Notes on Higher Order Programming in Scheme

by Alexander Stepanov

August 1986

INTRODUCTION

Why Scheme?

Because it allows us to deal with:

1. Data Abstraction - it allows us to implement ADT (abstact data types) in a very special way. The issue of data abstraction is addressed in other languages: clusters in CLU, modules in MODULA, generics in ADA. But only SCHEME allows us to treat ADT as "first class objects." It allows us to pass them as parameters, return them as values, store them in data structures. We can deal with abstract objects in the same way we deal with integers.

2. Procedural Abstraction - the notion of procedural abstraction (functional form) is overlooked in most conventional languages. And those languages which utilize functional forms do not treat them as first class objects. For example, APL restricts us to about five functional forms introduced by Iverson in 1961. And a major goal of this course is to show that procedural abstaraction is the main tool for design of algorithms.

Aplicative order

```
((lambda (x y) (* x (+ y 2))) 5 0)
```

How does SCHEME evaluate an expression?
1. it checks whether a first element of an expression is a "special form" ("magic word").
2. if it is not (and in our case it isn't - our first element is not a word at all - it is an expression) all elements of the expression are evaluated in some unspecified order (could be in parallel).
(2.1) If it is a special form, then SCHEME does a special thing.
3. Result of the evaluation of the first element (which better be a procedural object) is "applied" to the results of evalution of the rest of the elements.

In our case 5 evaluates to 5, 0 evaluates to 0 (numbers are "self-evaluating" objects, actually, all atomic object, with the exeption of symbols, are self-evaluating), but how does SCHEME evaluate (lambda (x y) (* x (+ y 2)))?
It looks at its first elmenent and finds that it is a special form "lambda". This special form creates a procedure with formal arguments x and y and procedure body (* x (+ y 2)).

How does SCHEME apply a procedure?
1. Current "environment" is extended by "binding" formal arguments to actual arguments (in our case ((x 5) (y 0)))
(in TI SCHEME we can actually see how it is done by changing our expression to

```
((lambda (x y)
   (display (environment-bindings (the-environment)))
   (* x (+ y 2)))
 5
 0)

)
```

2. Evaluating the body of the procedure in the extended environment
...

Global environment

Global environment is an environment which containes all initial bindings (in TI SCHEME system bindings are in user-global-environment which is a parent of user-initial-environment in which user's global bindings are)

define

we can extend our global environment by typing

(define foo 88)

which would add to it a binding (foo 88)

is "define" a procedure or a special form?

if it were a procedure it would get a value of "foo" and not "foo" and it would be impossible for it to create a binding (foo 88) define does not evaluate its first argument, but does evaluate its second argument.

if we say

foo

system will evaluate it and return the result

now say

bar

see what happens!

now let us define a global function

(define square (lambda (x) (* x x)))

there is a short hand for such defines; we can say

(define (square x) (* x x))

now, do

(square 2)

now, we can do the following

(define bar square)

(bar 2)

explain ...

Now we can define the most useful function which is going to be used throughout and which is not a part of standard SCHEME

(define (identity x) x)

Free variables

A variable in the body of a procedure is called "free" if it is not bound in this procedure

(lambda (x y) ((lambda (x y z) (+ x (* y z))) x y a))

a is a free variable

Lexical scoping

Free variables are associated to a lexically apparent binding (to a binding which "textually" encloses the body)

Try the following

```
(define b 1)

((lambda (a b) (a 5)) (lambda (x) (+ x b)) 2)
```

the second lambda has a free variable b which is associated with the global binding (b 1) even when it is called within the first lambda where b is bound to 2

Indefinite (unlimited) extent

All the objects in SCHEME, environment bindings including, live forever. It means that in some cases a binding in the environment of a procedure can be used after the procedure terminated

```
(define (make-add-a-constant c)
  (lambda (x) (+ x c)))

(define one-plus (make-add-a-constant 1))

(define two-plus (make-add-a-constant 2))

(define seven-plus (make-add-a-constant 7))
```

So we can define functions which make functions

Actually, make-add-a-constant is just an instant of more general and more useful functions:

```
(define (bind-1-of-2 function constant)
  (lambda (x) (function constant x)))

(define (bind-2-of-2 function constant)
  (lambda (x) (function x constant)))
```

that make a function of one variable out of a function of two

Problem:

```
(define foo (bind-1-of-2 / 1))
```

what does foo do?

square can be defined with the help of a following function

```
(define (D-combinator function)
  (lambda (x) (function x x)))
```

(it was introduced by M. Schoenfinkel in 1924, 50 years before SCHEME)

```
(define square (D-combinator *))
```

we also can make a function that composes two functions:

```
(define (compose f g)
  (lambda (x) (f (g x))))
```

and a function that takes two functions and returns a function that
applies them to an argument sequentially

```
(define (S-combinator f g)
  (lambda (x) (f x) (g x)))
```

Problem 1.1:

Define a function FUNCTIONAL-DIFFERENCE that takes two functions F(x) and G(x) as and returns a function W(x)=F(x)-G(x)

Problem 1.2:

Define a function T-combinator that takes a function f(x y) and returns a function g(x y)=f(y x)

What is ((T-combinator -) 5 2)?

Problem 1.3:

What does the following function do:

```
(define foobar
  ((t-combinator functional-difference)
   identity
   (d-combinator *)))
```

Conditonal

The primitive conditional construct in Scheme is

(if condition consequent alternative)

The condition is evaluated and if it returns a true value (anything, but #!false or ()) the consequent is evaluated and its value is returned, otherwise the alternative is evaluated and its value is returned.

If "if" does not have an alternative then the if expression is evaluated only for its effect and the result is not specified

We can define if-combinator

```
(define (if-combinator predicate f g)
  (lambda (x) (if (predicate x) (f x) (g x))))
```

Problem:

```
(define foo (if-combinator odd? 1+ identity))
```

what does foo do?

Actually, it is also useful to have another combinator

```
(define (when-combinator predicate function)
  (lambda (x) (if (predicate x) (function x))))
```

It has two arguments: predicate P and function F, it returns a function that applies F only to those arguments that satisfy P.

Factorial example

Now we can implement factorial in a traditional recursive way

```
(define factorial
  (lambda (n)
    (if (= n 0)
        1
        (* n (factorial (- n 1))))))
```

While the program does work it is not quite "first class".
its correctness depends on the global binding of "factorial"
so if we do something like

```
(define new-factorial factorial)

(define factorial *)
```

(new-factorial 5) is going to return 20 in stead of 120

So what we want is to make a recursive functional object to be independant of its global name namely, we want to bind name factorial to the procedural object in the environment of this procedural object.

There is a special form "named-lambda":

```
(named-lambda (name var1 ...) body)
```

which does just that.

It works just as lambda, but also binds a procedural object
it returns to name in the environment of the procedural object

And we can define factorial as:

```
(define factorial
  (named-lambda (factorial n)
    (if (= n 0)
        1
        (* n (factorial (- n 1))))))
```

now, the self-recursive reference is done through the local binding which cannot be affected by changing the global binding of factorial.

Tail Recursion

Our definition of factorial has one problem: it pushes the stack. The reason for that is that multiplication in the first call cannot be evaluated until the result of second call is returned and so on. But if we change our definition into

```
(define (factorial-loop i result n)
  (if (> i n)
      result
      (factorial-loop (+ i 1) (* result i) n)))
```

and

```
(define (factorial n)
  (factorial-loop 1 1 n))
```

SCHEME is not going to push the stack because there is no need to keep the environment ...

Actually, the better way to do this is by making factorial-loop local procedure in factorial:

```
(define (factorial n)
  (define (factorial-loop i result)
    (if (> i n)
        result
        (factorial-loop (+ i 1) (* result i))))
  (factorial-loop 1 1))
```

This kind of recursion is called tail-recursion and systems that do not push the stack for tail-recursive calls are called "properly tail recursive".

SCHEME is properly tail recursive.

We can ask what are the conditions that allow us to find a tail recursive representation of a recursive function.

It is possible to prove that any primitive-recursive function has a tail recursive form. In SCHEME we can construct the best possible proof of them all: we can implement a function which does the transformation of a primitive-recursive function into a tail recursive form. (we shall restrict ourselves to functions of one variable).

First, we shall make a function that makes a primitive recursive function given a transformation and an initial value

```
(define (make-primitive-recursive transformation initial-value)
  (named-lambda (function n)
    (if (= n 0)
        initial-value
        (transformation n (function (- n 1))))))
```

PROBLEM:

define FACTORIAL with the help of MAKE-PRIMITIVE-RECURSIVE

we can produce an equivalent iterative function with:

```
(define ((make-primitive-iterative transformation initial-value)
         n)
  (define (loop variable result)
    (if (= n variable)
        result
        (loop (+ variable 1)
              (transformation (+ variable 1) result))))
  (loop 0 initial-value))
```

In TI SCHEME not just functions, but environments are first class objects and we can extract transformation and initial value out of a functional object created with the help of make-primitive-recursive.

That allows us to define a function:

```
(define (recursive->iterative function)
  ((lambda (environment)
    (make-primitive-iterative
      (access transformation environment)
      (access initial-value environment)))
   (procedure-environment function)))
```

PROBLEM:

With the help of MAKE-PRIMITIVE-RECURSIVE and MAKE-PRIMITIVE-ITERATIVE implement functions MAKE-ADD-SELECT (PREDICATE) and MAKE-ADD-SELECT-ITERATIVE (PREDICATE) so that they return a function defined on non-negative integers such that for any integer N it returns the sum of those integers less-or-equal to N that satisfy PREDICATE.
Define ADD-ODD as (make-add-select odd?) and ADD-ODD-ITERATIVE as (make-add-select-iterative odd?);
what is the smallest integer i on your system such that (add-odd i) bombs and (add-odd-iterative i) does not?

Now, what if the value of a function on N depends not just on the value on F(N-1), but on F(N-1) and F(N-2)?

```
(define (make-two-recursive transformation value-0 value-1)
  (named-lambda (function n)
    (if (= n 0)
        value-0
        (if (= n 1)
            value-1
            (transformation n
```

```
                              (function (- n 1))
                              (function (- n 2)))))))
```

and the equivalent iterative function can be obtained with:

```
(define ((make-two-iterative transformation value-0 value-1) n)
  (define (loop variable first second)
    (if (= n variable)
        first
        (loop (1+ variable)
              (transformation (1+ variable) first second)
              first)))
  (if (= n 0) value-0
      (loop 1 value-1 value-0)))

(define (two-recursive->iterative function)
  ((lambda (environment)
    (make-two-iterative
      (access transformation environment)
      (access value-0 environment)
      (access value-1 environment)))
   (procedure-environment function)))
```

PROBLEM:

Define a function FIB(n) which returns n-th fibonacci number
with the help of TWO-RECURSIVE.
Time (fib 20).
Transform fib into an iterative function with the help of
TWO-RECURSIVE->ITERATIVE.
Time (fib 20).

Pairs

Primitives:

cons: (cons 1 2) ==> (1 . 2)

car: (car '(1 . 2)) ==> 1

cdr: (cdr '(1 . 2)) ==> 2

```
pair?: (pair? '(1 . 2)) ==> #!true
       (pair? 1)        ==> #!false

set-car!: (define a '(1 . 2)) ==> ??
          (set-car! a 0)      ==> ??
          a                   ==> (0 . 2)
```
  used to be known as rplaca

```
set-cdr!: (define a '(1 . 2)) ==> ??
          (set-cdr! a 0)      ==> ??
          a                   ==> (1 . 0)
```
  used to be known as rplacd

Lists

Primitives:

Empty list:

```
():    '() ==> ()
        (pair? '()) ==> #!false !!! nil is not a pair !!!
```
used to be known as nil

(1 . (2 . (3 . ()))) ==> (1 2 3)

```
null?: (null? '()) ==> #!false
```
used to be known as null

```
Unlike in LISP (car '()) ==> error
               (cdr '()) ==> error
```
TI SCHEME does not signal that error, but no code should depend on (cdr '()) returning '()

Proper list is a pair cdr of which is either a proper list or an empty list

Problem:

Define a predicate PROPER-LIST?

An improper (dotted) list is a chain of pairs not ending in the empty list.

Problem:

Define a predicate IMPROPER-LIST?

More about lambda.

There are three ways to specify formal arguments of a function:

1 - (lambda variable <body>) ==> the procedure takes any number of arguments; they are put in a list and the list is bound to a variable

2 - (lambda proper-list-of-distinct-variables <body>)
the procedure takes a fixed number of arguments equal the length of the proper-list-of-distinct-variables; it is an error to give it more or less

3 - (lambda improper-list-of-distinct-variables <body>)
    the extra arguments are bound to the last variable

Non-primitive (but standard) functions on lists

```
(define (caar x) (car (car x)))

(define (cadr x) (car (cdr x)))

(define (cdar x) (cdr (car x)))

(define (cddr x) (cdr (cdr x)))
```

... and up to four letters

```
(define list (lambda x x))
```

Explain!

Problem:

Define a function LENGTH that returns length of a list

Problem:

Define a function REVERSE that returns a newly allocated list consisting of the elements of list in reverse order

Equivalence predicates

<see pages 12-14 of R3R>

Destructive functions

Reverse returns a new list (a new chain of pairs), but we may want to reverse the original list.

A function F is called applicative iff

```
(lambda (x) ((lambda (y) (f x) (equal? x y)) (copy x)))
```

always returns #!true.

For an applicative function F a function F! is its destructive equivalent iff

1. (f x) == (f! (copy x))

2. (not (equal? x (f x)))
   implies
   ((lambda (y) (f x) (not (equal? x y))) (copy x))

From this two axioms we can derive:

Bang rule 1:

(w x) = (f (g x)) => (w! x) = (f! (g! x))

Bang rule 2:

(w! x) = (f! (g! x)) => (w x) = (f! (g x))

Problem:

implement REVERSE!

It is a little more difficult to right an iterative procedure COPY-LIST.

We can always do

```
(define (stupid-copy-list l)
  (if (pair? l)
      (cons (car l) (stupid-copy-list (cdr l)))
      l))
```

as a matter of fact, it is better to define it as:

```
(define (not-so-stupid-copy-list l)
  (reverse! (reverse l)))
```

there is a very good way to do it:

```
(define (rcons x y)
  (set-cdr! x (cons y '()))
  (cdr x))
```

```
(define (copy-list x)
  (define (loop x y)
    (if (pair? y)
        (loop (rcons x (car y)) (cdr y))
        (set-cdr! x y)))
  (if (pair? x)
      ((lambda (header) (loop header (cdr x)) header)
       (list (car x)))
      x))
```

COPY-LIST is still much slower than NOT-SO-STUPID-COPY-LIST

redefine RCONS as:

```
(define-integrable
  rcons
  (lambda (x y)
    (set-cdr! x (cons y '()))
    (cdr x)))
```

and recompile COPY-LIST

Problem:

Implement APPEND as a function of an arbitrary number of lists which returns a list containing the elements of the first list followed by the elements of the other lists the resulting list is always newly allocated, exept that it shares structure with the last list argument. The last argument may actually be any object; an improper list results if it is not a proper list (see R3R page 16).

Problem:

Implement APPEND!

Synactic extensions

So far the only special forms that we used are LAMBDA, IF, DEFINE, QUOTE and SET!

While these forms are powerful enough SCHEME includes several secondary special forms that are normally expressed with the help of the primitive ones.

While SCHEME does not specify a standard mechanism for syntactic expansions actual implementations provide macro mechanism to do the stuff.

Quasiquotation

<see R3R pages 10-11>

Macros

Macro is a function of one argument (macroexpander) associated with a keyword.

When SCHEME compiles an S-expression car of which is a macro keyword it replaces it with a value that is returned by the corresponding macroexpander applied to this S-expression

```
(macro m-square
  (lambda (body)
    `(* ,(cadr body) ,(cadr body))))
```

So if we say

```
(m-square 4)
```

it will expand into

```
(* 4 4).
```

But if we say

```
(m-square (sin 1.234))
```

it will expand into

```
(* (sin 1.234) (sin 1.234))
```

and we are going to evaluate (sin 1.234) twice

```
(macro better-m-square
  (lambda (body)
    (if (or (number? (cadr body))
            (symbol? (cadr body)))
        `(* ,(cadr body) ,(cadr body))
        `((lambda (temp) (* temp temp))
          ,(cadr body)))))
```

Derived special forms

the simpliest special form we can implement is BEGIN

```
(define (begin-expander body)
  `((lambda () . ,(cdr body)))

(macro my-begin begin-expander)
```

one of the most useful ones is COND

```
(define (cond-expander body)
  (define temp (gensym))
  (define (loop clauses)
    (if (pair? clauses)
        (if (pair? (car clauses))
            (if (eq? 'else (caar clauses))
                `(begin . ,(cdar clauses))
                (if (null? (cdar clauses))
                    `((lambda (,temp)
                        (if ,temp ,temp ,(loop (cdr clauses))))
                      ,(caar clauses))
                    `(if ,(caar clauses)
                         (begin . ,(cdar clauses))
                         ,(loop (cdr clauses)))))
            (syntax-error "Wrong clause in COND" body))
        #!false))
  (loop (cdr body)))

(macro my-cond cond-expander)
```

Let us implement a macro BEGIN0 that implements a special form that takes a sequence of forms, evaluates them and returns the value of the first one.

```
(define (begin0-expander body)
  (define temp (gensym))
  (cond ((null? (cdr body))
         (syntax-error "Expression has too few subexpressions"
                       body))
        ((null? (cddr body))
         (cadr body))
        (else `((lambda (,temp) ,@(cddr body) ,temp)
                ,(cadr body)))))

(macro my-begin0 begin0-expander)

(define (and-expander form)
  (cond ((null? (cdr form)) #!true)
        ((null? (cddr form)) (cadr form))
        (else
          `(if ,(cadr form)
               ,(and-expander (cdr form))
               #!false))))

(macro my-and and-expander)

(define (or-expander form)
  (define temp (gensym))
  (cond ((null? (cdr form)) #!false)
        ((null? (cddr form)) (cadr form))
        (else
          `((lambda (,temp)
              (if ,temp
                  ,temp
                  ,(or-expander (cdr form))))
            ,(cadr form)))))

(macro my-or or-expander)
```

Problem:

Define macro WHEN that takes a predicate and any number of forms. It first evaluates the predicate and if it returns a true value evaluates the forms sequentially returning the value of the last form.

```
(define (tak x y z)
  (if (not (< y x))
      z
      (tak (tak (-1+ x) y z)
           (tak (-1+ y) z x)
           (tak (-1+ z) x y))))

;;; (tak 18 12 6)

(define (constant-access-time)
  (define (test-loop x)
    (when (not (zero? x)) (test-loop (- x 1))))
  (timer (test-loop 10000)))

(define (parameter-access-time)
  (define (test-loop x y)
    (when (not (zero? x)) (test-loop (- x y) y)))
  (timer (test-loop 10000 1)))

(define (lexical-access-time)
  (let ((y 1))
    (define (test-loop x)
      (when (not (zero? x)) (test-loop (- x y))))
    (timer (test-loop 10000))))

(define (lexical-access-time-2)
  (let ((y 1))
    (let ((z 2))
      (define (test-loop x)
        (when (not (zero? x)) (test-loop (- x y))))
      (timer (test-loop 10000)))))

(define **y** 1)

(define (global-access-time)
  (define (test-loop x)
    (when (not (zero? x)) (test-loop (- x **y**))))
  (timer (test-loop 10000)))

(define (fluid-access-time-1)
  (define (test-loop x)
    (when (not (zero? x)) (test-loop (- x (fluid y)))))
  (timer (fluid-let ((y 1)) (test-loop 10000))))

(define (fluid-access-time-2)
  (define (test-loop x)
    (when (not (zero? x)) (test-loop (- x (fluid y)))))
  (timer (fluid-let ((y 1)
                     (z 3)) (test-loop 10000))))

(define (fluid-access-time-3)
  (define (test-loop x)
    (when (not (zero? x)) (test-loop (- x (fluid y)))))
  (timer (fluid-let ((y 1)
                     (x 2)
                     (z 3)) (test-loop 10000))))

(define (fluid-access-time-4)
  (define (test-loop x)
    (when (not (zero? x)) (test-loop (- x (fluid y)))))
  (timer (fluid-let ((y 1)
```

```
                    (x 2)
                    (z 3)
                    (w 4)) (test-loop 10000))))

(define (lambda-time)
  (define (test-loop x)
    (when (not (zero? x)) (test-loop ((lambda (x y) (- x y)) x
1))))
  (timer (test-loop 10000)))

(define (funcall-time)
  (define (test-loop x f)
    (when (not (zero? x)) (test-loop (f x 1) f)))
  (timer (test-loop 10000 (lambda (x y) (- x y)))))

(define (global-funcall-time)
  (define (test-loop x f)
    (when (not (zero? x)) (test-loop (f x 1) f)))
  (timer (test-loop 10000 -)))

(define (apply-time)
  (define (test-loop x)
    (when (not (zero? x)) (test-loop (- x (apply - '(2 1))))))
  (timer (test-loop 10000)))
```

```
(define (stupid-copy tree)
  (cond ((atom? tree)
         tree)
        (cons (copy (car tree)) (copy (cdr tree)))))

(define (tree-copy tree)
  (define stack-of-cdrs '())
  (define (tree-copy-loop l)
    (cond ((pair? (car l))
           (if (pair? (cdr l))
               (set! stack-of-cdrs (cons l stack-of-cdrs)))
           (set-car! l (cons (caar l) (cdar l)))
           (tree-copy-loop (car l)))
          ((pair? (cdr l))
           (set-cdr! l (cons (cadr l) (cddr l)))
           (tree-copy-loop (cdr l)))
          ((pair? stack-of-cdrs)
           (let ((i stack-of-cdrs)
                 (j (car stack-of-cdrs)))
             (set! stack-of-cdrs (cdr stack-of-cdrs))
             (set-car! i (cadr j))
             (set-cdr! i (cddr j))
             (set-cdr! j i)
             (tree-copy-loop i)))))
  (if (pair? tree)
      (let ((n (cons (car tree) (cdr tree))))
        (tree-copy-loop n)
        n)
      tree))
```

SCHEME it treats functions as first class objects; i.e., they can be passed as arguments to other functions, stored, and returned as results of functions. This allows us to create operators, i.e., functions that take other functions as arguments, and to write functions which write other functions. SCHEME is also lexically scoped. While we do not make use of this latter feature to a great extent here, we intend to make use of it by creating encapsulated data structures.

In keeping with standard SCHEME notation, we place a bang (!) at the end of the names of functions with side effects and a question mark (?) at the end of predicates.

Whenever possible, we use these newly defined operators, rather than the standard control structures built into SCHEME, to implement the remainder of our functions. We do this for two reasons. First, it further illustrates the use of these new operators. Second, and more important, it is usually easier to use these new operators than it is to use the standard SCHEME control structures and better code results from their use. Thus, many of the examples given below to illustrate the use of the new operators are useful functions (sometimes operators) in their own right.

Some of the functions below not only take functional arguments, but also return functions. These functions generally have names starting with make- . These are true meta-functions which allow us to define an entire class of functions. Note that SCHEME provides a shorthand notation for defining functions that return other functions. An ordinary function is defined by:

```
(define (func arg-1 ... arg-j) form-1 ... form-k)
```

where the arg's are arguments to the functions and the form's are any SCHEME form. A function which returns another function can be defined by:

```
(define ((make-func arg-1 ... arg-i) arg-j ... arg-k)
  form-1 ... form-m)
```

Such a function returns a function of arg-j through arg-k.

FOR-EACH-CDR

Format: (FOR-EACH-CDR function list)

Parameters:
function - A function of one argument.
list - A list containing elements suitable as arguments to the function.

Explanation: FOR-EACH-CDR first applies function to the entire list and then successively to each of its cdr's until the list is empty. The cdr is taken after the function is applied. It returns an unspecified value.

```
Usage:     (define a '(3 1 5 7))          ==> a
           (for-each-cdr
             (lambda (x)
               (set-car! x (1+ (car x))))
               a)                            ==> unspecified
           a                               ==> (4 2 6 8)

           (define (my-for-each function list)
             (for-each-cdr
               (lambda (x) (function (car x))) list))
                                                  ==> my-for-each
```

Implementation:

```
(define (for-each-cdr function list)
  (let loop ((l list))
    (when (pair? l)
          (function l)
          (loop (cdr l)))))
```

FOR-EACH-CDR!

Format: (FOR-EACH-CDR! function list)

Parameters:
function - A function of one argument.
list - A list containing elements suitable as arguments to the function.

Explanation: FOR-EACH-CDR first applies function first to the entire list and then successively to each of its cdr's until the list is empty. The cdr is taken before after the function is applied. It returns an unspecified value.

```
Usage:     (define a '(3 1 5 7))          ==> a
           (for-each-cdr!
             (lambda (x)
               (set-car! x (1+ (car x))))
               a)                            ==> unspecified
           a                               ==> (4 2 6 8)
```

Implementation:

```
(define (for-each-cdr! function list)
  (let loop ((l list))
    (when (pair? l)
          (let ((next (cdr l)))
            (function l)
            (loop next)))))
```

MAP!

Format: (MAP! function list)

Parameters:
function - A function of one argument.
list - A list containing elements suitable as arguments to the function.

Explanation: MAP! applies the function to each element of list in an unspecified order and replaces that element in the list by the result returned by the function.

```
Usage:   (define a '(1 2 3 4))         ==> a
         (map! 1+ a)                   ==> unspecified
         a                             ==> (2 3 4 5)
```

Implementation:

```
(define (map! function list)
  (for-each-cdr
    (lambda (x) (set-car! x (function (car x)))) list)
    list)
```

MAKE-ACCUMULATE

Format: ((MAKE-ACCUMULATE iterator)
           function
           initial-value
           structure)

Parameters:
iterator - An iterator
function - A function of two variables.
initial-value - A value suitable as the first argument to the function.
structure - A structure containing values suitable as the second argument to the function.

Explanation: Make-accumulate creates an accumulating function; i.e., a function which accumulates the results of another function applied to a structure. An accumulating function takes three arguments. The first is an initial value to start the accumulation process. This initial value is used both as a starting value for the result to be returned and as an initial argument to function. The second argument to an accumulating function is a function to be applied. The third argument is a structure to which the function is to be applied. Make-accumulate itself takes an iterator as an argument. This describes how the function is to be applied to the structure. Thus, the function returned by make-accumulate is specific to the iterator and can be called with various functions and structures. It is, of course, necessary that the iterator be compatible with the structure and that the function be compatible both with the structure and initial value. Accumulate-for-each is an accumulating function created by calling make-accumulate with the iterator for-each.

Implementation:

```
(define ((make-accumulate iterator)
           function
           initial-value
           structure)
  (iterator
    (lambda (x)
      (set! initial-value (function initial-value x)))
    structure)
  initial-value)
```

MAKE-COLLECT-CONS

Format: ((MAKE-COLLECT-CONS iterator) function structure)

Parameters:
iterator - An iterator.
function - A function of one variable.
structure - A structure compatible with the function and iterator.

Explanation: Make-collect-cons uses make-accumulate to define an accumulating function (see make-accumulate) which returns a list containing the results of function applied to structure. Iterator specifies how the function is applied to the structure. Function, structure and iterator must all be compatible.

Usage: (define map (make-collect-cons for-each)) ==> map
       (define (list-copy list) (map identity))   ==> list-copy

Implementation:

```
(define ((make-collect-cons iterator) function structure)
  (reverse!
    ((make-accumulate iterator)
     (lambda (x y) (cons (function y) x))
     '()
     structure)))
```

MAKE-COLLECT-APPEND!

Format: ((MAKE-COLLECT-CONS iterator) function structure)

Parameters:
iterator - An iterator.
function - A function of one variable.
structure - A structure compatible with the function and iterator.

Explanation: Make-collect-append defines an accumulating function (see make-accumulate) which returns a list containing the results of function applied to structure. The function (of one variable) returns a list. Make-collect-append returns a single list containing all the elements in all the lists returned when the function is applied to all the elements in the structure. Iterator specifies how the function is applied to the structure; i.e., in which order the function is applied to the elements of the structure. Function, structure and iterator must all be compatible.

Usage: (define map-append! (make-collect-append! for-each))

Implementation:

```
(define ((make-collect-append! iterator) function structure)
  (reverse!
    ((make-accumulate iterator)
     (lambda (x y) (reverse-append! (function y) x))
     '()
     structure)))
```

FOR-EACH-INTEGER

Format: (FOR-EACH-INTEGER function n)
(GENERATE-LIST function n)
(GENERATE-VECTOR function n)

Parameters:
function - A function of one integer.
n - A non-negative integer.

Explanation: The function is applied to the integers from 0 to n-1. This is equivalent to using for-each on (iota n); i.e., on a list containing the integers from 0 to n-1.

```
Usage: (define (iota n) (generate-list identity n)) ==> iota
       (iota 5)                                ==> (0 1 2 3 4)
       (define (vector-iota n) (generate-vector identity n))
                                                  ==> vector-iota
       (vector-iota 6)                  ==> #(0 1 2 3 4 5)
       (generate-vector (lambda (x) (* x x)) 4) ==> #(0 1 4 9)
```

Implementation:

```
(define (for-each-integer function n)
  (let loop ((i 0))
    (when (< i n)
          (function i)
          (loop (1+ i)))))

(define generate-list
   (make-collect-cons for-each-integer))

(define (generate-vector function n)
  (let ((v (make-vector n)))
    (for-each-integer
      (lambda (i) (vector-set! v i (function i)))
      (vector-length v))
    v))
```

VECTOR-MAP!

STRING-MAP!

Format: (VECTOR-MAP! function v)
        (STRING-MAP! function v)

Parameters:
function - A function of one variable. In the case of string-map!, the variable should be a character and the function should return a character. In the case of vector-map!, the variable should be of the type of the elements of the vector.
v - A vector (string).

Explanation: The function is applied to each element of the vector (string) and that element is replaced by the result returned by the function. The function is applied to the elements in an unspecified order and returns an unspecified value.

```
Usage: (define n '#(1 2 3))   ==> n
       (vector-map! even? n)  ==> unspecified
       n                      ==> #(#!FALSE #!TRUE #!FALSE)
        (define s (string-map! char-upcase "aBcDefG"))
                              ==> unspecified
       s                      ==> "ABCDEFG"
```

Implementation:

```
(define (vector-map! function v)
  (for-each-integer
    (lambda (i)
      (vector-set! v i (function (vector-ref v i))))
    (vector-length v))
  v)

(define (string-map! function s)
  (for-each-integer
    (lambda (i)
      (string-set! s i (function (string-ref s i))))
    (string-length s))
  s)
```

VECTOR-FOR-EACH

STRING-FOR-EACH

Format: (VECTOR-FOR-EACH function v)
        (STRING-FOR-EACH function v)

Parameters:
function - A function of one variable. In the case of string-for-each, the variable must be a character. In the case of vector-for-each, the variable should be of the same type as the elements of the vector.

Explanation: These are analogues of for-each. The function is applied to each member of the vector or string in order. It returns an unspecified value.

```
Usage: (define v '#((1 3) (5 7)))   ==> v
       (vector-for-each car v)      ==> unspecified
       v                            ==> '#((1 3) (5 7))
       (vector-for-each
         (lambda (x) (set-car! x (+ (car x) (cadr x))))
         v)                         ==> unspecified
       v                            ==> #((4 3) (12 7))
       (string-for-each print "aBc") [prints: #\a #\B #\c]
                                          ==> unspecified
```

Implementation:

```
(define (vector-for-each function v)
  (for-each-integer
    (lambda (i) (function (vector-ref v i)))
    (vector-length v)))

(define (string-for-each function s)
  (for-each-integer
    (lambda (i) (function (string-ref s i)))
    (string-length s)))
```

VECTOR-MAP

Format: (VECTOR-MAP function v)

Parameters:
function - A function of one variable.
v - A vector containing elements suitable as arguments to the function.

Explanation: The function is applied to each element of the vector and a vector is returned containing the results of these functional applications.

```
Usage: (define (vector-copy v) (vector-map identity v))
                                                ==> vector-copy
        (define v #((a b) c))                    ==> v
       (define w (vector-copy v))               ==> w
        w                                                ==> #((a b) c)
       (vector-set! v 1 'x)                     ==> unspecified
       (set-car! (vector-ref v 0) 7)            ==> unspecified
       v                                        ==> #((7 b) x)
       w                                        ==> #((7 b) c)

(define (vector-map function v)
  (generate-vector
    (lambda (i) (function (vector-ref v i)))
    (vector-length v)))
```

MAKE-REDUCE

Format: (MAKE-REDUCE predicate reduction)

Parameters:
predicate - A predicate which returns true iff the structure passed to it is non-empty; e.g., if a list is not null.
reduction - A reduction operator.
function - A function of two variables.
structure - A structure.
identity - (optional argument) Result to return if the structure is empty.

Explanation: Make-reduce creates a reduction operator which works on empty data structures as well as non-empty ones given a predicate which works on non-empty structures. A reduction operator is one which applies a function to all the elements of a structure and returns the result. It may or may not be destructive of the structure. Make-reduce returns a function of two arguments with an optional third argument. If the structure is non-empty, the reduction returned by make-reduce is the same as the reduction passed to it. If the structure is empty, the reduction returned will return the identity argument passed to make-reduce (if such an argument is present) or the empty list (if no identity argument is present.) For more information, see the description of REDUCE below.

Usage: (see the definition of REDUCE, below)

Implementation:

```
(define ((make-reduce non-empty-predicate? non-empty-reduction)
         operation structure . identity)
  (cond ((non-empty-predicate? structure)
         (non-empty-reduction operation structure))
        ((pair? identity) (car identity))
        (else (operation))))
```

REDUCE

Format: (REDUCE operation list)

Parameters:
operation - A function of two variables. The function should return a value suitable as an argument to it.
list - A list containing elements suitable as arguments to the operation.

Explanation: Reduce applies the operation to all elements of the list and returns the result. The operation should be associative. Reduce returns the empty list if called with an empty list, regardless of what the operation itself would return if it were called with an empty list.

```
Usage: (define (mag+ x y) (+ (abs x) (abs y)))  ==> mag+
       (reduce mag+ '(1 -5 -4 7))               ==> 17
       (reduce mag+ '())                                ==> '()
       (+)                                         ==> 0
```

Implementation:

```
(define reduce
  (make-reduce
    pair?
    (lambda (operation list)
      (accumulate-for-each
         operation
         (car list)
         (cdr list)))))
```

APPLY-UNTIL

Format: (APPLY-UNTIL predicate function structure)

Parameters:
predicate - A function of one variable which returns true or false.
function - A function of one variable which returns a value suitable as an argument to the function.
structure - A data object suitable as an argument to the function.

Explanation: Apply-until tests the predicate on the structure. If the predicate is true, apply until returns the structure. If not, apply-until invokes itself with the value returned by the function. Thus, apply-until continues to invoke itself until the predicate returns true. The function may or may not be destructive of its operand.

```
Usage: (apply-until
         (lambda (x) (<? x 1.5))
         (lambda (x) (/ x 5))
         10)                                ==> 0.4
        (define a '(3 1 -4 -5 6))         ==> a
       (apply-until
         (lambda (x) (negative? (car x)))
         cdr
         a)                               ==> (-4 -5 6)
       a                                          ==> (3 1 -4 -5 6)
```

Implementation:

```
(define (apply-until predicate? function x)
  (if (predicate? x)
      x
      (apply-until predicate? function (function x))))
```

PARALLEL-REDUCE!

PAIRWISE-REDUCE!

Format: (PAIRWISE-REDUCE-NON-EMPTY-LIST! operation list)

(PAIRWISE-REDUCE operation list)

(PARALLEL-REDUCE operation list)

Parameters:
operation - A function of two variables. The list should contain elements suitable as arguments to this function and the function should itself return a value suitable as an argument to itself. The function should be associative.
list - A list, possibly empty.

Explanation: Parallel-reduce! is a reduction operation. It applies an operation on the elements of a list in parallel in a pairwise fashion; i.e., it applies the operation to the first two elements in the list, then to the next two elements in the list, etc. This leaves a list with half as many elements. Parallel-reduce! then works on the halved list, halving its size again. This is continued until a single element remains containing the value to be returned. Parallel-reduce! modifies the list it is passed and returns the result of the operation. After its invocation, the list passed as input contains a single element whose value is the result of applying the operation to all the elements of the original list (or is an empty list if the original list was empty.) On a single processor and for operations without side-effects, parallel reduction is similar to ordinary (sequential) reduction. However, for operations with side effects, in particular when intermediate results are saved, parallel reduction can give rise to much more efficient algorithms. Pairwise-reduce! carries out one round of parallel-reduce!, halving the list.

```
Usage: (define a '(2 5 8 11 13))          ==> a
       (pairwise-reduce! - a)             ==> (-3 -3 13)
       a                                         ==> (-3 -3 13)
       (define b '("ex" "c" "elle" "nt")) ==> b
       (parallel-reduce! string-append b) ==> "excellent"
       (car b)                            ==> "excellent"
```

Implementation:

```
(define (pairwise-reduce-non-empty-list! operation list)
  (for-each-cdr
    (lambda (x)
      (when (pair? (cdr x))
            (set-car! x (operation (car x) (cadr x)))
            (set-cdr! x (cddr x))))
    list)
  list)

(define pairwise-reduce!
  (make-reduce pair? pairwise-reduce-non-empty-list!))

(define parallel-reduce!
  (make-reduce
    pair?
    (lambda (operation list)
      (apply-until
        (lambda (x) (null? (cdr x)))
        (lambda (x)
          (pairwise-reduce-non-empty-list! operation x))
        list)
      (car list))))
```

VECTOR-REDUCE

Format: (VECTOR-REDUCE operation v)

Parameters:
operation - A function of two variables. The list should contain elements suitable as arguments to this function and the function should itself return a value suitable as an argument to itself. The function should be associative.
v - A non-empty vector.

Explanation: Vector-reduce is a reduction operator (see REDUCE). It takes a vector as input and returns the result of applying the operation to the elements of the vector.

```
Usage: (define v #(1 2 3 4))                  ==> v
       (vector-reduce + v)                  ==> 10
       (vector-reduce - v)                  ==> unspecified
       (vector-reduce + #())                  ==> error
```

Implementation:

```
(define vector-reduce
  (make-reduce
    (lambda (v) (>= (vector-length v) 0))
    (lambda (operation vector)
      ((make-accumulate
         (lambda (function v)
           (let ((length (vector-length v)))
             (do ((i 1 (1+ i)))
                 ((>= i length))
                 (function (vector-ref v i))))))
       operation
       (vector-ref vector 0)
       vector))))
```

MAKE-ITERATE-UNTIL

Format: ((MAKE-ITERATE-UNTIL predicate iterator . return-value)
           function
           structure)

Parameters:
predicate - A function of one variable which returns true or false.
iterator - A function of two variables. The first is a function and the second is a structure to iterate the function over.
return-value - (optional argument) A value to return if the predicate is not satisfied by any element of the structure.
function - The function to be used by the iterator.
structure - The structure for the iterator to work on.

Explanation: Make-iterate-until takes an ordinary iterator and a predicate and creates a new iterator. The new iterator applies the predicate to each element of the structure. If the predicate is true, the new iterator aborts and returns that element as its value. Otherwise, it applies the function to the element of the structure and continues execution. If the predicate returns false for all members of the structure, the iterator returns the return-value (if one was passed in) or the empty list.

Usage:
```
(define (iterate-on-non-zeros f s)
  ((make-iterate-until zero? map) f s))
                          ==> iterate-on-non-zeros
(iterate-on-non-zeros
  (lambda (x) (print (/ x)))
  '(2 -4 0 3))
                 [prints: 0.5 -0.25]
                 ==> 0
```

Implementation:

```
(define ((make-iterate-until predicate iterator . return-value)
         function structure)
  (call/cc (lambda (exit)
             (iterator (lambda (x)
                         (if (predicate x)
                             (exit x)
                             (function x)))
                       structure)
             (if return-value
                 (car return-value)
                 '())))))
```

MAKE-ITERATE-WHILE

Format: ((MAKE-ITERATE-WHILE predicate iterator . return-value)
           function
           structure)

Parameters:
predicate - A function of one variable which returns true or false.
iterator - A function of two variables. The first is a function and the second is a structure to iterate the function over.
return-value - (optional argument) A value to return if the predicate is not satisfied by any element of the structure.
function - The function to be used by the iterator.
structure - The structure for the iterator to work on.

Explanation: Make-iterate-while takes an ordinary iterator and a predicate and creates a new iterator. The new iterator applies the predicate to the first element of the structure. If the predicate is false, the new iterator returns the value returned by the function when called with that element as its argument. Otherwise, it applies the function to the next element of the structure and continues execution. If the predicate returns false for all members of the structure, the iterator returns the return-value (if one was passed in) or the empty list.

Usage: ((make-iterate-while positive? for-each "DONE")
           (lambda (x) (print (+ (* 2 x) 3)))
            '(9 7 4 2)) [prints: 21 17 11 7 3]
                                 ==> "DONE"

Implementation:

```
(define ((make-iterate-while predicate iterator . return-value)
         function structure)
  (call/cc (lambda (exit)
             (iterator (lambda (x)
                         (if (predicate x)
                             (function x)
                             (exit x)))
                       structure)
             (if return-value
                 (car return-value)
                 '()))))
```

MEMBER-IF

Format: (MEMBER-IF predicate list)

Parameters:
predicate - A function of one variable which returns true or false.
list - A list containing elements suitable as arguments to the predicate.

Explanation: Given a list and a predicate, returns the sublist starting with the first element which satisfies the predicate, or the empty list if no element in the list satisfies the predicate.

Usage: (member-if even? '(1 2 3)) ==> (2 3)

Implementation:

```
(define (member-if predicate? list)
  ((make-iterate-until
     (lambda (x) (predicate? (car x)))
     for-each-cdr)
   identity
   list))
```

FILTER

Format: (FILTER predicate list)

Parameters:
predicate - A function of one variable which returns true or false.
list - A list containing elements suitable as arguments to the predicate.

Explanation: Given a list and a predicate, returns the sublist containing all elements which satisfy the predicate, or the empty list if no element in the list satisfies the predicate.

```
Usage: (define a (iota 6))          ==> a
       (filter even? a)             ==> (0 2 4)
       a                                   ==> (0 1 2 3 4 5 6)
```

Implementation:

```
(define (filter predicate list)
  (map-append!
    (lambda (x)
      (if (predicate x)
          (cons x '())
          '()))
    list))
```

FILTER!

Format: (FILTER! predicate list)

Parameters:
predicate - A function of one variable which returns true or false.
list - A list containing elements suitable as arguments to the predicate.

Explanation: Given a list and a predicate, returns the sublist containing all elements which satisfy the predicate, or the empty list if no element in the list satisfies the predicate. Filter! modifies the input list deleting all elements which do not satisfy the predicate.

```
Usage: (define a '(1 2 3 4 5))   ==> a
       (filter! even? a)         ==> (2 4)
        a                         ==> (2 4)
```

Implementation:

```
(define (filter! predicate list)
  (let ((first (member-if predicate list)))
    (if first
        (apply-until
          (lambda (x) (null? (cdr x)))
          (lambda (x)
            (cond ((predicate (cadr x))
                   (cdr x))
                  (else
                    (set-cdr! x (cddr x))
                    x)))
          first))
    first))
```

Iterators

One of the central ideas of higher order programming is the idea of using higher order functional forms (functions that produce functions) in stead of using recursion (tail or otherwise).

We can implement a function that adds squareroots of all even numbers in an interval (a, b); but if we want to add square roots of all numbers in a list we shall need another program; and another one for vectors; and another one for heaps ...

We can simplify our life by introducing iterators, that are somewhat like universal quantifiers on data structures.

Simpliest class of functional forms are iterators iterator is a function that takes a structure and returns a function that takes a function f of one argument as its argument and applies f to every element of the structure.

Most primitive kind of iterators can be produced with

```
(define (primitive-iterator initial-value transform)
  (lambda (function)
    (define (loop x)
      (function x)
      (loop (transform x)))
    (loop initial-value)))
```

Sometimes the function we pass to the iterator is destructive and can affect x; to handle cases like that we define

```
(define (primitive-iterator! initial-value transform)
  (lambda (function)
    (define (loop x)
      ((lambda (next) (function x) (loop next))
       (transform x)))
    (loop initial-value)))
```

For example, we can iterate through natural numbers with

```
(define for-each-natural-number
  (primitive-iterator 1 1+))
```

Problem:

What will happen if you say (for-each-natural-number print)?

(before you try it find out Ctrl-Break on your keyboard).

Here you can ask what good does it do to have a non-terminating iterators. But we can make functions that starting with any iterator can produce other iterators out of it.

For example, restrict-iterator takes a predicate and an iterator and returns a new iterator which applies function only to those elements that satisfy the predicate

```
(define (restrict-iterator predicate iterator)
  (lambda (function)
    (iterator (when-combinator predicate function))))
```

And we can compose an iterator with a function

```
(define ((compose-iterator f iterator) g)
  (iterator (compose g f)))
```

And we can terminate the iteration with the following two iterator-manipulating functions:

```
(define (iterator-until predicate iterator marker)
  (lambda (function)
    (call-with-current-continuation
      (lambda (exit)
        (iterator (if-combinator predicate exit function))
        marker))))

(define (iterator-while predicate iterator marker)
  (lambda (function)
    (call-with-current-continuation
      (lambda (exit)
        (iterator (if-combinator predicate function exit))
        marker))))
```

Where call-with-current-continuation (or call/cc) is a function that ...

There is an "extra" feature in iterators created with iterator-until and iterator-while: in case of "unnatural" termination they return a value that caused it otherwise they return a marker

We can define a product of iterators

```
(define (product-of-iterators operation iterator1 iterator2)
  (lambda (function)
    (iterator1
      (lambda (x)
        (iterator2
          (lambda (y)
            (function (operation x y))))))))
```

First class continuations allow us to step through an iterator:

```
(define (make-step-iterator function iterator)
  (lambda (return)
    (iterator
      (lambda (x)
        (set! return
              (call-with-current-continuation
                (lambda (rest) (function x) (return rest))))))
    #!false))

(define (step-iterator iterator)
  (call-with-current-continuation
    (lambda (here)
      (iterator here))))

(define (sum-of-iterators operation iterator1 iterator2)
  (lambda (function)
    (let ((value1 '())
          (value2 '()))
      (let loop ((step1 (step-iterator
                          (make-step-iterator
                            (lambda (x) (set! value1 x))
                            iterator1)))
                 (step2 (step-iterator
                          (make-step-iterator
                            (lambda (x) (set! value2 x))
                            iterator2))))
        (cond ((and step1 step2)
               (function (operation value1 value2))
               (loop (step-iterator step1)
                     (step-iterator step2)))
              (step1 step1)
              (step2 step2)
              (else #!false))))))
```

```
(define (for-each-in-interval first last)
  (iterator-until
    (bind-1-of-2 < last)
    (primitive-iterator first 1+)
    'will-never-use-this-marker))
```

it would also be nice to implement reduction (reduction operator was introduced by Kenneth Iverson in APL)

```
(define (reduce iterator)
  (lambda (function . initial-value)
    (define (add-to x)
      (set! initial-value (function initial-value x)))
    (cond (initial-value
           (set! initial-value (car initial-value))
           (iterator add-to)
           initial-value)
          (else
           (let ((marker #!false))
             (define (first-time x)
               (set! initial-value x)
               (set! marker #!true)
               (set! first-time add-to))
             (iterator (lambda (x) (first-time x)))
             (if marker initial-value (function)))))))
```

where set! is a special form that changes a value of a binding

With all that we can give a new definition of factorial

```
(define (factorial n)
  ((reduce (for-each-in-interval 1 n)) *))
```

Problem

what does this function do:

```
(define (foo n)
  ((reduce
     (compose-iterator (compose / factorial)
(for-each-in-interval 0 n)))
   +))
```

?

Problem

implement a function that takes an iterator and computes a mean of elements through which iteration is done

Functional forms on lists

```
(define (for-each-cdr list)
  (iterator-while pair? (primitive-iterator list cdr) '()))

(define for-each-cdr
  (compose (bind-1-of-2 iterator-while pair?)
           (bind-2-of-2 primitive-iterator cdr)))

(define (for-each-cdr! list)
```

```
  (iterator-while pair? (primitive-iterator! list cdr) '()))

(define (for-each list)
  (compose-iterator car (for-each-cdr list)))

(define (map! list)
  (lambda (function)
    ((for-each-cdr list)
     (lambda (x) (set-car! x (function (car x)))))))

(define (reverse-append a b)
  ((reduce (for-each a)) (T-combinator cons) b))

(define (reverse-append! a b)
  ((reduce (for-each-cdr! a))
   (lambda (x y) (set-cdr! y x) y)
   b))


(define (vector-for-each-index v)
  (for-each-in-interval 0 (-1+ (vector-length v))))

(define (vector-for-each v)
  (compose-iterator (lambda (x) (vector-ref v x))
                    (vector-for-each-index v)))

(define (vector-map! v)
  (lambda (function)
    ((vector-for-each-index v)
     (lambda (i)
        (vector-set! v i (function (vector-ref v i)))))))

(define ((collect-cons iterator) function)
  (let ((header (list 9)))
    (set-cdr! header '())
    ((reduce iterator)
     rcons
     header)
    (cdr header)))

(define (map list)
  (collect-cons (for-each list)))

(define (list-copy list) ((map list) identity))

(define ((collect-append! iterator) function)
  (reverse!
    ((reduce iterator)
     (lambda (x y) (reverse-append! (function y) x))
     '())))

(define (map-append! list) (collect-append! (for-each list)))


(define (member-if predicate? list)
  ((iterate-until
     (compose predicate? car)
     (for-each-cdr list)
     '())
   identity))
```

```
(define (filter predicate list)
  ((collect-cons (restrict-iterator predicate (for-each list)))
   identity))

(define (filter! predicate list)
  ((collect-append! (restrict-iterator (compose predicate car)
                                       (for-each-cdr! list)))
   identity))
```

Tools for sorting study

```
(macro timer
  (lambda (x)
    (let ((exp (cadr x)))
      `(let ((time0 (runtime)))
         ((lambda () ,exp))
         (/ (- (runtime) time0) 100)))))

(define (random-list n . p)
  (if (null? p)
      (let loop ((i 1) (tail '()))
        (if (> i n)
            tail
            (loop (1+ i) (cons (%random) tail))))
      (let loop ((i 1) (tail '()) (p (car p)))
        (if (> i n)
            tail
            (loop (1+ i) (cons (random p) tail) p)))))

(define (random-vector n . p)
  (if (null? p)
      (do ((v (make-vector n))
           (i 0 (+ i 1)))
          ((>= i n) v)
          (vector-set! v i (%random)))
      (do ((p (car p))
           (v (make-vector n))
           (i 0 (+ i 1)))
          ((>= i n) v)
          (vector-set! v i (random p)))))

(define (iota n)
  (let loop ((i (-1+ n)) (tail '()))
    (if (< i 0)
        tail
        (loop (- i 1) (cons i tail)))))

(define (reverse-iota n) (reverse! (iota n)))

(define (random-iota n . p)
  (set! p (if (null? p) n (car p)))
  (let loop ((i (-1+ n)) (tail '()))
    (if (< i 0)
        tail
        (loop (-1+ i) (cons (+ i (random p)) tail)))))

(define (list-copy x) (append x '()))

(define (make-time-sort copy-function)
  (lambda (sort)
    (gc t)
    (let ((x (copy-function *test-list*)))
      (timer (sort  x >)))))

(define time-sort (make-time-sort list-copy))

(define time-vsort (make-time-sort list->vector))

(define (make-comp-count copy-function)
```

```
  (lambda (sort)
    (letrec ((comp-count0 0)
             (comp-count1 0)
             (comp (lambda (x y)
                     (cond ((> 16000 comp-count0)
                            (set! comp-count0 (1+ comp-count0)))
                           (else
                             (set! comp-count1 (1+ comp-count1))
                             (set! comp-count0 1)))
                     (> x y))))
      (sort (copy-function *test-list*) comp)
      (+ comp-count0 (* comp-count1 16000)))))

(define comp-count (make-comp-count list-copy))

(define v-comp-count (make-comp-count list->vector))

(define (make-test x) (set! *test-list* x)
*the-non-printing-object*)

(define *test-list* '())

(define (make-statistic function title-string)
  (lambda (sort length n)
    (do ((nl #\newline)
         (i 0 (1+ i))
         (l '()))
        ((>= i n)
         (for-each
           display
           (list
             "     " title-string nl
             "number of elements: " length nl
             "number of tests: " n nl
             "mean: " (mean l) nl
             "standard-deviation: "  (standard-deviation l) nl))
         *the-non-printing-object*)
        (make-test (random-list length))
        (set! l (cons (function sort) l)))))

(define statistic-comp-count
  (make-statistic comp-count "COUNTING COMPARISONS"))

(define statistic-v-comp-count
  (make-statistic v-comp-count "COUNTING COMPARISONS"))

(define statistic-time-sort
  (make-statistic time-sort "TIMING"))

(define statistic-time-vsort
  (make-statistic time-vsort "TIMING"))

(define (mean l)
  (let loop ((result 0) (n 0) (l l))
    (if (null? l)
        (/ result n)
        (loop (+ result (car l)) (1+ n) (cdr l)))))

(define (variance l)
  (let ((m (mean l)))
    (let loop ((result 0) (n -1) (l l))
```

```
      (if (null? l)
          (/ result n)
          (loop (+ result (let ((i (- (car l) m))) (* i i)))
                (1+ n)
                (cdr l))))))

(define (standard-deviation l) (sqrt (variance l)))

(define (average-deviation l)
  (let ((m (mean l)))
    (let loop ((result 0) (n 0) (l l))
      (if (null? l)
          (/ result n)
          (loop (+ result (abs (- (car l) m))) (1+ n) (cdr
l))))))
```

We shall first consider merge-sort. This will lead us to several new functional forms and allow us at first to produce a more efficient code for merge-sort itself and then to produce a new sorting algorithm which has some very unusual properties.

Recursive Merge-Sort.

The traditional version of merge-sort is based on the divide-and-conquer programming paradigm. First, we split the list of items in two halves, merge-sort them separately, and then merge them together. The following is the SCHEME translation of a COMMON LISP code from Winston and Horn:

```
(define (winston-sort x predicate)
  (define (merge a b)
    (cond ((null? a) b)
          ((null? b) a)
          ((predicate (car a) (car b))
           (cons (car a) (merge (cdr a) b)))
          (else
            (cons (car b) (merge a (cdr b))))))
  (define (head l n)
    (cond ((negative? n) '())
          (else (cons (car l) (head (cdr l) (- n 2))))))
  (define (tail l n)
    (cond ((negative? n) l)
          (else (tail (cdr l) (- n 2)))))
  (define (first-half l) (head l (- (length l) 1)))
  (define (last-half l) (tail l (- (length l) 1)))
  (cond ((null? (cdr x)) x)
        (else (merge (winston-sort (first-half x) predicate)
                     (winston-sort (last-half x) predicate)))))
```

Splitting linked lists in two is a time consuming activity. The same list is traversed twice at first by FIRST-HALF and then by SECOND-HALF, not counting two traversals by LENGTH.

Improving merge.

The traditional merge algorithm can be implemented thus:

```
(define (merge! l1 l2 predicate)
  (define (merge-loop l1 l2 last)
    (cond ((null? l1) (set-cdr! last l2))
          ((null? l2) (set-cdr! last l1))
          ((predicate (car l1) (car l2)) (set-cdr! last l1)
           (merge-loop (cdr l1) l2 l1))
          (else (set-cdr! last l2)
                (merge-loop l1 (cdr l2) l2))))
  (cond ((null? l1) l2) ;we do not need NULL tests for sorting
        ((null? l2) l1)
        ((predicate (car l1) (car l2))
         (merge-loop (cdr l1) l2 l1) l1)
        (else (merge-loop l1 (cdr l2) l2) l2)))

(define merge1!
  (let ((result (list '())))
    (lambda (l1 l2 predicate)
      (let loop ((l1 l1) (l2 l2) (last result))
        (cond ((null? l1) (set-cdr! last l2) (cdr result))
              ((null? l2) (set-cdr! last l1) (cdr result))
              ((predicate (car l1) (car l2)) (set-cdr! last l1)
               (loop (cdr l1) l2 l1))
              (else (set-cdr! last l2) (loop l1 (cdr l2)
l2)))))))
```

It can be seen that one of NULL? tests in MERGE-LOOP is unneeded. Only the list which was advanced during previous iteration can be empty. And we can keep this information around by putting the one which advanced as a first argument to the tail-recursive process which does the merging. That immediately allows us to reduce the number of pointer manipulations by a factor of two, since we need to do SET-CDR! only when the previous winner loses. All that allows us to come up with:

```
(define (unstable-merge! l1 l2 predicate)
  (define (merge-loop i j)
    (let ((k (cdr i)))
      (cond ((null? k) (set-cdr! i j))
            ((predicate (car k) (car j)) (merge-loop k j))
            (else (set-cdr! i j) (merge-loop j k)))))
  (cond ((null? l1) l2)
        ((null? l2) l1)
        ((predicate (car l1) (car l2))
         (merge-loop l1 l2) l1)
        (else (merge-loop l2 l1) l2)))
```

It can be easily seen that we can sort a list by first transforming it into a list of one element lists and then reducing merge on it:

```
(define (?-sort! l predicate)
  (reduce (lambda (x y) (merge! x y predicate)) (listify! l)))
```

where LISTIFY! is:

```
(define (listify! l) (map! list l))
```

And our ?-sort! sorts. But it sorts extremely slowly. This sequence of merges transforms merge-sort into insertion-sort.

It is now easy to see that what we need is another reduction operator. Instead of reducing the list from left to right (or from right to left - both orders are possible in COMMON LISP) we want to reduce the list in a tournament fashion - with logN rounds. We can do it with the help of the following two functional forms:

```
(define (pairwise-reduce! operation l)
  (let loop ((x l))
    (cond ((null? (cdr x)) l)
          (else (set-car! x (operation (car x) (cadr x)))
                (set-cdr! x (cddr x)) (loop (cdr x))))))

(define (parallel-reduce! operation l)
  (if (null? (cdr l)) (car l)
      (parallel-reduce! operation
                        (pairwise-reduce! operation l))))
```

PARALLEL-REDUCE! is an iterative analog of divide-and-conquer. When used with an associative operation, such as merge, it produces the same result as REDUCE, but very often more quickly. For non-associative operations it produces a different result, which may be valuable in itself and leads to new algorithms.

Now we can easily implement merge-sort:

```
(define (merge-sort! l predicate)
  (parallel-reduce! (lambda (x y) (merge! x y predicate))
           (listify! l)))
```

It can be seen that all the processes involved are iterative and all function calls can be easily removed. We generate exactly N extra conses. But the number of extra conses can be further reduced if LISTIFY! will make not a list of one element lists, but a list of sorted lists with 8 elements each created with the help of the insertion sort. While this can be done, this does not really improve the performance since LISTIFY! takes a very small percentage of total time declining when N grows.

```
(define (put-in-adder! x register function zero)
  (let ((y (car register)) (z (cdr register)))
    (cond ((eqv? y zero) (set-car! register x))
          (else (set-car! register zero)
                (set! x (function x y))
                (if (null? z) (set-cdr! register (list x))
                    (put-in-adder! x z function zero))))))
```

It can be used for many different things from simulating binary 1+ to implementing binomial queues.

We can now define a new version of merge-sort:

```
(define (adder-merge-sort! l predicate)
  (define register (list '()))
  (define (local-merge! x y) (merge! y x predicate))
  (define (local-put-in-adder! x)
    (set-cdr! x '())
```

```
    (put-in-adder! x register local-merge! '()))
  (for-each-cdr! local-put-in-adder! l)
  (reduce local-merge! register))
```

It generates logN conses, and is very quick.

```
(define (v-put-in-adder! x register function zero)
  ;;we assume that register is long and there will be no overflow
  (let loop ((x x) (i 0))
    (let ((y (vector-ref register i)))
      (cond ((eqv? y zero) (vector-set! register i x))
            (else (vector-set! register i zero)
                  (loop (function x y) (1+ i)))))))

(define v-adder-merge-sort!
  (let ((register (make-vector 32)))
    (lambda (l predicate)
      (define function (lambda (x y) (merge! y x predicate)))
      (vector-fill! register '())
      (for-each-cdr!
        (lambda (x)
          (set-cdr! x '())
          (v-put-in-adder! x register function '()))
        l)
      (vector-reduce function register))))



(define (make-mergesort! merge!)
  (lambda (l predicate)
    (parallel-reduce!
      (lambda (x y) (merge! x y predicate))
      (map! list l))))

(define mergesort! (make-mergesort! merge!))
```

and unstable-merge! makes it about 10% faster

```
(define unstable-mergesort! (make-mergesort! unstable-merge!))
```

hand-optimization of unstable-mergesort! gives us

```
(define (merge-sort! x predicate)
  (define (merge i j)
    (let ((k (cdr i)))
      (cond ((null? k) (set-cdr! i j))
            ((predicate (car k) (car j)) (merge k j))
            (else (set-cdr! i j) (merge j k)))))
  (do ((l x (cdr l)))
      ((null? l))
      (set-car! l (list (car l))))
  (do ()
      ((null? (cdr x)) (car x))
      (do ((l x (cdr l)))
          ((null? (cdr l)))
          (let ((i (car l))
                (j (cadr l)))
            (cond ((predicate (car i) (car j)) (merge i j))
                  (else (set-car! l j) (merge j i))))
          (set-cdr! l (cddr l)))))
```

```
(define (grab x y)
  (set-cdr! x (cons y (cdr x)))
  x)

(define (make-tournament-play predicate)
  (lambda (x y)
    (if (predicate (car x) (car y))
        (grab x y)
        (grab y x))))

(define (make-tournament initializer reduction)
  (lambda (forest predicate)
    (reduction
      (make-tournament-play predicate)
      forest)))

(define sequential-tournament! (make-tournament right-reduce!))

(define parallel-tournament! (make-tournament parallel-reduce!))

(define (make-tournament-sort! tournament1 tournament2)
  (lambda (plist predicate)
    (let ((p (tournament1 (map! list plist) predicate)))
      (for-each-cdr
        (lambda (x) (set-cdr! x (tournament2 (cdr x) predicate)))
        p)
      p)))

(define tournament-sort-p!
  (make-tournament-sort! parallel-tournament!
     parallel-tournament!))

(define tournament-sort-s!
  (make-tournament-sort! parallel-tournament!
     sequential-tournament!))

(define tournament-sort-s-s!
  (make-tournament-sort! sequential-tournament!
       sequential-tournament!))
```

```
(macro grab!
  (lambda (body)
    (let ((x (cadr body))
          (y (caddr body))
          (z (gensym))
          (w (gensym)))
      `(let ((,z ,x) (,w ,y))
         (set-cdr! ,w (cdar ,z))
         (set-cdr! (car ,z) ,w)
         ,z))))

(macro tournament-play!
  (lambda (body)
    (let ((x (cadr body))
          (y (caddr body))
          (predicate (cadddr body)))
      `(if (,predicate (caar ,x) (caar ,y))
           (grab! ,x ,y)
           (grab! ,y ,x)))))

(define (sequential-tournament! forest predicate)
  (cond
    ((null? forest) '())
    ((null? (cdr forest)) (car forest))
    (else
      (let ((x (reverse! forest)))
        (do ((result x (tournament-play! result next predicate))
           (next (cdr x) after-next)
           (after-next (cddr x) (cdr after-next)))
          ((null? after-next)
           (car (tournament-play! result next predicate)))))))))

(define (parallel-tournament! forest predicate)
  (define (tournament-round! so-far to-be-done)
    (cond ((null? to-be-done) so-far)
          ((null? (cdr to-be-done))
           (set-cdr! to-be-done so-far)
           to-be-done)
          (else
            (let* ((i (cdr to-be-done))
                   (j (cdr i))
                   (new (tournament-play! to-be-done
                                          i
                                predicate)))
              (set-cdr! new so-far)
              (tournament-round! new j)))))
  (if (null? forest)
      '()
      (do ((x forest (tournament-round! '() x)))
          ((null? (cdr x)) (car x)))))
```

VECTOR UTILITIES

(vector-last v) - returns the index of the last element in a vector.

(vector-swap! v i j) - interchanges the values of elements i and j in a vector.

(vector-reverse! v) - reverses a vector in place (destructively).

(vector-move! v to from) - move the value from element from to

```
        element to.

(vector-compare predicate v first second) - compare element
        first with element second using predicate.

(define-integrable (vector-last v)
  (-1+ (vector-length v)))

(define-integrable (vector-swap! v i j)
  (let ((temp (vector-ref v i)))
    (vector-set! v i (vector-ref v j))
    (vector-set! v j temp)))

(define (vector-reverse! v)
  (do ((first 0 (1+ first))
       (last (vector-last v) (-1+ last)))
      ((>= first last) v)
      (vector-swap! v first last)))

(define-integrable (vector-move! v to from)
  (vector-set! v to (vector-ref v from)))

(define-integrable (vector-compare predicate v first second)
  (predicate (vector-ref v first) (vector-ref v second)))
```

SIFTING

Sift is an algorithmic primitive which can be used to build a variety of sorting algorithms. It is a generalization of the bubbling operation in heaps. Given a vector, v, containing elements to be sorted, sift considers chains of elements. A chain is a sequence of elements whose indices in the vector are related functionally to one another. When bubbling up in an ordinary heap, for example, the next element in a chain has an index which is found by halving the current index. Sift also takes a value whose proper place within the chain is to be found. The proper place of a value within a chain is defined by a predicate, which is used to compare pairs of values. If (predicate a b) is satisfied, then a belongs ahead of b in the chain. Usually, the value passed to sift is a value already in the chain and currently out of place with respect to the predicate. Sift is invoked with this value and with a chain which is otherwise correct with respect to the predicate. After sifting, this value is in the correct place in the chain. Thus, a proper chain with one more element has been created. Starting with chains containing one element (which are trivially correct), sift is called to create larger chains which lead to a variety of structures useful in sorting. Examples of these are heaps (of many kinds), and partially sorted subsequences of elements. As we will see below, many variants of heapsort, shellsort, and selection sort can be created using sift.

(sift v position next-function value fill-pointer predicate) -
      v - vector containing values to be sorted.
      current - position in v where sift is to start.
      next-function - function which returns the position
      of the next element to be considered in the sift;
      returns null if current position is the last element
      to be considered.
      value - the value to be placed in v.
      fill-pointer - last occupied position in v.
      predicate - predicate indicating ordering desired by
        the sort; i.e., (predicate v[i] v[j]) is satisfied for
        i < j at the end of the sort.

 (sift-all! v step-function start fill-pointer predicate) -
      iteratively invokes sift starting from positions
      start,start-1,... 0. This can be used to set up a
      heap, do an insertion sort, or do one phase of Shellsort.

```
(define (sift! v current next-function value fill-pointer
               predicate)
 (let ((next (next-function v current fill-pointer predicate)))
  (cond ((or (null? next) (predicate value (vector-ref v next)))
         (vector-set! v current value))
        (else (vector-set! v current (vector-ref v next))
              (sift! v next next-function value fill-pointer
                     predicate)))))

(define (sift-all! v next-function start fill-pointer predicate)
 (do ((i start (- i 1)))
   ((< i 0) v)
   (sift! v i next-function (vector-ref v i) fill-pointer
          predicate)))
```

INSERTION SORT

To implement Insertion Sort using the sift primitive, we need only define an appropriate next-function.

(insertion-next step) - next-function for insertion sort. Also, suitable for implementing one phase of Shellsort. Generates next postion by adding a constant to current position.

(insertion-step-sort! v step predicate) - uses insertion-next and sift-all! to sort, or in the case of Sheelsort, to do one phase of a sort by sorting every step-th element in v.

(insertion-sort! v predicate) - Insertion Sort. Invokes insertion-step-sort! with step=1.

```
(define (insertion-step step)
 (lambda (v current fill-pointer predicate)
  (let ((next (+ current step)))
   (if (> next fill-pointer) '() next))))

(define (insertion-step-sort! v step predicate)
 (let ((l (vector-last v)))
  (sift-all! v (insertion-step step) (- l step) l predicate)))

(define (insertion-sort! v predicate)
 (insertion-step-sort! v 1 predicate))
```

SHELLSORT

Refs: D.E. Knuth, "The Art of Computer Programming,"
Vol. 3, "Sorting and Searching," pp. 84-95.
Donald L. Shell, CACM, Vol. 2, 1959, pp.30-32.
Collected Algorithms from CACM: Algorithm #201
Properties: Sorts vectors in place, not stable, partial sorting not possible, worst case complexity O[N^2], average case complexity varies and is in practice competitve with the best sorts.

Shellsort takes as input a vector of values to be sorted and a sequence of increments. These increments control the sorting process. Each increment is used in turn to define the distance between elements in the vector. Elements in the vector at this distance are considered as a chain (see the description of the sifting operation above) and are sorted. The final increment in the sequence is 1 and so at the end of Shellsort, the vector is totally sorted. Thus, Shellsort can be thought of as a series of insertion sorts. The purpose of the initial sorts in the sequence is to quickly bring elements to positions which are close to the proper positions for these elements so that each individual pass of the algorithm does not have to work too hard it is well known that insertion sort is very fast when the elements to be sorted do not have to move far. Picking a good sequence of increments is an art. We offer several good choices below.

```
(define (make-shellsort! increment-function)
  (lambda (v predicate)
    (for-each
      (lambda (step) (insertion-step-sort! v step predicate))
      (increment-function (vector-length v)))
  v))
```

INCREMENT SEQUENCES FOR SHELLSORT

The following are sequences shown to be good for Shellsort. (Reference: "Handbook of Algorithms and Data Structures", G. H. Gonnet Addison-Wesley, 1984)

(knuth-increments n) - function yielding the sequence recommended by Knuth in his book. n is the number of elements in the vector of elements to be sorted. The sequence generated is (...., 40, 13, 4, 1). The sequence is generated starting with the value 1 at the end of the sequence. The next (i.e., preceding) value is generated from the current one by multiplying by 3 and adding 1. The final (first) element in the sequence is the largest such number which is less than n.

(shellsort-knuth! v predicate) - Shellsort using Knuth increments.

(pratt-increments n) - increments by shown by Pratt to guarantee O[n * (log (n)^2)] worst case preformance but very slow in practice. Elements of the sequence are composites of powers of 2 and powers of 3. For example if n is 50, the sequence is (48,36,32,27,24,18,16,12,9,6,4,3,2,1).

(shellsort-pratt! v predicate) - Shellsort using Pratt increments.

(gonnet-increments n) - increments recommended by Gonnet in his

```
            book. The sequence is generated by starting with
            floor(.4545n) and continuing to take floor(.4545i)
            until 1 is reached.

(shellsort-gonnet! v predicate) - Shellsort using Gonnet
increments.

(stepanov-increments n) - increments recommended by A. Stepanov.
         The sequence is generated by taking floor(e^i + .5);
         i.e., powers of e rounded to the nearest integer. Again,
         the sequence is generated in reverse order and ends with
         the largest such value less than n. These increments are
         the most efficient ones we have found thus far.
(shellsort-stepanov! v predicate) - Shellsort using Stepanov
                                    increments.

(define (knuth-increments n)
  (do ((i 1 (+ (* i 3) 1))
       (tail '() (cons i tail)))
      ((>= i n) (or (cdr tail) tail))))

(define shellsort-knuth! (make-shellsort! knuth-increments))

(define (pratt-increments n)
  (define (powers base n)
    (do ((x 1 (* x base))
         (result '() (cons x result)))
        ((>= x n) result)))
  (filter (lambda (x) (< x n))
          (parallel-reduce!
            (lambda (x y) (merge! x y >))
            (outer-product * (powers 2 n) (powers 3 n)))))

(define shellsort-pratt! (make-shellsort! pratt-increments))
```

```
(define (gonnet-increments n)
  (define (gonnet n) (floor (* n .45454)))
  (do ((i (gonnet n) (gonnet i))
       (result '() (cons i result)))
      ((>= 1 i) (reverse! (cons 1 result)))))

(define shellsort-gonnet! (make-shellsort! gonnet-increments))

(define (stepanov-increments n)
  (do ((i 1 (+ i 1))
       (e 1 (floor (+ 0.5 (exp i))))
       (tail '() (cons e tail)))
      ((>= e n) tail)))

(define shellsort-stepanov! (make-shellsort!
stepanov-increments))
```

HEAPS USING SIFTING

Heaps can also be implemented using the sift primitive, inclusing an entire family of Heapsort algorithms. These algorithms also use some of the vector utilities described above. All of the heap utilities implemented above are reimplemented here using the same names for the functions. Thus, if this entire file is loaded and compiled, these are the functions which will be used, since they the last (most recent) ones defined.

next-functions for sift:

(heap-son v father fill-pointer predicate)

- This is a next-function for sift. Given father, a position in the vector (v, fill-pointer, and predicate are as above in the description of sift) it returns the position of the "larger" successor of father. Thus, if father = i, it returns the false value if 2i+2 is greater than n. (Recall that our vectors are indexed starting from 0; thus a vector of n elements has elements with indices 0,1,...n-1 and the children of an element with index i are those with indices 2i+1 and 2i+2.) It returns 2i+1 if (predicate v[2i+1] v[2i+2]) is true or if 2i+3 is greater than n; and it returns 2i+2 if (predicate v[2i+1] v[2i+2]) is false. This is the appropriate next-function for bubbling down in ordinary heaps.

(heap-up-pointer son) - floor((son-1)/2)

(heap-father v son fill-pointer predicate) - The appropriate next-function for bubbling up in an ordinary heap. It returns (heap-up-pointer son) if son is positive and the false value otherwise.

```
(define (heap-son v father fill-pointer predicate)
  (let ((son (* 2 (1+ father))))
    (cond ((>= fill-pointer son)
           (if (predicate (vector-ref v son)
                          (vector-ref v (-1+ son)))
               son
               (-1+ son)))
          ((= fill-pointer (-1+ son)) (-1+ son))
          (else '()))))

(define (heap-up-pointer son) (quotient (-1+ son) 2))

(define (heap-father v son fill-pointer predicate)
  (if (>= 0 son) '() (heap-up-pointer son)))
```

```
(define (downheap! v father value fill-pointer predicate)
  (sift! v father heap-son value fill-pointer predicate))

(define (upheap! v son value predicate)
  (sift! v son heap-father value son
         (lambda (x y) (predicate y x))))

(define (build-heap! v fill-pointer predicate)
  (sift-all! v heap-son (heap-up-pointer fill-pointer)
  fill-pointer predicate))

(define (heap-set! v position value fill-pointer predicate)
  (if (predicate (vector-ref v position) value)
      (downheap! v position value fill-pointer predicate)
      (upheap! v position value predicate)))
```

HEAPSORT

Williams' Heapsort Algorithm
Refs: Knuth Volume 3 , p. 145-149
Collected Algorithms from CACM: Algorithm #232
CACM, Vol. 7 (1964) pp. 347-348
Properties: sorts vectors in place, not stable, partial sort possible, worst case running time O[N*log(N)].

Heapsort works by setting up a heap. A heap is a binary tree with the following properties. The descendents of node i are nodes 2i and 2i+1. Thus, the links pointing to the descendents of a node are implicit in the nodes' positions in the vector. A node satisfies the predicate (passed as an argument to heapsort) with respect to all its descendents. Thus, for example, if the predicate is <, each node is less than all its descendents. Heapsort begins by building a heap (using build-heap). The heap is built by checking that the predicate is satisfied and interchanging a node with its smaller (in the sense of the predicate) descendent if necessary, so that after the exchange the predicate is satisfied. Traditionally, for the sake of efficiency, the heap is built upside down, in reverse order of the predicate. Here, for clarity, the heap is built right side up. The function of "bubbling down an element, in some cases several levels in the heap, until the predicate is satisfied or the element reaches the bottom of the heap, is handled by downheap. After the heap is set up, the element which should be in the first position in the sorted vector is at the top of the heap (in position 1). The first and last element in the heap are interchanged and the last element is removed from further consideration by decreasing the size of the heap. The new top heap element (taken from the bottom of the heap in the above exchange) is bubbled down. The process of exchange and bubbling is repeated until the entire vector is sorted. At this point, the vector in in reverse order, so reverse! is called to put the vector in the desired sorted order.

(heapsort! v predicate) - Heapsort. v is the vector to be sorted using the predicate.

(read-heap! v fill-pointer predicate) - pop all the elements out of the heap in order.=@

HEAPSORT USING SIFTING

(heapsort! v predicate) - Heapsort. See description above.
This is the traditional version of Heapsort. The heap is built in reverse order of the predicate, which allows the read operation to pop out the elements in reverse ordr and then place them in their proper positions in the sorted vector when the popped element and the last element in the heap are interchanged.

(read-heap! v fill-pointer predicate) - pop all the elements out of a heap. See description above.

(reverse-heapsort! v predicate) - This is the more natural version of Heapsort, as described in the section above. The heap is built in the natural order and the sorted list is reversed at the end of the sort.

(top-down-build-heap! v fill-pointer predicate) - The heap can be built from the top down. This is useful if the elements are not all available at the time the heap is originally being formed. This has worst case complexity O[nlog(n)].

(top-down-heapsort! v predicate) - Heapsort using top-down-build-heap.

```
(define (read-heap! v fill-pointer predicate)
  (do ((position fill-pointer (-1+ position)))
      ((>= 0 position) v)
      (vector-swap! v position 0)
      (downheap! v 0 (vector-ref v 0) (-1+ position) predicate)))

(define (heapsort! v predicate)
  (build-heap! v (vector-last v) (lambda (x y) (predicate y x)))
  (read-heap! v (vector-last v) (lambda (x y) (predicate y x))))

(define (reverse-heapsort! v predicate)
  (build-heap! v (vector-last v) predicate)
  (read-heap! v (vector-last v) predicate)
  (vector-reverse! v))
```

TOP-DOWN-BUILD-HEAP Top-down-build-heap! allows us to build a heap one element at a time. It is O[N*log(N)] in the worst case and O[N] on the average. We can also implement heapsort with top-down-build-heap!

```
(define (top-down-build-heap! v fill-pointer predicate)
  (do ((position 1 (1+ position)))
      ((> position fill-pointer) v)
      (upheap! v position (vector-ref v position) predicate)))

(define (top-down-heapsort! v predicate)
  (top-down-build-heap! v (vector-last v) predicate)
  (read-heap! v (vector-last v) predicate)
  (vector-reverse! v))
```

3-HEAPS 3-heaps are slightly faster (3% fewer comparisons and 2% less time) than ordinary heaps (2-heaps). In 3-heaps, each non-terminal node has up to 3 children. This results in a shallower tree but requires an additional comparison per level. Of all the possible breadths of heaps, we found 3-heaps to be the best. Note that this section redefines the functions heap-son and heap-up-pointer and should not be loaded unless

you intend to use 3-heaps instead of ordinary heaps.

```
(define (heap-son v father fill-pointer predicate)
  (define (test i j)
    (predicate (vector-ref v i) (vector-ref v j)))
  (let ((son (* 3 (1+ father))))
    (cond ((>= fill-pointer son)
           (if (test son (- son 1))
               (if (test son (- son 2)) son (- son 2))
               (if (test (- son 1) (- son 2))
                   (- son 1)
                   (- son 2))))
          ((= fill-pointer (-1+ son))
           (if (test (- son 1) (- son 2)) (- son 1) (- son 2)))
          ((= fill-pointer (- son 2)) (- son 2))
          (else '()))))

(define (heap-up-pointer son) (quotient (-1+ son) 3))
```

D-HEAPS

Using sifting, d-heaps (heaps with d successors per node) can be implemented. This is useful in order to carry out experiments on the relative efficiency of different values of d, which is interesting in the case where there are additions, deletions and changes in value of the vector elements. It is possible, by giving some nodes d children and other d+1 children to form d-heaps for non-integer values of d. We do not do this here, however.

(largest-in-the-range v first last predicate) - returns the largest element between position first and position last, where v[i] is largest if (predicate v[i] v[j]) is true for all j in the range.

(make-d-heap-son d) - returns a heap-son function for a d-heap. For example (define heap-son (make-d-heap-son 4)) sets up the heap-son function for a 4-heap.
(make-d-heap-up-pointer d) - returns a heap-up-pointer function for a d-heap.

```
(define (largest-in-the-range v first last predicate)
  (if (> first last) '()
      (do ((next (1+ first) (1+ next)))
          ((> next last) first)
          (if (predicate (vector-ref v next)
                         (vector-ref v first))
              (set! first next)))))

(define (make-d-heap-son d)
  (lambda (v father fill-pointer predicate)
    (let ((x (* d father)))
      (largest-in-the-range
        v (+ x 1) (min (+ x d) fill-pointer) predicate))))

(define (make-d-heap-up-pointer d)
  (lambda (son) (quotient (-1+ son) d)))

(define (selection-sort! v predicate)
  (do ((last (vector-last v))
       (i 0 (1+ i)))
```

```
((>= i last) v)
(vector-swap! v i
              (largest-in-the-range v i last predicate))))
```

```
(macro make-encapsulation
  (lambda (body)
    (let ((parameters (cadr body))
          (variables (caddr body))
          (local-procedures (cadddr body))
          (methods (car (cddddr body))))
      `(lambda ,parameters
         (let* ,variables
           (letrec ,(append local-procedures methods)
             (let ((list-of-methods
                     (list . ,(map (lambda (x)
                                     `(cons ',(car x) ,(car x)))
                                   methods))))
               (lambda (message)
                 (let ((method (assq message list-of-methods)))
                   (if (null? method)
                       (error
"no such method in this encapsulation: " message)
                       (cdr method)))))))))))

(macro old-use-methods
  (lambda (body)
    `(let ,(map (lambda (x)
                  (if (pair? x)
                      `(,(car x) (,(cadr body) ',(cadr x)))
                      `(,x (,(cadr body) ',x))))
                (caddr body))
       . ,(cdddr body))))

(macro use-methods
  (lambda (body)
    (define (clause-parser clause)
      (map (lambda (x)
                  (if (pair? x)
                      `(,(car x) (,(car clause) ',(cadr x)))
                      `(,x (,(car clause) ',x))))
                (cadr clause)))
    `(let ,(map-append! clause-parser (cadr body))
       . ,(cddr body))))

(define (make-encapsulation-iterator encapsulation)
  (let ((pop! (encapsulation 'pop!))
        (empty? (encapsulation 'empty?)))
    (lambda (function)
      (do ()
          ((empty?))
          (function (pop!))))))



(define make-stack
  (make-encapsulation
    ()
    ((s '()))
    ((check-underflow
      (lambda () (if (empty?) (error "stack underflow")))))
    ((push! (lambda (item)
              (set! s (cons item s))
              *the-non-printing-object*))
     (pop! (lambda ()
             (check-underflow)
```

```
             (let ((temp (car s)))
               (set! s (cdr s))
               temp)))
      (top (lambda ()
             (check-underflow)
             (car s)))
      (empty? (lambda () (null? s)))
      (size (lambda () (length s))))))

(define make-vector-stack
  (make-encapsulation
    (n)
    ((v (make-vector n))
     (position -1)
     (last (-1+ n)))
    ()
    ((push (lambda (item)
             (if (>= position last) (error "stack overflow"))
             (set! position (1+ position))
             (vector-set! v position item)
             '()))
     (pop (lambda () (if (< position 0)
                         (error "stack underflow"))
             (let ((temp position))
               (set! position (-1+ position))
               (vector-ref v temp))))
     (top (lambda ()
             (if (< position 0) (error "stack underflow"))
             (vector-ref v position)))
     (empty? (lambda () (< position 0)))
     (full? (lambda () (= position last)))
     (size (lambda () (1+ position))))))
```

```
(define make-graph
  (make-encapsulation
    (n)
    ((v (generate-vector (lambda (i) (make-vector 3 '())) n)))
    ((node-ref
       (lambda (node i) (vector-ref (vector-ref v node) i)))
     (node-set!
       (lambda (node i value)
         (vector-set! (vector-ref v node) i value))))
    ((number-of-nodes (lambda () n))
     (for-each-node
       (lambda (function) (for-each-integer function n)))
     (self-print (lambda () (vector-for-each print v)))
     (self (lambda () v))
     (label (lambda (node) (node-ref node 0)))
     (set-label! (lambda (node value) (node-set! node 0 value)))
     (predecessor (lambda (node) (node-ref node 1)))
     (set-predecessor!
       (lambda (node value) (node-set! node 1 value)))
     (adjacency-list (lambda (node) (node-ref node 2)))
     (first-node (lambda (link) (vector-ref link 0)))
     (second-node (lambda (link) (vector-ref link 1)))
     (link-length (lambda (link) (vector-ref link 2)))
     (reverse-link
       (lambda (link)
         (vector
           (vector-ref link 1)
           (vector-ref link 0)
           (vector-ref link 2))))
     (add-directed-link
       (lambda (link)
         (let ((node1 (first-node link)))
           (vector-set! (vector-ref v node1) 2
                        (cons link (adjacency-list node1))))))
     (add-undirected-link
       (lambda (link)
         (let ((node1 (first-node link))
               (node2 (second-node link)))
           (vector-set! (vector-ref v node1) 2
                        (cons link (adjacency-list node1)))
           (vector-set! (vector-ref v node2) 2
                        (cons (reverse-link link)
                              (adjacency-list node2))))))
     (for-each-link-of-node
       (lambda (function node)
         (for-each function (adjacency-list node)))))))
```

Random Graph Generators

```
(define (random-edge n length)
  (let loop ((i (random n))
             (j (random n)))
    (if (= i j)
        (loop (random n) (random n))
        (vector i j (random length)))))

(define (d-graph n m . r)
  (let* ((r (if (null? r) 100 (car r)))
```

```
          (graph (make-graph n))
          (add (graph 'add-directed-link))
          (add-random-link
            (lambda (x) (add (random-edge n r)))))
    (for-each-integer add-random-link m)
    graph))

(define (u-graph n m . r)
  (let* ((r (if (null? r) 100 (car r)))
          (graph (make-graph n))
          (add (graph 'add-undirected-link))
          (add-random-link
            (lambda (x) (add (random-edge n r)))))
    (for-each-integer add-random-link m)
    graph))
```

Make a scan-based algorithm. This includes Bellman's Algorithm.

Arguments:
make-data-structure
value-function
better?

```
(define (make-scan-based-algorithm
          make-data-structure value-function better?)
  (lambda (graph root)
    (use-methods
      ((graph (set-label! set-predecessor!
               second-node link-length
               for-each-node for-each-link-of-node
               number-of-nodes)))
      (let* ((encapsulation
              (make-data-structure (number-of-nodes) better?))
             (push!? (encapsulation 'push!?))
             (label (encapsulation 'v-ref))
             (iterate-pop!
              (make-encapsulation-iterator encapsulation)))
        (for-each-node (lambda (x) (set-predecessor! x '())))
        (push!? root 0)
        (iterate-pop!
          (lambda (node)
            (for-each-link-of-node
              (lambda (link)
                (let ((new-node (second-node link)))
                  (if (push!?
                        new-node
                        (value-function (label node)
                                        (link-length link))))
                  (set-predecessor! new-node link))))
             node)))
        (for-each-node
          (lambda (node) (set-label! node (label node)))))))))
```

Make a scan-based algorithm with node marking.
This includes Dijkstra's and Prim's algorithms.

Arguments:
make-data-structure
value-function
better?

```
(define (make-scan-based-algorithm-with-mark
          make-data-structure value-function better?)
  (lambda (graph root)
```

```
    (use-methods
      ((graph (set-label! set-predecessor!
               second-node link-length
               for-each-node for-each-link-of-node
               number-of-nodes)))
      (let* ((encapsulation
              (make-data-structure (number-of-nodes) better?))
             (push!? (encapsulation 'push!?))
             (label (encapsulation 'v-ref))
             (iterate-pop!
              (make-encapsulation-iterator encapsulation))
             (mark (make-vector (number-of-nodes) 'unscanned)))
        (for-each-node (lambda (x) (set-predecessor! x '())))
        (push!? root 0)
        (iterate-pop!
          (lambda (node)
            (vector-set! mark node 'scanned)
            (for-each-link-of-node
              (lambda (link)
                (let ((new-node (second-node link)))
                  (if (and (eqv? (vector-ref mark new-node)
                                 'unscanned)
                           (push!?
                             new-node
                             (value-function
                               (label node)
                               (link-length link))))
                      (set-predecessor! new-node link))))
              node)))
        (for-each-node
          (lambda (node) (set-label! node (label node)))))))
```

Specific Algorithms

```
(define bellman
  (make-scan-based-algorithm
    make-vector-deque-with-values     ;make-data-structure
    +                                 ;value-function
    < ))                              ;predicate

(define dijkstra
  (make-scan-based-algorithm
    make-heap-with-membership-and-values     ;make-data-structure
    +                                        ;value-function
    < ))                                     ;predicate

(define dijkstra-m
  (make-scan-based-algorithm-with-mark
    make-heap-with-membership-and-values     ;make-data-structure
    +                                        ;value-function
    < )                                      ;predicate

(define prim
  (make-scan-based-algorithm-with-mark
```

```
make-heap-with-membership-and-values     ;make-data-structure
(lambda (x y) y)                         ;value-function
< ))                                     ;predicate
```

Vector which only allows storage of improved values

```
(define make-vector-with-predicate
  (make-encapsulation
    (n predicate)
    ((v (make-vector n 'empty)))
    ()
    ((set!? (lambda (index value)
         (cond ((or (eqv? (vector-ref v index) 'empty)
                    (predicate value (vector-ref v index)))
                (vector-set! v index value)
                #!TRUE)
               (else
                 #!FALSE))))
     (ref (lambda (index) (vector-ref v index)))
     (values (lambda () v)))))
```

Deque implemented using a vector

```
(define make-vector-deque
  (make-encapsulation
    (n)
    ((v (make-vector n))
     (number-of-nodes 0)
     (front 0)
     (rear 0)
     (last (-1+ n)))
    ((check-overflow
       (lambda () (if (full?) (error "deque overflow"))))
     (check-underflow
       (lambda () (if (empty?) (error "deque underflow"))))
     (increase-nodes! (lambda ()
                        (check-overflow)
                        (set! number-of-nodes
                              (1+ number-of-nodes))))
     (decrease-nodes! (lambda ()
                        (check-underflow)
                        (set! number-of-nodes
                              (-1+ number-of-nodes)))))
    ((full?
       (lambda () (= number-of-nodes n)))
     (empty?
       (lambda () (= number-of-nodes 0)))
     (in-rear! (lambda (value)
                 (increase-nodes!)
                 (vector-set! v rear value)
                 (set! rear (if (= rear last) 0 (1+ rear)))
                 *the-non-printing-object*))
     (in-front! (lambda (value)
                  (increase-nodes!)
                  (set! front (if (= front 0) last (-1+ front)))
                  (vector-set! v front value)
                  *the-non-printing-object*))
     (out-front! (lambda ()
                   (decrease-nodes!)
                   (let ((temp front))
                     (set! front
                           (if (= front last) 0 (1+ front)))
                     (vector-ref v temp))))
     (out-rear! (lambda ()
                  (decrease-nodes!)
                  (set! rear (if (= rear 0) last (-1+ rear)))
                  (vector-ref v rear)))
     (peek-front (lambda ()
                   (check-underflow)
                   (vector-ref v front)))
     (peek-rear (lambda ()
                  (check-underflow)
                  (vector-ref v (if (= rear 0)
                                    last
                                    (-1+ rear)))))
     (length (lambda () number-of-nodes)))))
```

Deque implemented with a vector-with-predicate

```
(define make-vector-deque-with-values
  (make-encapsulation
    (n predicate)
    ((v (make-vector-with-predicate n predicate))
     (queue (make-vector-deque n))
     (in-q (make-vector n 'never-was-in)))
    ((v-set!? (v 'set!?))
     (in-front! (queue 'in-front!))
     (in-rear! (queue 'in-rear!))
     (out-front! (queue 'out-front!)))
    ((push!?
       (lambda (index value)
         (cond ((v-set!? index value)
                (case (vector-ref in-q index)
                  (never-was-in (in-rear! index))
                  (was-in (in-front! index)))
                (vector-set! in-q index 'in)
                #!TRUE)
               (else #!FALSE))))
     (pop!
       (lambda ()
         (let ((value (out-front!)))
           (vector-set! in-q value 'was-in)
           value)))
     (v-ref (v 'ref))
     (empty? (queue 'empty?)))))
```

```
 Heap which keeps track of which elements
 of a fixed set are currently members.


(define make-heap-with-membership
  (make-encapsulation
    (n predicate)
    ((v (make-vector n))
     (member-v (make-vector n '()))
     (fill-pointer -1))
    ((heap-set!
       (lambda (index value)
         (vector-set! v index value)
         (vector-set! member-v value index)))
     (sift!
       (lambda (current step-function value predicate)
         (let ((next (step-function current)))
           (cond ((or (null? next)
                      (predicate value (vector-ref v next)))
                  (heap-set! current value))
                 (else
                  (heap-set! current (vector-ref v next))
                  (sift! next step-function value predicate))))))
     (heap-son
       (lambda (father)
         (let ((son (* 2 (1+ father))))
           (cond ((>= fill-pointer son)
                  (if (predicate (vector-ref v son)
                                 (vector-ref v (-1+ son)))
                      son
                      (-1+ son)))
                 ((= fill-pointer (-1+ son)) (-1+ son))
                 (else '()))))))
     (heap-father
       (lambda (son)
         (if (>= 0 son) '() (quotient (-1+ son) 2))))
     (downheap!
       (lambda (father value)
         (sift! father heap-son value predicate)))
     (upheap!
       (lambda (son value)
         (sift! son heap-father value
                (lambda (x y) (predicate y x))))))
    ((empty? (lambda () (= fill-pointer -1)))
     (push!
       (lambda (value)
         (let ((index (vector-ref member-v value)))
           (cond ((null? index)
                  (set! fill-pointer (1+ fill-pointer))
                  (upheap! fill-pointer value))
                 (else (upheap! index value))))))
     (pop!
       (lambda ()
         (let ((temp (vector-ref v 0)))
           (vector-set! member-v temp '())
           (set! fill-pointer (-1+ fill-pointer))
           (downheap! 0 (vector-ref v (1+ fill-pointer)))
           temp))))))
```

Heap with membership implemented using
a vector-with predicate.

```
(define make-heap-with-membership-and-values
  (make-encapsulation
    (n predicate)
    ((v (make-vector-with-predicate n predicate))
     (ref (v 'ref))
     (heap (make-heap-with-membership
             n
             (lambda (x y) (predicate (ref x) (ref y))))))
    ((v-set!? (v 'set!?))
     (push! (heap 'push!)))
    ((push!?
       (lambda (index value)
         (cond ((v-set!? index value)
                (push! index)
                #!TRUE)
               (else #!FALSE))))
     (pop! (heap 'pop!))
     (v-ref ref)
     (empty? (heap 'empty?)))))
```