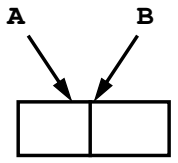
Sharing vs. Copying

In languages without side-effects an object can be copied by copying a pointer (reference) to the object; a complete new copy of the object isn't needed.

Hence in Scheme `(define A B)` normally means

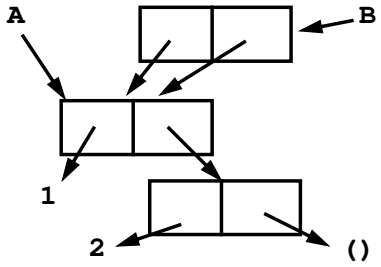


But, if side-effects are possible we may need to force a physical copy of an object or structure:

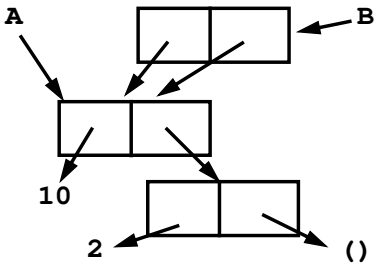

```
(define (copy obj)
  (if (pair? obj)
    (cons (copy (car obj))
          (copy (cdr obj)))
    obj
  )
)
```

For example,

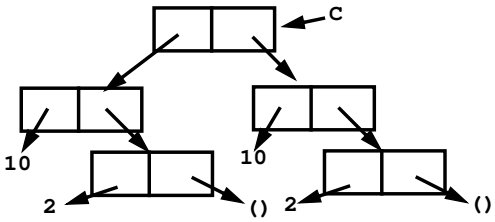
```
(define A '(1 2))
(define B (cons A A))
B = ( (1 2) 1 2)
```



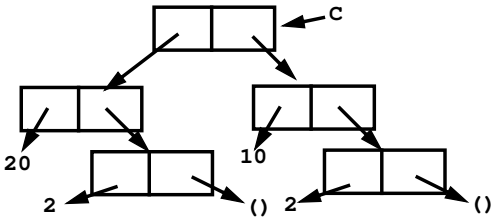

```
(set-car! (car B)  10)
B = ( (10 2) 10 2)
```




```
(define C (cons (copy A) (copy A)))
```




```
(set-car! (car C) 20)
C = ((20 2) 10 2)
```



Similar concerns apply to strings and vectors, because their internal structure can be changed.

Shallow & Deep Copying

A copy operation that copies a pointer (or reference) rather than the object itself is a *shallow copy*.

For example, In Java,

```
Object O1 = new Object();
Object O2 = new Object();
O1 = O2; // shallow copy
```

If the structure within an object is physically copied, the operation is a *deep copy*.

In Java, for objects that support the clone operation,

```
O1 = O2.clone(); // deep copy
```

Even in Java's deep copy (via the `clone()` operation), objects referenced from within an object are shallow copied. Thus given


```
class List {
  int  value;
  List next;
}
List L,M;
M = L.clone();
```

`L.value` and `M.value` are independent, but `L.next` and `M.next` refer to the same `List` object.

A complete deep copy, that copies all objects linked directly or indirectly, is expensive and tricky to implement.

(Consider a complete copy of a circular linked list).

Equality Checking in Scheme

In Scheme = is used to test for numeric equality (including comparison of different numeric types). Non-numeric arguments cause a run-time error. Thus

```
(= 1 1)  ⇒ #t
(= 1 1.0)  ⇒ #t
(= 1 2/2)  ⇒ #t
(= 1 1+0.0i)  ⇒ #t
```


To compare non-numeric values, we can use either:

pointer equivalence (do the two operands point to the same address in memory)

structural equivalence (do the two operands point to structures with the same size, shape and components, even if they are in different memory locations)

In general pointer equivalence is faster but less accurate.

Scheme implements both kinds of equivalence tests.

(eqv? obj1 obj2)

This tests if **obj1** and **obj2** are the exact same object. This works for atoms and pointers to the same structure.

(equal? obj1 obj2)

This tests if **obj1** and **obj2** are the same, component by component. This works for atoms, lists, vectors and strings.

```
(eqv? 1 1) ⇒ #t
(eqv? 1 (+ 0 1)) ⇒ #t
(eqv? 1/2 (- 1 1/2)) ⇒ #t
(eqv? (cons 1 2) (cons 1 2)) ⇒
   #f
(eqv? "abc" "abc") ⇒ #f
```



```
(equal? 1 1) ⇒ #t
(equal? 1 (+ 0 1)) ⇒ #t
(equal? 1/2 (- 1 1/2)) ⇒ #t
(equal? (cons 1 2) (cons 1 2)) ⇒
  #t
(equal? "abc" "abc") ⇒ #t
```

In general it is wise to use **equal?** unless speed is a critical factor.

I/O in Scheme

Scheme provides simple read and write functions, directed to the "standard in" and "standard out" files.

(read)

Reads a single Scheme object (an atom, string, vector or list) from the standard in file. No quoting is needed.

(write obj)

Writes a single object, **obj**, to the standard out file.

(display obj)

Writes **obj** to the standard out file in a more readable format. (Strings aren't quoted, and characters aren't escaped.)

(newline)

Forces a new line on standard out file.

Ports

Ports are Scheme objects that interface with systems files. I/O to files is normally done through a port object bound to a system file.

(open-input-file "path to file")

This returns an input port associated with the **"path to file"** string (which is system dependent). A run-time error is signalled if **"path to file"** specifies an illegal or inaccessible file.

(read port)

Reads a single Scheme object (an atom, string, vector or list) from **port**, which must be an input port object.

`(eof-object? obj)`

When the end of an input file is reached, a special eof-object is returned. `eof-object?` tests whether an object is this special end-of-file marker.

`(open-output-file "path to file")`

This returns an output port associated with the `"path to file"` string (which is system dependent). A run-time error is signalled if `"path to file"` specifies an illegal or inaccessible file.

`(write obj port)`

Writes a single object, `obj`, to the output port specified by `port`.

`(display obj port)`

Writes `obj` to the output port specified by `port`. display uses a more readable format than write does. (Strings aren't quoted, and characters aren't escaped.)

`(close-input-port port)`

This closes the input port specified by `port`.

`(close-output-port port)`

This closes the output port specified by `port`.

Example—Reading & Echoing a File

We will iterate through a file, reading and echoing its contents. We need a good way to do iteration; recursion is neither natural nor efficient here.

Scheme provides a nice generalization of the `let` expression that is similar to C's for loop.

```
(let X ( (id1 val1) (id2 val2) ... )
      ...
      (X v1 v2 ...)
)
```

A name for the let (`X` in the example) is provided. As usual, `val1` is evaluated and bound to `id1`, `val2` is evaluated and bound to `id2`, etc. In the body of the let, the let may be "called" (using its name) with a fresh

set of values for the let variables. Thus `(X v1 v2 ...)` starts the next iteration of the let with `id1` bound to `v1`, `id2`, bound to `v2`, etc.

The calls look like recursion, but they are implemented as loop iterations.

For example, in

```
(let loop ( (x 1) (sum 0) )
  (if (<= x 10)
      (loop (+ x 1) (+ sum x))
      sum
  )
)
```

we sum the values of `x` from 1 to 10.

Compare it to

```
for (x=1,sum=0; x <= 10;
     sum+=x,x+=1)
 {}
```

Now a function to read and echo a file is straightforward:

```
(define (echo filename)
  (let (
   (p (open-input-file filename)))
    (let loop ( (obj (read p)))
      (if (eof-object? obj)
          #t ;normal termination
          (begin
            (write obj)
            (newline)
            (loop (read p))
          )
      )
    )
  )
)
```


We can create an alternative to **echo** that uses

```
(call-with-input-file
   filename function)
```

This function opens **filename**, creates an input port from it, and then calls **function** with that port as an argument:

```
(define (echo2 filename)
  (call-with-input-file filename
    (lambda (port)
      (let loop ( (obj (read port)))
        (if (eof-object? obj)
         #t
         (begin
           (write obj)
           (newline)
           (loop (read port))
         )
        )
      )
    )
) )
```