka (set! x new) would change the value of x to new.

However, the most important usage of set! is that we can gain the ability to change specific portions of the data.

set! allowed us to change a particular variable in its entirety, aka (set! x new) would change the value of x to new.

However, the most important usage of set! is that we can gain the ability to change specific portions of the data.

This usage along with ability to inspect specified portions of the data are the essential tools one needs to capture the idea of data structures.



Data structures in FP are different from data structures in OOP.

Data structures for FP have the property of keeping previous versions of themselves unmodified.
The above image shows a BST which creates a new version of itself after the insertion of 5 by simply creating a copy of the nodes (and their pointers) which lie on the path from the root to the new node. On the other hand, structures such as arrays (OOP) admit a destructive update ,i.e. an update which cannot be reversed.

A **queue** is a sequence in which items are inserted at one end (called the rear of the queue) and deleted from the other end (the front). A queue is sometimes called a FIFO (first in, first out) buffer.

In terms of data abstraction, we can regard a queue as defined by the following set of operations:

1. a constructor:
   (make-queue)
2. two selectors:
   (empty-queue? ⟨queue⟩) tests if the queue is empty.
   (front-queue ⟨queue⟩)
3. two mutators:
   (insert-queue! ⟨queue⟩ ⟨item⟩)
   (delete-queue! ⟨queue⟩)

A queue could be represented using as an ordinary list;

- the front of the queue would be the car of the list,
- inserting an item in the queue would amount to appending a new element at the end of the list,
- and deleting an item from the queue would just be taking the cdr of the list.

This representation is inefficient, as insertion requires $\Theta(n)$ cdr operations and thus linear time.

The modification that avoids the drawback is to represent the queue as a **list, together with an additional pointer that indicates the final pair** in the list.

A queue is represented, then, as a pair of pointers, front-ptr and rear-ptr, which indicate, respectively, the first and last pairs in an ordinary list.

These pointers can be updated efficiently using set!.

Without set!, one could support the feature of deleting the first item of the queue, by performing a cdr on the queue.

```
(define (insert-queue! queue item)
  (let ((new-pair (cons item '())))
    (cond ((empty-queue? queue)
           (set-front-ptr! queue new-pair)
           (set-rear-ptr! queue new-pair)
           queue)
          (else
           (set-cdr! (rear-ptr queue) new-pair)
           (set-rear-ptr! queue new-pair)
           queue))))
```

However, insertion requires us to change the rear-pointer.