Stack Implemented as a Function

(define (stack)
  (let ((s '()) ; var # args
    (lambda (op . args)
      (cond
        ((equal? op 'push!) (set! s (cons (car args) s)) (car s))
        ((equal? op 'pop!) (if (null? s)
                               #f
                               (let ((top (car s)))
                                (set! s (cdr s))
                                top)))
        ((equal? op 'empty?) (null? s))
        (else #f))
      ))))
)

Higher-Order Functions

A higher-order function is a function that takes a function as a parameter or one that returns a function as its result.

A very important (and useful) higher-order function is map, which applies a function to a list of values and produces a list of results:

(map sqrt '(1 2 3 4 5)) ⇒ (1 1.414 1.732 2 2.236)
(map (lambda (x) (* x x)) '(1 2 3 4 5)) ⇒ (1 4 9 16 25)

Map may also be used with multiple argument functions by supplying more than one list of arguments:

(map + '(1 2 3) '(4 5 6)) ⇒ (5 7 9)
The Reduce Function

Another useful higher-order function is `reduce`, which reduces a list of values to a single value by repeatedly applying a binary function to the list values. This function takes a binary function, a list of data values, and an identity value for the binary function:

\[
\text{(define } \text{reduce } f \text{ L id)} \\
\text{if (null? L)} \\
\text{id} \\
(f \text{ (car L)} \\
\text{(reduce } f \text{ (cdr L) id)} \\
) \\
\text{(reduce } + \text{ '(1 2 3 4 5) 0) } \Rightarrow 15 \\
\text{(reduce } * \text{ '(1 2 4 6 8 10) 1) } \Rightarrow 3840
\]

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

```
(define A (list 1 2))
(define B (list 3 4))
```

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,

\[
\begin{align*}
&\text{(define A '(1 2))} \\
&\text{(define B (cons A A))} \\
&B = ( (1 2) 1 2)
\end{align*}
\]

\[(\text{set-car! (car B) 10})\]

\[
B = ( (10 2) 10 2)
\]

\[
\begin{align*}
&\text{(define C (cons (copy A) (copy A)))} \\
&C
\end{align*}
\]

\[
\begin{align*}
&\text{(set-car! (car C) 20)} \\
&C = ((20 2) 10 2)
\end{align*}
\]

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

\section*{Shallow & Deep Copying}

A copy operation that copies a pointer (or reference) rather than the object itself is a \textit{shallow copy}. For example, In Java,

\[
\begin{align*}
&\text{Object O1 = new Object();} \\
&\text{Object O2 = new Object();} \\
&\text{O1 = O2; // shallow copy}
\end{align*}
\]

If the structure within an object is physically copied, the operation is a \textit{deep copy}.

In Java, for objects that support the clone operation,

\[
\begin{align*}
&\text{O1 = O2.clone(); // deep copy}
\end{align*}
\]

Even in Java's deep copy (via the \texttt{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.