

Concept of pairs would have limited utility if it only combined two elements to create a compound entity.

The ability to create pairs whose elements are pairs is the essence of list structure's importance as a representational tool.

We refer to this as the *closure property* of cons. In general, an operation for combining data objects satisfies the closure property if the result of combining things with that operation can themselves be combined using the same operation. Closure is the key to power in any means of combination because it permits us to create hierarchical structures.

Concept of pairs would have limited utility if it only combined two elements to create a compound entity.

The ability to create pairs whose elements are pairs is the essence of list structure's importance as a representational tool.

We refer to this as the *closure property* of cons. In general, an operation for combining data objects satisfies the closure property if the result of combining things with that operation can themselves be combined using the same operation. Closure is the key to power in any means of combination because it permits us to create hierarchical structures.

This closure is being used in mathematical sense (say real numbers are closed under multiplication, but not under square root), unlike the closure which was associated with environments of a procedure. An unfortunate accidental overloading of terminology.

How does one design procedures to take arbitrary number of arguments?

The closure property of Pairs allows us to provide multiple operands to a procedure.

Another way to supply multiple operands to a procedure is to use `define` with **dotted-tail** notation.

How does one design procedures to take arbitrary number of arguments?

The closure property of Pairs allows us to provide multiple operands to a procedure.

Another way to supply multiple operands to a procedure is to use `define` with **dotted-tail** notation.

In a procedure definition, a parameter list that has a dot before the last parameter name *implies*, when the procedure is called,

- ▶ the initial parameters (if any) will have as values the initial arguments
- ▶ the final parameter's (i.e the one after the dot) value will be a list of any remaining arguments.

```
; This procedure performs summation of a sequence of numbers  
; which could possibly be empty
```

```
(define (plus . x)  
  (define (iterator x sum)  
    (if(equal? x '())  
        sum  
        (iterator (cdr x) (+ sum (car x)))))  
  )  
  )  
  (iterator x 0)  
)
```

The `plus` procedure takes zero or more arguments and returns the sum of all the operands. If there are no numbers supplied, then zero is returned, otherwise if just one number *a* is supplied then *a* is returned.

```
; This procedure multiplies two or more numbers  
; If the procedure has less than two operands, it should throw an error.
```

```
(define (multiply x y . z)  
  (define (iterative-multiplier x prod)  
    (if (equal? x '())  
        prod  
        (iterative-multiplier (cdr x) (* prod (car x)))))  
  )  
  (iterative-multiplier z (* x y))  
)
```

The `multiply` procedure takes two or arguments and returns the product of all the arguments. However if less than two arguments are supplied to the procedure then it throws an error.

```

29 ; Test cases
30
31 (display "This examples showcases the utility of dotted-tail notation \n\n")
32 (display "(plus 4 2 1) is " ) (plus 4 2 1)
33 (display "(plus 4) is " ) (plus 4)
34 (display "(plus) is " ) (plus)
35
36 (newline)
37
38
39 (display "(multiply 3 4 5) evaluates to " ) (multiply 3 4 5)
40 (display "(multiply 3 6) evaluates to " ) (multiply 3 6)
41 (display "(multiply 3) evaluates to " ) (multiply 3)

```

Welcome to [DrRacket](#), version 8.7 [cs].

Language: `scheme`, with debugging; memory limit: 128 MB.

This examples showcases the utility of dotted-tail notation

```

(plus 4 2 1) is 7
(plus 4) is 4
(plus) is 0



```

```

(multiply 3 4 5) evaluates to 60
(multiply 3 6) evaluates to 18

```

```

(multiply 3) evaluates to   multiply: arity mismatch;
the expected number of arguments does not match the given number
expected: at least 2
given: 1

```

The multiply procedure throws an error if the number of operands is less than 2, whereas the plus procedure works perfectly fine.

Suppose we want to scale a list of numbers by a factor of *< fact >*.

```
; This procedure scales all the elements in the input
; <items> by a factor of <fact>

(define (scale-list fact items)
  (if (null? items)
      '()
      (cons (* (car items) fact) (scale-list fact (cdr items)))))

; Alternatively one can abstract the idea of some procedure being applied
; to a sequence of inputs by using the following map procedure

(define (map procedure items)
  (if (null? items)
      '()
      (cons (procedure (car items)) (map procedure (cdr items)))))

(define scale-by (lambda (x) (lambda (y) (* x y))))

> (map (scale-by 5) (list 1 2 3 4))
(5 10 15 20)
> (scale-list 5 (list 1 2 3 4))
(5 10 15 20)
```

map is an important construct, not only because it captures a common pattern, but because it establishes a higher level of abstraction in dealing with lists.

Scheme standardly provides a `map` procedure that is more general than the one described here. This more general `map` takes

- ▶ a procedure of n arguments
- ▶ n lists each of size k

and applies the procedure to all the first elements of the lists, all the second elements of the lists, and so on, returning a list (of size k) of the results.

```
> (map + (list 1 2 3) (list 40 50 60) (list 700 800 900))
(741 852 963)
> (map (lambda (x y) (+ x (* 2 y)))
      (list 1 2 3)
      (list 4 5 6))
(9 12 15)
```

The above example makes use of the standard `map` provided by Scheme.

In `scale-list`, the recursive structure of the program draws attention to the element-by-element processing of the list.

Defining `scale-list` in terms of `map` suppresses that level of detail and emphasizes that scaling transforms a list of elements \rightarrow a list of results.

The difference between the two definitions is that we think about the process differently.

In `scale-list`, the recursive structure of the program draws attention to the element-by-element processing of the list.

Defining `scale-list` in terms of `map` suppresses that level of detail and emphasizes that scaling transforms a list of elements \rightarrow a list of results.

The difference between the two definitions is that we think about the process differently.

In effect, `map` helps establish an abstraction barrier that isolates

- ▶ the implementation of procedures that transform lists from
- ▶ the details of how the elements of the list are extracted and combined.

This allows us to focus solely on the conceptual framework of operations that transform sequences to sequences.

The procedure `for-each` is similar to `map`.

It takes as arguments a procedure and a list of elements.

However, rather than forming a list of the results, `for-each` just applies the procedure to each of the elements in turn, from left to right.

`map` returns a list, whereas `for-each` does not return anything. So `for-each` can be used with procedures that perform an action, such as printing.

```
(define (custom-print x)
  (newline)
  (display "Next square is ")
  (display x))
(for-each custom-print (map square (list 3 4 5 6)))
```

```
Next square is 9
Next square is 16
Next square is 25
Next square is 36
```

The representation of sequences in terms of lists generalizes naturally to represent sequences whose elements may themselves be sequences.

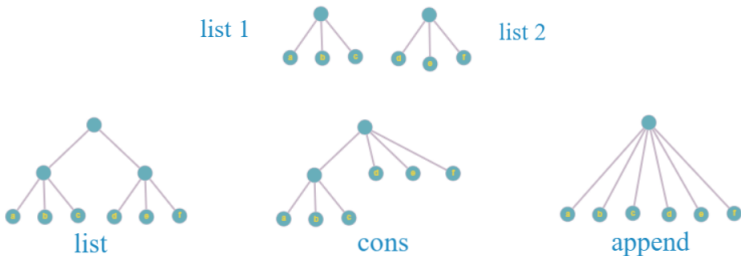
One way to think of sequences whose elements are sequences is as *trees*.

The elements of the sequence are the branches of the tree, and elements that are themselves sequences are subtrees.

For example $((1\ 2)\ 3\ 4)$ would represent a tree whose root node has three children. The first child is itself a tree with two nodes 1 and 2. The other two children of the root node are 3 and 4 in that order.

One can now think of the operations `list`, `cons` and `append` in terms of tree operations.

assume `list1` and `list2` are (a b c) and (d e f) respectively.
then `list`, `cons` and `append` for operands `list1` and `list2` (in that order) would produce the following trees.



The use of conventional interfaces is a powerful design principle for working with data structures. These conventional interfaces provide a means to communicate between procedures.

The key to organizing programs so as to more clearly reflect the signalflow structure is to concentrate on the “signals/intermediate results” that flow from one stage in the process to the next.

If we represent these intermediate-results as lists, then we can use list operations to implement the processing at each of the stages.

`map` is one of the higher order list functions.

Other frequently used higher order list functions also exist, such as `reduce`

The `reduce` function takes a binary procedure `<proc>` and applies it *right associatively* to a list `<input-list>` of an arbitrary number of elements.

```
; Higher order function reduce/accumulate
```

```
(define (reduce proc input-list base)
  (if (null? input-list)
      base
      (proc (car input-list) (reduce proc (cdr input-list) base))))
```

```
> (reduce + (list 5 6 7 8) 100)
126
> (reduce + (map square (list 5 6 7 8)) 100)
274
> (reduce / (list 20 50 100) 1)
40
> (/ 20 (/ 50 (/ 100 1)))
40
```


Filtering a sequence to select only those elements that satisfy a given predicate can be accomplished by [filter](#).

`<pred?>` is a boolean procedure that checks for a condition and if an element *a* of the `<input-list>` satisfies that condition then it is added to the output list.

```
; Higher order function filter
```

```
(define (filter pred? input-list)
  (cond ((null? input-list) '())
        ((pred? (car input-list))
         (cons (car input-list) (filter pred? (cdr input-list))))
        (else (filter pred? (cdr input-list))))
  )
```

```
> (filter number? (list #t 33/2" abc" 'e 4 'g 4.6))
(16 $\frac{1}{2}$  4 4.6)
```

Expressing programs as sequence operations helps us make program designs that are modular, i.e., designs that are constructed by combining *relatively independent pieces*.

We can encourage modular design by providing

- ▶ library of standard components and
- ▶ conventional interface for connecting the components in flexible ways.

Modular construction is a powerful strategy for controlling design complexity.

Sequences, implemented here as lists, serve as a conventional interface that permits us to combine processing modules.

Additionally, when we uniformly represent structures as sequences, we have localized the data-structure dependencies in our programs to a small number of sequence operations.

Sequences, implemented here as lists, serve as a conventional interface that permits us to combine processing modules.

Additionally, when we uniformly represent structures as sequences, we have localized the data-structure dependencies in our programs to a small number of sequence operations.

By changing these, we can experiment with alternative representations of sequences, while leaving the overall design of our programs intact.

So one could also think of designing another variant of scheme which has arrays/trees as a primitive data-structure and use it to represent sequences, without altering the higher order functions in the library.