

Now that we have introduced `set!` to the language, we can try to use `set!` to modify lists.

Let's try to see if `set!` allows us to modify other objects as well.

```
#!/
An attempt to build set-car! procedure,
which changes the first element of a list
does not work as one might expect it to work
|#

(define (set-car! x lis)
  (set! lis (cons x (cdr lis)))
  (display "Modifying list1 using a procedure we get \n")
  lis
)

(set-car! 5 list1)
(display "However, if one checks the contents of list1, we get\n")
list1

#!/
This does not change list1 because the set-car! procedure
creates a new list with desired contents
without modifying the existing list.
The default list mechanism is not designed to
accommodate change in content
|#
```

The output is

```
list1 is
(1 2)
Modifying the list using a procedure we get
(5 2)
However, if one checks the content of list1, we get
(1 2)
```

One could use such a procedure and try to modify the list. However, since lists are not designed to be mutable, the procedure is designed to create its own modified version of the list we are trying to modify.

Although the intention is to modify an existing list (say list1), it ends up creating a different list without modifying the existing list (here list1).

The challenge in this case, is not to create a modified version of an entity, but rather to associate a pre-existing identity to the entity instead of creating a new entity with a new identity.

In order to model compound objects with changing state, we will design data abstractions to include, in addition to selectors and constructors, operations called **mutators**, which modify data objects.

In order to model compound objects with changing state, we will design data abstractions to include, in addition to selectors and constructors, operations called **mutators**, which modify data objects.

We introduced pairs as a general-purpose “glue” for synthesizing compound data while dealing with functional programming.

A natural extension would be defining basic mutators for pairs, so that pairs can serve as building blocks for constructing mutable data objects in imperative programming.

The basic operations on pairs— `cons`, `car`, and `cdr`—can be used to construct list structure and to select parts from list structure, but they are incapable of modifying list structure.

The same is true of the list operations like `append` and `list`, which can be defined in terms of `cons`, `car`, and `cdr`.

To modify list structures we need new operations like `set-car!` and `set-cdr!` to change the `car` and `cdr` of a pair.

The idea of `set-car!` and `set-cdr!` is provided in scheme, but the procedure-name (and other minor details) can be different across implementations.

We shall use the following libraries for now.

```
; Libraries to implement mutable lists
(require compatibility/mlist)
(require rnrs/mutable-pairs-6)
```

The textbook mentions operators like `set-car!` and `set-cdr!`. Whereas the aforementioned library uses operators like `mlist`, `set-mcar!` and `set-mcdr!` to avoid any confusion while dealing with lists and mutable lists.

Mutable pairs in scheme

```
(define mutable-pair (mcons 3 4))  
mutable-pair  
(set-mcar! mutable-pair 5)  
mutable-pair  
(set-mcdr! mutable-pair 6)  
mutable-pair
```

Notice the curly brackets instead of round brackets to display the output.

```
{3 . 4}  
{5 . 4}  
{5 . 6}
```

mcar and mcdr work exactly car and cdr but for mutable pairs.

Lists vs Mutable Lists

```
(define list1 (list 1 2))  
(display "list1 is \n")  
list1  
  
(define mutable-list1 (mlist 1 2))  
(display "mutable-list1 is \n")  
mutable-list1
```

Scheme displays lists as delimited by `()`, and
mutable lists as delimited by `{}`
One can use `mlist` to create mutable lists.

```
list1 is  
(1 2)  
mutable-list1 is  
{1 2}
```

```
; changing the car part of a mutable list
```

```
(set-mcar! mutable-list1 5)  
(display "Updating the head\n")  
mutable-list1
```

```
; changing the cdr part of a mutable list
```

```
(set-mcdr! mutable-list1 (mlist 11 12))  
(display "Updating the tail\n")  
mutable-list1|
```

The output would be as follows

```
mutable-list1 is  
{1 2}  
Updating the head  
{5 2}  
Updating the tail  
{5 11 12}
```

one can use cons and mcons to construct two different types of lists

```
-->(cons 4 mutable-list1) creates a new 'list' with two items  
(4 . {5 11 12})  
-->(mcons 4 mutable-list1) creates a new 'mlist' with 4 added  
before the first element in mutable-list1  
{4 5 11 12}
```

mcons does to mlists what cons does for lists.

However, lists and mutable lists are considered different types of entities and thus performing cons (which is designed for pairs/lists which was immutable) on items which are mutable would create a pair instead of a mutable list.

```
> (define combo (cons 4 mutable-list1) )  
> combo  
(4 . {5 11 12})  
> (set-mcar! (cdr combo) 10)  
> combo  
(4 . {10 11 12})
```

In this example, combo is an immutable list, consisting of mutable list.

By introducing set! to the language, we realised that a variable somehow refers to a place where a value can be stored, and the value stored at this place can change.

The theoretical issues of “sameness” and “change” raised by the introduction of assignment.

These issues arise in practice when individual pairs are shared among different data objects.

```
(define x (mlist 'a 'b))
(define z1 (mcons x x))
(define z2 (mcons
            (mlist 'a 'b)
            (mlist 'a 'b)))

(define (set-to-wow! x)
  (set-mcar! (mcar x) 'wow)
  x)

(set-to-wow! z1)
(set-to-wow! z2)
```

The contents of z1 and z2 are exactly the same. However applying set-to-wow! procedure to z1 and z2 produces two different outcomes.

```
{{wow b} wow b}
{{wow b} a b}
```

In general, sharing is completely undetectable if we operate on lists using only cons, car, and cdr (i.e. in the functional programming paradigm).

However, if we allow mutators on list structure, sharing becomes significant as evident through the `set-to-wow!` procedure.

One way to detect sharing in list structures is to use the predicate `eq?`, as a way to test whether two symbols are equal.

(eq? x y) tests whether x and y are the same object (that is, whether x and y are equal as pointers).

Thus, with z1 and z2

(eq? (car z1) (cdr z1)) is true and

(eq? (car z2) (cdr z2)) is false

`(eq? x y)` tests whether `x` and `y` are the same object (that is, whether `x` and `y` are equal as pointers).

Thus, with `z1` and `z2`
`(eq? (car z1) (cdr z1))` is true and
`(eq? (car z2) (cdr z2))` is false

Thus, sharing can also be dangerous, since modifications made to structures will also affect other structures that happen to share the modified parts.

The mutation operations `set-mcar!` and `set-mcdr!` should be used with care. Unless we have a good understanding of how our data objects are shared, mutation can have unanticipated results.


```
(define x (mlist 'a 'b))
(define y (mcons x x))
(display "x and y before updation are\n")
x
y

; Intend to change the first item
; of the first item in y to character c.

(set-mcar! (mcar y) 'c)
(display "x and y after updation are\n")
x
y
; Not clear, if the intention was to
; also change the 2nd item of the same object y.
; Not clear, if the intention was to change the
; first item of some other object x.

x and y before updation are
{a b}
{{a b} a b}
x and y after updation are
{c b}
{{c b} c b}
```

x is used to construct y .

So a change in x causing a change in y can make sense in some situations.

However changing one portion of y causing the change in other portion of y as well as a change in x , may not always be intentional/expected.

Inadvertent sharing/ unanticipated interactions between objects can be source of a lot bugs in large programs.

y has its own identity irrespective of its value. By extension, one realizes that mcons also has its own identity. This instance of **mcons** is different from some instance of mcons present somewhere else.

Introducing the possibility of things having identity and sharing, possibly having multiple names for the same thing (in this example (mcar y) and x are two names for the same entity) we get a lot of power.

But one might have to pay for it with unnecessary complexity and bugs caused by unexpected interactions.

One naive way to check for equality would be to compare the contents of the two entities being compared.

2 is considered equal to 2, whereas it would be considered equivalent to $(+ 1 1)$. If 2 is referring to some other entity other than just numbers, say birthdate, it would not make sense to say that 12th March is equivalent to $1(+ 1 1)$ th march.

So equality and equivalence can be context sensitive and thus possibly have different answers in different contexts.

$x^3 + 1$ is neither equal nor equivalent to $x - 2$. But if the context was modular arithmetic with respect to the modulus 3, then they would be considered equivalent, since the output of the two functions would be same for every integer input.

Even if we define equality to return true, only if the entire content of the two entities are exactly the same, without concerning ourselves with the context, we can still face some issues.

The time required to answer the equality query becomes dependent on the size of the content of the two entities, which is not a problem on its own, but if the object is referring to an infinite set of items, the naive way of checking equality may not terminate.

```

(require compatibility/mlist)
(require rnrs/mutable-pairs-6)

(define (last-pair x)
  (if (null? (mcar x)) x (last-pair (mcdrr x))))

(define z (mlist 'a 'b 'c))

(display "z before make-cycle ")
z

(define (make-cycle x)
  (set-mcdrr! (last-pair x) x)
  x)

(set! z (make-cycle z))


(display "z after make-cycle ")
z
(display "1st item of z is ") (mcar z)
(display "2nd item of z is ") (mcar (mcdrr z))
(display "3rd item of z is ") (mcar (mcdrr (mcdrr z)))
(display "4th item of z is ") (mcar (mcdrr (mcdrr (mcdrr z))))

; Checking equality of 1st and 4th item of z

(eq? (mcar z) (mcar (mcdrr (mcdrr (mcdrr z)))))

; This would not terminate
=length z

```

z before make-cycle {a b c}
z after make-cycle #0=(a b c . #0#)
1st item of z is a
2nd item of z is b
3rd item of z is c
4th item of z is a
#t
 user break

One could in theory rewrite `mlength` to correctly take into account that `z` is a circular linked-list while trying to find its length. However one is still not sure if all the issues have been fixed and whether in theory the `set!` operator is technically sound which does not cause other existing infrastructure to break-down.