14. Introduction to higher-order functions

Higher-order functions is another key area in the functional programming paradigm; Perhaps the most important at all. In this chapter we will explore this exiting area, and we will give a number of web-related examples.

14.1. Higher-order functions

Lecture 4 - slide 2

The idea of higher-order functions is of central importance for the functional programming paradigm. As we shall see on this and the following pages, this stems from the fact that higher-order functions can be further generalized by accepting functions as parameters. In addition, higher-order functions may act as function generators, because they allow functions to be returned as the result from other functions.

Let us first define the concepts of higher-order functions and higher-order languages.

A *higher-order function* accepts functions as arguments and is able to return a function as its result

A *higher-order language* supports higher-order functions and allows functions to be constituents of data structures

When some functions are 'higher-order' others are bound to be 'lower-order'. What, exactly, do we mean by the 'order of functions'. This is explained in below.

- The order of data
  - **Order 0**: Non function data
  - **Order 1**: Functions with domain and range of order 0
  - **Order 2**: Functions with domain and range of order 1
  - **Order *k***: Functions with domain and range of order *k-1*

Order 0 data have nothing to do with functions. Numbers, lists, and characters are example of such data.

Data of order 1 are functions which work on 'ordinary' order 0 data. Thus order 1 data are the functions we have been concerned with until now.

Data of order 2 - and higher - are example of the functions that have our interest in this lecture.

With this understanding, we can define higher-order functions more precisely.

Functions of order *i*, *i* >= 2, are called higher-order functions

14.2. Some simple and general higher-order functions

Lecture 4 - slide 3

It is time to look at some examples of higher-order functions. We start with a number of simple ones.

The `flip` function is given in two versions below. `flip` takes a function as input, which is returned with reversed parameters, cf. Program 14.1.

The first version of `flip` uses the shallow syntactic form, discussed in Section 8.12. The one in ??? uses the raw lambda expression, also at the outer level.

```
(define (flip f)
  (lambda (x y)
    (f y x)))
```

Program 14.1 *The function flip changes the order of it's parameters. The function takes a function of two parameters, and returns another function of two parameters. The only difference between the input and output function of flip is the ordering of their parameters.*

The read expression in Program 14.1 and ??? are the values returned from the function flip.

```
(define flip
  (lambda (f)
    (lambda (x y)
      (f y x))))
```

Program 14.2 *An alternative formulation of flip without use of the sugared define syntax.*

The function `negate`, as shown in Program 14.3, takes a predicate `p` as parameter. `negate` returns the negated predicate. Thus, if `(p x)` is true, then `((negate p) x)` is false.

```
(define (negate p)
  (lambda (x)
    (if (p x) #f #t)))
```

Program 14.3 *The function negate negates a predicate. Thus, negate takes a predicate function (boolean function) as parameter and returns the negated predicate. The resulting negated predicate returns true whenever the input predicate returns false, and vise versa.*

The function `compose` in Program 14.4 is the classical function composition operator, known by all high school students as 'f o g'

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

Program 14.4 *The function compose composes two functions which both are assumed to take a single argument. The resulting function composed of `f` and `g` returns `f(g(x))`, or in Lisp `(f (g x))`, given the input `x` . The `compose` function from the general LAML library accepts two or more parameters, and as such it is more general than the compose function shown here.*

Exercise 4.1. *Using `flip`, `negate`, and `compose`*

Define and play with the functions `flip`, `negate`, and `compose`, as they are defined on this page .

Define, for instance, a flipped `cons` function and a flipped minus function.

Define the function `odd?` in terms of `even?` and `negate`.

Finally, compose two HTML mirror functions, such as `b` and `em`, to a new function.

Be sure you understand your results.

14.3. Linear search in lists

Lecture 4 - slide 4

Let us program a simple, but useful higher-order function which searches a list by linear search. The function `find-in-list`, shown in Program 14.5 takes a predicate `pred` and a list `lst` as parameters. This predicate is applied on the elements in the list. The first element which satisfy the predicate is returned.

Search criterias can be passed as predicates to linear search functions

```
;; A simple linear list search function.
;; Return the first element which satisfies the predicate pred.
;; If no such element is found, return #f.
(define (find-in-list pred lst)
  (cond ((null? lst) #f)
        ((pred (car lst)) (car lst))
        (else (find-in-list pred (cdr lst)))))
```

Program 14.5 *A linear list search function. A predicate accepts as input an element in the list, and it returns either true (#t) or false (#f). If the predicate holds (if it returns true), we have found what we searched for. The predicate `pred` is passed as the first parameter to `find-in-list`. As it is emphasized in blue color, the predicate is applied on the elements of the list. The first successful application (an application with true result) terminates the search, and the element is returned. If the first case in the conditional succeeds (the brown condition) we have visited all elements in the list, and we conclude that the element looked for is not there. In that case we return false.*

The dialogue below shows examples of linear list search with `find-in-list`.

```
1> (define hair-colors (pair-up '(ib per ann) '("black" "green" "pink")))

2> hair-colors
((ib . "black") (per . "green") (ann . "pink"))

3> (find-in-list (lambda (ass) (eq? (car ass) 'per)) hair-colors)
(per . "green")

4> (find-in-list (lambda (ass) (equal? (cdr ass) "pink")) hair-colors)
(ann . "pink")

5> (find-in-list (lambda (ass) (equal? (cdr ass) "yellow")) hair-colors)
#f

6> (let ((pink-person
          (find-in-list
            (lambda (ass) (equal? (cdr ass) "pink")) hair-colors)))
    (if pink-person (car pink-person) #f))
ann
```

Program 14.6 *A sample interaction using* `find-in-list`*. We define a simple association list which relates persons (symbols) and hair colors (strings). The third interaction searches for per's entry in the list. The fourth interaction searches for a person with pink hair color. In the fifth interaction nothing is found, because no person has yellow hair color. In the sixth interaction we illustrate the convenience of boolean convention in Scheme: everything but #f counts as true. From a traditional typing point of view the* `let` *expression is problematic, because it can return either a person (a symbol) or a boolean value. Notice however, from a pragmatic point of view, how useful this is.*

Exercise 4.2. *Linear string search*

Lists in Scheme are linear linked structures, which makes it necessary to apply linear search techniques.

Strings are also linear structures, but based on arrays instead of lists. Thus, strings can be linearly searched, but it is also possible to access strings randomly, and more efficiently.

First, design a function which searches a string linearly, in the same way as `find-in-list`. Will you just replicate the parameters from `find-in-list`, or will you prefer something different?

Next program your function in Scheme.

Exercise 4.3. *Index in list*

It is sometimes useful to know *where in a list* a certain element occurs, if it occurs at all. Program the function `index-in-list-by-predicate` which searches for a given element. The comparion between the given element and the elements in the list is controlled by a comparison parameter to `index-in-list-by-predicate`. The function should return the list position of the match (first element is number 0), or #f if no match is found.

Some examples will help us understand the function:

```
(index-in-list-by-predicate '(a b c c b a) 'c eq?) => 2

(index-in-list-by-predicate '(a b c c b a) 'x eq?) => #f

(index-in-list-by-predicate '(two 2 "two") 2
  (lambda (x y) (and (number? x) (number? y) (= x y)))) => 1
```

Be aware if your function is tail recursive.

Exercise 4.4. *Binary search in sorted vectors*

Linear search, as illustrated by other exercises, is not efficient. It is often attractive to organize data in a sorted vector, and to do binary search in the vector.

This exercise is meant to illustrate a real-life higher-order function, generalized with several parameters that are functions themselves.

Program a function `binary-search-in-vector`, with the following signature:

```
(binary-search-in-vector v el sel el-eq? el-leq?)
```

`v` is the sorted vector. `el` is the element to search for. If `v-el` is an element in the vector, the actual comparison is done between `el` and `(sel v-el)`. Thus, the function `sel` is used as a selector on vector elements. Equality between elements is done by the `el-eq?` function. Thus, `(el-eq? (sel x) (sel y))` makes sense on elements x and y in the vector. The ordering of elements in the vector is defined by the `el-leq?` function. `(el-leq? (sel x) (sel y))` makes sense on elements x and y in the vector.

The call `(binary-search-in-vector v el sel el-eq? el-leq?)` searches the vector via binary search and it returns an element `el-v` from the vector which satisfies (el-eq? (sel el-v) el). If no such element can be located, it returns #f.

Here are some examples, with elements being cons pairs:

```
(binary-search-in-vector '#( (2 . x) (4 . y) (5 . z) (7 . i) (9 . c)
                             (11 . c)) 7 car = <=)    =>
(7 . i)

(binary-search-in-vector '#( (2 . x) (4 . y) (5 . z) (7 . i) (9 . c)
                             (11 . c)) 2 car = <=)    =>
(2 . x)
```

```
    (binary-search-in-vector '#( (2 . x) (4 . y) (5 . z) (7 . i) (9 . c)
                                 (11 . c)) 10 car = <=)    =>
    #f
```

Be sure to program a tail recursive solution.

14.4. Generation of list selectors

Lecture 4 - slide 5

The function `find-in-list` took a function as parameter. In this section we will give an example of a higher-order function which returns a function as result.

It is attractive to *generate* generalizations of the list selector functions `car`, `cadr`, etc

The function `make-selector-function` generates a list selector function which returns element number *n* from a list. It should be noticed that the first element in a list is counted as number one. This is contrary to the convention of the function `list-ref` and other similar Scheme function, which counts the first element in a list as number zero. This explains the `(- n 1)` expression in Program 14.7.

```
(define (make-selector-function n)
  (lambda (lst) (list-ref lst (- n 1))))
```

Program 14.7 *A simple version of the* `make-selector-function` *function.*

In the web version of the material (slide view or annotated slide view) you will find yet another version of the function `make-selector-function`, which provides for better error messages, in case element number *n* does not exist in the list. We have taken it out of this version because of its size and format.

The dialogue below shows examples of definitions and uses of list selector functions generated by `make-selector-function`.

```
1> (define first (make-selector-function 1 "first"))

2> (first '(a b c))
a

3> (first '())
The selector function first: The list () is is too short for selection.
It must have at least 1 elements.
>

4> (define (make-person-record firstname lastname department)
      (list 'person-record firstname lastname department))
```

```
5> (define person-record
      (make-person-record "Kurt" "Normark" "Computer Science"))

6> (define first-name-of (make-selector-function 2 "first-name-of"))

7> (define last-name-of (make-selector-function 3 "last-name-of"))

8> (last-name-of person-record)
"Normark"

9> (first-name-of person-record)
"Kurt"
```

Program 14.8 *Examples usages of the function make-selector-function. In interaction 1 through 3 we demonstrate generation and use of the first function. Next we outline how to define accessors of data structures, which are represented as lists. In reality, we are dealing with list-based record structures. In my every day programming, such list structures are quite common. It is therefore immensely important, to access data abstractly (via name accessors, instead of via the position in the list (car, cadr, etc). In this context, the make-selector-function comes in handy.*

14.5. References

[-] Foldoc: higher order function
http://wombat.doc.ic.ac.uk/foldoc/foldoc.cgi?query=higher+order+function

15. Mapping and filtering

In this chapter we will focus on higher-order functions that work on lists. It turns out that the appropriate combinations of these make it possible to solve a variety of different list processing problems.

15.1. Classical higher-order functions: Overview

Lecture 4 - slide 7

We start with an overview of the classical higher-order functions on lists, not just mapping and filtering, but also including reduction and zipping functions which we cover in subsequent sections.

There exists a few higher-order functions via which a wide variety of problems can be solved by simple combinations

- Overview:
    - **Mapping**: Application of a function on all elements in a list
    - **Filtering**: Collection of elements from a list which satisfy a particular condition
    - **Accumulation**: Pair wise combination of the elements of a list to a value of another type
    - **Zipping**: Combination of two lists to a single list

The functions mentioned above represent abstractions of *algorithmic patterns* in the functional paradigm

The idea of patterns has been boosted in the recent years, not least in the area of object-oriented programming. The classical higher-order list functions encode recursive patterns on the recursive data type list. As a contrast to many patterns in the object-oriented paradigm, the patterns encoded by `map`, `filter`, and others, can be programmed directly. Thus, the algorithmic patterns we study here are not design patterns. Rather, they are programming patterns for the practical functional programmer.

15.2. Mapping

Lecture 4 - slide 8

The idea of mapping is to apply a function on each element of a list, hereby collecting the list of the function applications

A mapping function applies a function on each element of a list and returns the list of these applications

The function `map` is an essential Scheme function

The idea of mapping is illustrated below.

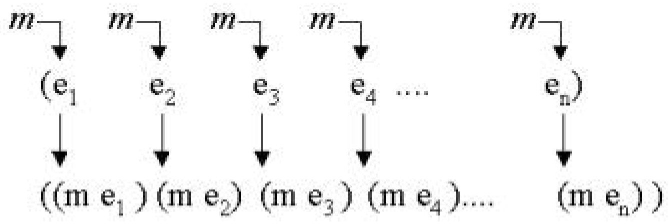


Figure 15.1 *Mapping a function m on a list. m is applied on every element, and the list of these applications is returned.*

15.3. The mapping function

Lecture 4 - slide 9

It is now time to study the implementation of the mapping function. We program a function called `mymap` in order not to redefine Scheme's own mapping function (a standard function in all Scheme implementations).

```
(define (mymap f lst)
  (if (null? lst)
      '()
       (cons (f (car lst))
             (mymap f (cdr lst)))))
```

Program 15.1 *An implementation of map. This is not a good implementation because the recursive call is not a tail call. We leave it as an exercise to make a memory efficient implementation with tail recursion - see the exercise below.*

Exercise 4.5. *Iterative mapping function*

In contrast to the function `mymap` on this page , write an iterative mapping function which is tail recursive.

Test your function against `mymap` on this page, and against the native `map` function of your Scheme system.

Exercise 4.6. *Table exercise: transposing, row elimination, and column elimination.*

In an earlier section we have shown the application of some very useful table manipulation functions. Now it is time to program these functions, and to study them in further details.

Program the functions `transpose`, `eliminate-row`, and `eliminate-column`, as they have been illustrated earlier. As one of the success criteria of this exercise, you should attempt to use higher-order functions as much and well as possible in your solutions.

Hint: Program a higher-order function, `(eliminate-element n)`. The function should return a function which eliminates element number n from a list.

15.4. Examples of mapping

Lecture 4 - slide 10

We will now study a number of examples.

Expresion	**Value**
`(map string? (list 1 'en "en" 2 'to "to"))`	`(#f #f #t #f #f #t)`
`(map (lambda (x) (* 2 x)) (list 10 20 30 40))`	`(20 40 60 80)`
`(ul (map (compose li b (lambda (x) (font-color red x))) (list "a" "b" "c") ) )`	(ul (map (compose li (compose b (lambda (x) (font-color red x)))) (list "a" "b" "c")))
Same as above	`(ul (map (compose li (compose b (lambda (x) (font-color red x)))) (list "a" "b" "c") ) )`

Table 15.1 *In the first row we map the* `string?` *predicate on a list of atoms (number, symbols, and strings). This reveals (in terms of boolean values) which of the elements that are strings. In the second row of the table, we map a 'multiply with 2' function on a list of numbers. The third row is more interesting. Here we map the composition of* `li` *,* `b` *, and red font coloring on the elements a, b, and c. When passed to the HTML mirror function* `ul` *, this makes an unordered list with red and bold items. Notice that the* `compose` *function used in the example is a higher-order function that can compose two or more functions. The function* `compose` *from* `lib/general.scm` *is such a function. Notice also that the HTML mirror function* `ul` *receives a list, not a string. The fifth and final row illustrates the raw HTML output, instead of the nicer rendering of the unordered list, which we used in the third row.*

15.5. Filtering

Lecture 4 - slide 11

As the name indicates, the `filter` function is good for examining elements of a list for a certain property. Only elements which possess the property are allowed through the filter.

> A filtering function applies a predicate (boolean function) on every element of a list. Only elements on which the predicate returns true are returned from the filtering function.
>
> The function `filter` is not an essential Scheme function - but is part of the LAML general library

The figure below illustrates the filtering idea.

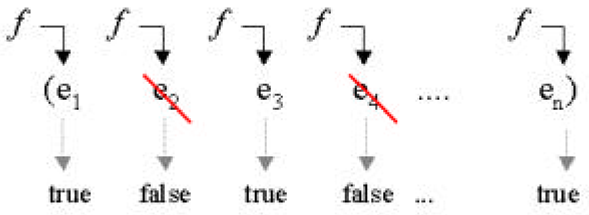


Figure 15.2 *Filtering a list with a predicate f. The resulting list is the subset of the elements which satisfy f (the elements on which f returns true).*

15.6. The filtering function

Lecture 4 - slide 12

The next item on the agenda is an implementation of `filter` .

> For practical purposes it is important to have a memory efficient `filter` function

As a consequence of the observation above, we now program a tail recursive version of `filter`. Notice that it is the function `filter-help`, which does the real filtering job.

```
(define (filter pred lst)
  (reverse (filter-help pred lst '())))

(define (filter-help pred lst res)
  (cond ((null? lst) res)
        ((pred (car lst))
           (filter-help pred (cdr lst)  (cons (car lst) res)))
        (else
           (filter-help pred (cdr lst)  res))))
```

Program 15.2 *An implementation of filter which is memory efficient. If the predicate holds on an element of the list (the red fragment) we include the element in the result (the brown fragment). If not (the green fragment), we drop the element from the result (the purple fragment).*

Exercise 4.7. *A straightforward filter function*

The `filter` function illustrated in the material is memory efficient, using tail recursion.

Take a moment here to implement the straightforward recursive filtering function, which isn't tail recursive.

15.7. Examples of filtering

Lecture 4 - slide 13

As we did for mapping, we will also here study a number of examples. As before, we arrange the examples in a table where the example expressions are shown to the left, and their values to the right.

Expression	**Value**

`(filter`
`  even?`
` '(1 2 3 4 5))`
	`(2 4)`

`(filter`
`  (negate even?)`
`  '(1 2 3 4 5))`
	`(1 3 5)`

`(ol`
` (map li`
`  (filter string?`
`   (list 1 'a "First" "Second" 3))))`
	1. First
	2. Second

Same as above	`<ol><li>First</li> <li>Second</li></ol>`

Table 15.2 *In the first row we filter the first five natural numbers with the* **`even?`** *predicate. In the second row, we filter the same list of numbers with the* **`odd?`** *predicate. Rather than using the name odd? we form it by calculating* **`(negate even?)`** *. We have seen the higher-order function* **`negate`** *earlier in this lecture. The third and final example illustrates the filtering of a list of atoms with the* **`string?`** *predicate. Only strings pass the filter, and the resulting list of strings is rendered in an ordered list by means of the mirror function of the ol HTML element.*

15.8. References

[-] Foldoc: filter
http://wombat.doc.ic.ac.uk/foldoc/foldoc.cgi?query=filter

[-] The LAML general library
http://www.cs.auc.dk/~normark/scheme/distribution/laml/lib/man/general.html

[-] Foldoc: map
http://wombat.doc.ic.ac.uk/foldoc/foldoc.cgi?query=map

16. Reduction and zipping

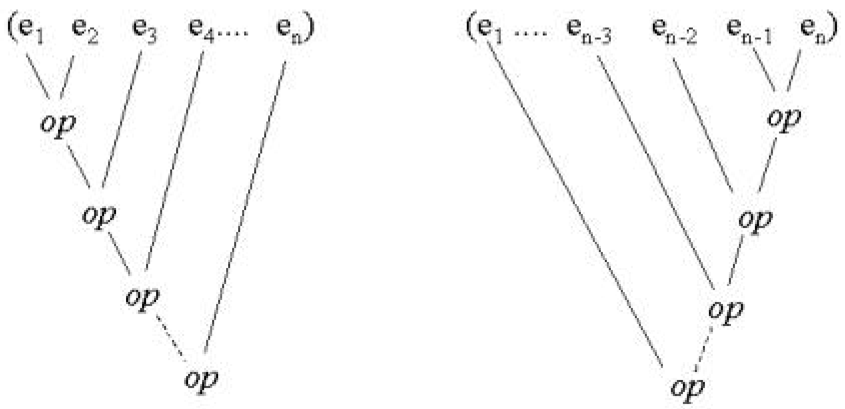
The reduction and zipping functions work on lists, like map and filter from Chapter 15.

16.1. Reduction

Lecture 4 - slide 15

List reduction is useful when we need somehow to 'boil down' a list to a 'single value'. The boiling is done with a binary function, as illustrated in Figure 16.1.

Reduction of a list by means of a binary operator transforms the list to a value in the range of the binary operator.



$(e_1\ e_2\ e_3\ e_4 \ldots\ e_n)$
op
op
op
op
$(e_1 \ldots\ e_{n-3}\ e_{n-2}\ e_{n-1}\ e_n)$
op
op
op
op

Figure 16.1 *Left and right reduction of a list. Left reduction is - quite naturally - shown to the left, and right reduction to the right.*

There is no natural value for reduction of the empty list. Therefore we assume as a precondition that the list is non-empty.

The intuitive idea of reduction will probably be more clear when we meet examples in Table 16.1 below.

Examples of left and right reduction are given in the table below. Be sure to understand the difference between left and right reduction, when the function, with which we reduce, is not commutative.

Expression	Value
`(reduce-left - '(1 2 3 4 5))`	`-13`
`(reduce-right - '(1 2 3 4 5))`	`3`
`(reduce-left string-append (list "The" " " "End"))`	`"The End"`
`(reduce-left append (list (list 1 2 3) (list 'a 'b 'c)))`	`(1 2 3 a b c)`

Table 16.1 *Examples of reductions. The - left reduction of the list corresponds to calculating the expression (- (- (- (- 1 2) 3) 4) 5). The - right reduction of the list corresponds to calculating the expression (- 1 (- 2 (- 3 (- 4 5)))).*

16.2. The reduction functions

Lecture 4 - slide 16

We will now implement the reduction functions introduced above in Section 16.1. Both right reduction and left reduction will be implemented, not least because they together illustrate a good point about iterative and tail recursive processing of lists. The explanations of this is found in the captions of Program 16.1 and Program 16.2.

The function `reduce-right` is a straightforward recursive function
The function `reduce-left` is a straightforward iterative function

```
(define (reduce-right f lst)
  (if (null? (cdr lst))
      (car lst)
      (f (car lst)
         (reduce-right f (cdr lst)))))
```

Program 16.1 *The function reduce-right. Notice the fit between the composition of the list and the recursive pattern of the right reduction.*

```
(define (reduce-left f lst)
  (reduce-help-left f (cdr lst) (car lst)))

(define (reduce-help-left f lst res)
  (if (null? lst)
      res
      (reduce-help-left f (cdr lst) (f res (car lst)))))
```

Program 16.2 *The function reduce-left. There is a misfit between left reduction and the recursive composition of the list with heads and tails. However, an iterative process where we successively combine e1 and e2 (giving r1), r1 and e3 etc., is straightforward. As we have seen several times, this can be done by a tail recursive function, here reduce-help-left.*

In summary, right reduction is easy to program with a recursive function. The reason is that we can reduce the problem to `(f (car lst) X)`, where `X` a right reduction of `(cdr lst)` with `f`. The right reduction of `(cdr lst)` is smaller problem than the original problem, and therefore we eventually meet the case where the list is trivial (in this case, a single element list).

The left reduction combines the elements one after the other, iteratively. First we calculate `(f (car el) (cadr el))`, provided that the list is of length 2 or longer. Let us call this value `Y`. Next `(f Y (caddr el))` is calculated, and so on in an iterative way. We could easily program this with a simple loop control structure, like a for loop.

16.3. Accumulation

Lecture 4 - slide 17

In this section we introduce a variation of reduction, which allows us also to reduce the empty list. We chose to use the word *accumulation* for this variant.

It is not satisfactory that we cannot reduce the empty list

We remedy the problem by passing an extra parameter to the reduction functions

We call this variant of the reduction functions for *accumulation*

It also turns out that the accumulation function is slightly more useful than `reduce-left` and `reduce-right` from Section 16.2. The reason is that we control the type of the parameter `init` to `accumulate-right` in Program 16.3. Because of that, the signature of the accumulate function becomes more versatile than the signatures of `reduce-left` and `reduce-right`. Honestly, this is not easy to spot in Scheme, whereas in languages like Haskell and ML, it would have been more obvious.

Below we show the function `accumulate-right`, which performs right accumulation. In contrast to `reduce-right` from Program 16.1`accumulate-right` also handles the extreme case of the empty list. If the list is empty, we use the explicitly passed `init` value as the result.

```
(define (accumulate-right f init lst)
  (if (null? lst)
      init
      (f (car lst) (accumulate-right f init (cdr lst)))))
```

Program 16.3 *The function accumulate-right. The recursive pattern is similar to the pattern of reduce-right.*

The table below shows a few examples of right accumulation, in the sense introduced above.

Expression	Value
`(accumulate-right - 0 '())`	`0`
`(accumulate-right - 0 '(1 2 3 4 5))`	`3`
`(accumulate-right append '() (list (list 1 2 3) (list 'a 'b 'c)))`	`(1 2 3 a b c)`

Table 16.2 *Examples of right accumulations. The first row illustrates that we can accumulate the empty list. The second and third rows are similar to the second and third rows in Table 15.1.*

In relation to web programming we most often append accumulate lists and strings

`accumulate-right` is part of the general LAML library

Due to their deficiencies, the reduction functions are not used in LAML

16.4. Zipping

Lecture 4 - slide 18

The zipping function is named after a zipper, as known from pants and shirts. The image below shows the intuition behind a list zipper.

Two equally long lists can be pair wise composed to a single list by means of *zipping* them

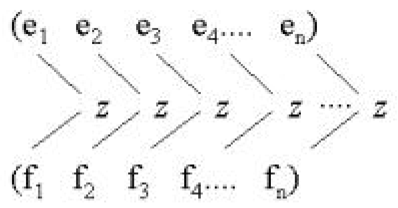


Figure 16.2 *Zipping two lists with the function* `z`*. The head of the resulting list is (*`z` `e`$_i$ `f`$_i$*), where the element* `e`$_i$ *comes from the first list, and* `f`$_i$ *comes from the other.*

We implement the zipping function in the following section.

16.5. The zipping function

Lecture 4 - slide 19

The `zip` function in Program 16.4 takes two lists, which are combined element for element. As a precondition, it is assumed that both input list have the same size.

```
(define (zip z lst1 lst2)
  (if (null? lst1)
      '()
      (cons
        (z (car lst1) (car lst2))
        (zip z (cdr lst1) (cdr lst2)))))
```

Program 16.4 *The function zip.*

Below we show examples of zipping with the `zip` function. For comparison, we also show an example that involves `string-merge`, which we discussed in Section 11.7.

Expression	**Value**
`(zip cons '(1 2 3) '(a b c))`	`((1 . a) (2 . b) (3 . c))`
`(apply string-append (zip string-append '("Rip" "Rap" "Rup") '(", " ", and " "")))`	`"Rip, Rap, and Rup"`
`(string-merge '("Rip" "Rap" "Rup") '(", " ", and "))`	`"Rip, Rap, and Rup"`

Table 16.3 *Examples of zipping.*

`Zip` is similar to the function `string-merge` from the LAML general library

However, `string-merge` handles lists of strings non-equal lengths, and it concatenates the zipped results

17. Currying

Currying is an idea, which is important in contemporary functional programming languages, such as Haskell. In Scheme, however, the idea is less attractive, due to the parenthesized notation of function calls.

Despite of this, we will discuss the idea of currying in Scheme via some higher-order functions like `curry` and `uncurry`. We will also study some ad hoc currying of Scheme functions, which has turned out to be useful for practical HTML authoring purposes, not least when we are dealing with tables.

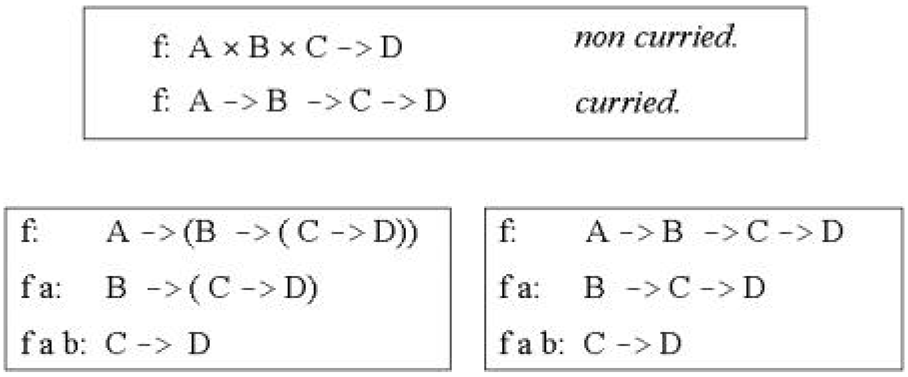
17.1. The idea of currying

Lecture 4 - slide 21

Currying is the idea of interpreting an arbitrary function to be of one parameter, which returns a possibly intermediate function, which can be used further on in a calculation.

Currying allows us to understand every function as taking at most one parameter. Currying can be seen as a way of generating intermediate functions which accept additional parameters to complete a calculation

The illustration below shows what happens to function signatures (parameter profiles) when we introduce currying.



f: A × B × C -> D   *non curried.*

f: A -> B -> C -> D   *curried.*

f: A -> (B -> (C -> D))
f a: B -> (C -> D)
f a b: C -> D

f: A -> B -> C -> D
f a: B -> C -> D
f a b: C -> D

Figure 17.1 *The signatures of curried functions. In the upper frame we show the signature of a function f, which takes three parameters. The frames below show the signature when f is curried. In the literature, the notation shown to the bottom right is most common. The frame to the left shows how to parse the notation (the symbol -> associates to the right).*

Currying and Scheme is not related to each other. Currying must be integrated at a more basic level to be elegant and useful

17.2. Currying in Scheme

Lecture 4 - slide 22

Despite the observations from above, we can explore and play with currying in Scheme. We will not, however, claim that it comes out as elegant as, for instance, in Haskell.

It is possible to generate curried functions in Scheme.

But the parenthesis notation of Lisp does not fit very well with the idea of currying

The function `curry2` generates a curried version of a function, which accepts two parameters. The curried version takes one parameter at a time. Similarly, `curry3` generates a curried version of a function that takes three parameters.

The functions `uncurry2` and `uncurry3` are the inverse functions.

It is worth a consideration if we can generalize `curry2` and `curry3` to a generation of `curry`*n* via a higher-order function `curry`, which takes *n* as parameter. We will leave that as an open question.

```
(define (curry2 f)
  (lambda(x)
    (lambda(y)
      (f x y))))

(define (curry3 f)
  (lambda(x)
    (lambda(y)
      (lambda(z)
       (f x y z)))))

(define (uncurry2 f)
  (lambda (x y)
    ((f x) y)))

(define (uncurry3 f)
  (lambda (x y z)
    (((f x) y) z)))
```

Program 17.1 *Generation of curried and uncurried functions in Scheme.*

Exercise 4.8. *Playing with curried functions in Scheme*

Try out the functions `curry2` and `curry3` on a number of different functions.

You can, for instance, use then curry functions on plus (+) and map.

Demonstrate, by a practical example, that the uncurry functions and the curry functions are inverse to each other.

17.3. Examples of currying

Lecture 4 - slide 23

Let us here show a couple of examples of the curry functions from Section 17.2.

Curried functions are very useful building blocks in the functional paradigm In particular, curried functions are adequate for mapping and filtering purposes

The function font-1 is assumed to take three parameters. The font size (an integer), a color (in some particular representation that we do not care about here) and a text string on which to apply the font information. We show a possible implementation of font-1 in terms of the font mirror function in Program 17.2.

Expression	**Value**
`(font-1 4 red "Large red text")`	Large red text

`(define curried-font-1 (curry3 font-1))`
`(define large-font (curried-font-1 5))`
`((large-font blue) "Very large blue text")`
	Very large blue text

`(define small-brown-font ((curried-font-1 2) brown))`
`(small-brown-font "Small brown text")`
	Small brown text

`(define large-green-font ((curried-font-1 5) green))`
`(list-to-string (map large-green-font (list "The" "End")) " ")`
	The End

Table 17.1 *Examples of currying in Scheme.*

```
(define (font-1 size color txt)
  (font 'size (as-string size)
        'color (rgb-color-encoding color)
        txt))
```

Program 17.2 *A possible implementation of font-1 in terms of the font HTML mirror function.*

17.4. Ad hoc currying in Scheme (1)

Lecture 4 - slide 24

In some situations we would wish that the `map` function, and similar functions, were curried in Scheme. But we cannot generate an *f*- mapper by evaluating the expression `(map f)`. We get an error message which tells us that `map` requires at least two parameters.

In this section we will remedy this problem by a pragmatic, ad hoc currying made via use of a simple higher-order function we call `curry-generalized`.

It is possible to achieve 'the currying effect' by generalizing functions, which requires two or more parameters, to only require a single parameter

In order to motivate ourselves, we will study a couple of attempts to apply a curried mapping function.

Expression	**Value**
`(map li (list "one" "two" "three"))`	`("<li>one</li>" "<li>two</li>" "<li>three</li>")`
`(define li-mapper (map li))`	*map: expects at least 2 arguments, given 1*
`(define li-mapper ((curry2 map) li))` `(li-mapper (list "one" "two" "three"))`	`("<li>one</li>" "<li>two</li>" "<li>three</li>")`

Table 17.2 *A legal mapping and an impossible attempt to curry the mapping function. The last example shows an application of curry2 to achieve the wanted effect, but as it appears, the solution is not very elegant.*

In Program 17.3 we program the function `curry-generalized`. It returns a function that generalizes the parameter `f`. If we pass a single parameter to the resulting function, the value of the red lambda expression is returned. If we pass more than one parameter to the resulting function, `f` is just applied in the normal way.

```
(define (curry-generalized f)
  (lambda rest
    (cond ((= (length rest) 1)
             (lambda lst (apply f (cons (car rest) lst))))
          ((>= (length rest) 2)
             (apply f (cons (car rest) (cdr rest)))))))
```

Program 17.3 *The function curry-generalized. This is a higher-order function which generalizes the function passed as parameter to* `curry-generalized`. *The generalization provides for just passing a single parameter to* `f`*, in the vein of currying.*

The blue expression aggregates the parameters - done in this way to be compatible with the inner parts of the red expression. In a simpler version `(cons (car rest) (cdr rest))` would be replace by `rest`.

In the next section we see an example of curry generalizing the `map` function.

17.5. Ad hoc currying in Scheme (2)

Lecture 4 - slide 25

We may now redefine `map` to `(curry-generalized map)`. However, we usually bind the curry generalized mapping function to another name, such as `gmap` (for generalized `map`).

This section shows an example, where we generate a `li` mapper, by `(gmap li)`.

Expression	**Value**

`(define gmap (curry-generalized map))`
`(define li-mapper (gmap li))`
`(li-mapper (list "one" "two" "three"))`
	`("<li>one</li>"`
	` "<li>two</li>"`
	` "<li>three</li>")`

`(gmap li (list "one" "two" "three"))`	`("<li>one</li>"`
	` "<li>two</li>"`
	` "<li>three</li>")`

Table 17.3 *Examples of curry generalization of map. Using* ***`curry-generalized`*** *it is possible to make a* ***`li`****-mapper in an elegant and satisfactory way. The last row in the table shows that* ***`gmap`*** *can be used instead of* ***`map`****. Thus,* ***`gmap`*** *can in all respect be a substitution for* ***`map`****, and we may chose to redefine the name* ***`map`*** *to the value of* ***`(curry-generalized map)`****.*

If we redefine map to `(curry-generalized map)`, the new mapping function can be used instead of the old one in all respects. In addition, `(map f)` now makes sense; `(map f)` returns a function, namely an *f mapper*. Thus `((map li) "one" "two" "three")` does also make sense, and it gives the result shown in one of value cells to the right of Table 17.3.

17.6. References

[-] Foldoc: curried function
http://wombat.doc.ic.ac.uk/foldoc/foldoc.cgi?query=curried+function

18. Web related higher-order functions

We finish our coverage of higher-order functions with a number of examples from the web domain.

18.1. HTML mirror generation

Lecture 4 - slide 27

In this section we will, in a principled way, show how to generate simple HTML mirror functions in Scheme. Please notice that the HTML mirror functions in LAML are more sophisticated and elaborate than the ones discussed here.

There are three different cases to consider: double tag elements, single tag elements, and tags that can be both single and double.

A well-known tag, that can be both single and double is the p tag.

The higher-order functions `generate-double-tag-function` and `generate-single-tag-function` are the top level functions. They rely on a couple of other functions, which we program in Program 18.2 - Program 18.4.

```
(define (generate-double-tag-function tag-name)
  (lambda (contents . attributes)
    (double-tag tag-name contents attributes)))

(define (generate-single-tag-function tag-name)
  (lambda attributes
    (single-tag tag-name attributes)))
```

Program 18.1 *The two higher-order functions for the HTML mirror generation. This version corresponds to the an earlier version of LAML's HTML mirror.*

```
(define (single-tag name attributes)
 (start-tag name attributes))

(define (double-tag name contents attributes)
 (string-append (start-tag name attributes)
                (as-string contents)
                (end-tag name)))
```

Program 18.2 *Functions that generate single and double tags.*

The functions `start-tag` and `end-tag` are used in Program 18.2 and implemented in Program 18.3.

```
(define (start-tag kind attributes)
  (if (null? attributes)
      (string-append "<" kind ">")
      (let ((html-attributes (linearize-attributes attributes)))
         (string-append "<" kind " " html-attributes " >"))))

(define (end-tag kind)
  (string-append "</" kind ">"))
```

Program 18.3 *Functions that generate individual single and double tags.*

The missing aspect at this point is the attribute handling stuff. It is made in Program 18.4.

```
(define (linearize-attributes attr-list)
  (string-append
    (linearize-attributes-1
      (reverse attr-list) "" (length attr-list))))

(define (linearize-attributes-1 attr-list res-string lgt)
  (cond ((null? attr-list) res-string)
        ((>= lgt 2)
          (linearize-attributes-1
           (cddr attr-list)
           (string-append
            (linearize-attribute-pair
             (car attr-list) (cadr attr-list)) " " res-string)
           (- lgt 2)))
        ((< lgt 2)
          (error "The attribute list must have even length"))))

(define (linearize-attribute-pair val attr)
  (string-append (as-string attr)
                  " = " (string-it (as-string val))))
```

Program 18.4 *Functions for attribute linearization. The parameter attr-list is a property list.*

Recall that property lists, as passed to the function `linearize-attributes` in Program 18.4 have been discussed in Section 6.6.

There are several things to notice relative to LAML. First, the HTML mirror in LAML does not generate strings, but an internal representation akin to abstract syntax trees.

Second, the string concatenation done in Program 18.1 through Program 18.4, where a lot of small strings are aggregated, generates a lot of 'garbage strings'. The way this is handled (by the `render` functions in LAML) is more efficient, because we write string parts directly into a stream (or into a large, pre-allocated string).

You will find more details about LAML in Chapter 25 and subsequent chapters.

18.2. HTML mirror usage examples

Lecture 4 - slide 28

Let us now use the HTML mirror generation functions, which we prepared via `generate-double-tag-function` and `generate-single-tag-function` in Section 18.1.

The example assumes loading of `laml.scm` and the function `map-concat`, which concatenates the result of a map application. The real mirrors use implicit (string) concatenation

As noticed above, there some differences between the real LAML mirror functions and the ones programmed in Section 18.1. The functions from above require string appending of such constituents as the three `tr` element instances in the table; This is inconvenient. Also, the mirror functions from above require that each double element gets exactly one content string followed by a number of attributes. The real LAML mirror functions accept pieces of contents and attributes in arbitrary order (thus, in some sense generalizing the XML conventions where the attributes come before the contents inside the start tag). Finally, there is no kind of contents nor attribute validation in the mirror functions from above. The LAML mirror functions validate both the contents and the attributes relative to the XML Document Type Definition (DTD).

Expression	**Value**

`(let* ((functions`
`         (map generate-double-tag-`
`function`
`              (list "table" "td"`
`"tr")))`
`       (table (car functions))`
`       (td (cadr functions))`
`       (tr (caddr functions)))`
` (table`
`  (string-append`
`   (tr`
`     (map-concat td (list "c1" "c2"`
`"c3"))`
`     'bgcolor "#ff0000")`
`   (tr`
`     (map-concat td (list "c4" "c5"`
`"c6")))`
`   (tr`
`     (map-concat td (list "c7" "c8"`
`"c9"))))`
`  'border 3))`
	`<table border="3">`
	`   <tr bgcolor="#ff0000">`
	`     <td> c1 </td>`
	`     <td> c2 </td>`
	`     <td> c3 </td>`
	`   </tr>`
	`   <tr>`
	`     <td> c4 </td>`
	`     <td> c5 </td>`
	`     <td> c6 </td>`
	`   </tr>`
	`   <tr>`
	`     <td> c7 </td>`
	`     <td> c8 </td>`
	`     <td> c9 </td>`
	`   </tr>`
	`</table>`

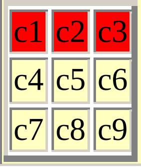
Same as above	 c1 c2 c3 c4 c5 c6 c7 c8 c9

Table 18.1 *An example usage of the simple HTML mirror which we programmed on the previous page. The bottom example shows, as in earlier similar tables, the HTML rendering of the constructed table. The* `map-concat` *function used in the example is defined in the general LAML library as* `(define (map-concat f lst) (apply string-append (map f lst)))`*. In order to actually evaluate the expression you should load* `laml.scm` *of the LAML distribution first.*

To show the differences between the simple mirror from Section 18.1 and the real mirror we will show the same example using the XHTML mirror functions in Section 18.3.

18.3. Making tables with the real mirror

Lecture 4 - slide 29

The real mirror provide for more elegance than the simple mirror illustrated above

Here we will use the XHTML1.0 transitional mirror

In the example below there is no need to string append the tr forms, and there is no need to use a special string appending mapping function, like `map-concat` from Table 18.1. Attributes can appear before, within, or after the textual content. This makes the HTML mirror expression simpler and less clumsy. The rendering result is, however, the same.

Expression	**Rendered value**

`(table`
`  'border 3`
`   (tr`
`     (map td (list "c1" "c2" "c3"))`
`     'bgcolor "#ff0000")`
`   (tr`
`     (map td (list "c4" "c5" "c6")))`
`   (tr`
`     (map td (list "c7" "c8" "c9")))`
`)`
	`<table border = "3">`
	`  <tr bgcolor = "#ff0000">`
	`   <td>c1</td>`
	`   <td>c2</td>`
	`   <td>c3</td>`
	`  </tr>`
	`  <tr>`
	`   <td>c4</td>`
	`   <td>c5</td>`
	`   <td>c6</td>`
	`  </tr>`
	`  <tr>`
	`    <td>c7</td>`
	`    <td>c8</td>`
	`    <td>c9</td>`
	`  </tr>`
	`</table>`

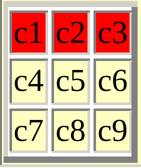
Same as above	 c1 c2 c3 c4 c5 c6 c7 c8 c9

Table 18.2 *A XHTML mirror expression with a table corresponding to the table shown on the previous page and the corresponding HTML fragment. Notice the absence of string concatenation. Also notice that the border attribute is given before the first tr element. The border attribute could as well appear after the tr elements, or in between them.*

You might think, that the example above also could be HTML4.01. But, not quite, in fact. In HTML4.01 there need to be a `tbody` (table body) form in between the `tr` instances and the `table` instance. Without this extract level, the table expression will not be valid. Try it yourself! It is easy.

[How, you may ask. In Emacs do `M-x set-interactive-laml-mirror-library` and enter `html-4.01`. Then do `M-x run-laml-interactively`. Copy the table expression from above, and try it out. You can shift to XHTML1.0 by `M-x set-interactive-laml-mirror-library` and asking for `xhtml-1.0-transitional`, for instance. Then redo `M-x run-laml-interactively`. Be sure to use `xml-render` on the result of `(table ...)` to make a textual rendering.]

18.4. Tables with higher-order functions

Lecture 4 - slide 30

In the context of higher-order functions there are even better ways to deal with tables than the one shown in Table 18.2 from Section 18.3.

The table expression in the last line in Table 18.3 shows how.

Instead of explicit composition of `td` and `tr` elements we can use a mapping to apply `tr` to rows and `td` to elements

Expression	Value

`(define rows`
`  '(("This" "is" "first" "row")`
`    ("This" "is" "second" "row")`
`    ("This" "is" "third" "row")`
`    ("This" "is" "fourth" "row"))`
`)`
 	
`(table 'border 5`
`    (gmap (compose tr (gmap td))`
`rows))`
	This | is | first | row
	This | is | second | row
	This | is | third | row
	This | is | fourth | row

Table 18.3 *In the table expression we map - at the outer level - a composition of `tr` and a `td`-mapper. The `td`-mapper is made by `(gmap td)`.*

Recall that we already have discussed the ad hoc currying, which is involved in `gmap`, cf. the discussion in Section 17.4.

The last example illustrates that `(gmap td)` is a useful building block, which can be composed with other functions.

The last example depends on the fact that the HTML mirror functions accept lists of elements and attributes.

You should consult Chapter 26 to learn about the exact parameter passing rules of the HTML mirror functions in LAML.

18.5. HTML element modifications

Lecture 4 - slide 31

It is often useful in some context to bind an attribute of a HTML mirror function (or a number of attributes) to some fixed value(s). This can be done by the higher-order function `modify-element`, which we discuss below.

The idea behind the function `modify-element` is to perform an a priori binding of some attributes and some of the contents of a mirror function.

The function `modify-element` is simple. First notice that it accepts a function, namely the `element` parameter. It also returns a function; In effect, it returns `element` with `attributes-and-contents` appended to the parameters of the modified element. As another possibility, we could have prepended it.

```
(define (modify-element element . attributes-and-contents)
  (lambda parameters
   (apply element
    (append parameters attributes-and-contents))))
```

Program 18.5 *The function modify-element.*

In the table below we illustrate three examples where `td`, `ol`, and `ul` are modified with a priori bindings of selected attributes.

Expression	**Value**

`(define td1`
` (modify-element td`
`  'bgcolor (rgb-color-list red)`
`  'width 50))`
 	
`(table 'border 5`
`  (map (compose tr (gmap td1)) rows))`
	This | is | first | row
	This | is | second | row
	This | is | third | row
	This | is | fourth | row

`(define ol1`
`  (modify-element ol 'type "A"))`
 	
`(ol1`
` (map`
`   (compose li as-string)`
`   (number-interval 1 10)))`
	A. 1
	B. 2
	C. 3
	D. 4
	E. 5
	F. 6
	G. 7
	H. 8
	I. 9
	J. 10

`(define ul1`
`  (modify-element ul 'type "square"))`
 	
`(ul1`
` (map`
`  (compose li as-string)`
`  (number-interval 1 10)))`
	▪ 1
	▪ 2
	▪ 3
	▪ 4
	▪ 5
	▪ 6
	▪ 7
	▪ 8
	▪ 9
	▪ 10

Table 18.4 *Examples of element modification using the function* ***`modify-element`***.

LAML supports two related, but more advanced functions called `xml-in-laml-parametrization` and `xml-in-laml-abstraction`. The first of these is intended to transform an 'old style function' to a function with XML-in-LAML parameter conventions, as explained in Chapter 26. The second function is useful to generate functions with XML-in-LAML parameter conventions in general.

18.6. The function simple-html-table

Lecture 4 - slide 32

We will now show show an implementation of the function `simple-html-table`

> In an earlier exercise - 'restructuring of lists' - we have used the function `simple-html-table`
>
> We will now show how it can be implemented

```
(define simple-html-table
 (lambda (column-widht list-of-rows)
  (let ((gmap (curry-generalized map))
        (td-width
          (modify-element td 'width
                          (as-string column-widht))))
    (table
      'border 1
      (tbody
       (gmap (compose tr (gmap td-width)) list-of-rows))))))
```

Program 18.6 *The function simple-html-table. Locally we bind* ***`gmap`*** *to the curry generalized map function. We also create a specialized version of* ***`td`****, which includes a* ***`width`*** *attribute the value of which is passed as parameter to* ***`simple-html-table`****. In the body of the* `let` *construct we create the table in the same way as we have seen earlier in this lecture.*

18.7. The XHTML mirror in LAML

Lecture 4 - slide 33

In order to illustrate the data, on which the HTML mirrors in LAML rely, the web edition of the material includes a huge table with the content model and attribute details of each of the 77 XHTML1.0 strict elements.

> LAML supports an exact mirror of the 77 XHTML1.0 strict elements as well as the other XHTML variants
>
> The LAML HTML mirror libraries are based on a parsed representation of the HTML DTD (Document Type Definition). The table below is automatically generated from the same data structure.

The table is too large to be included in the paper version of the material. Please take a look in the corresponding part of the web material to consult the table.

18.8. Generation of a leq predicate from enumeration

Lecture 4 - slide 34

As the last example related to higher-order functions we show the function `generate-leq`, see Program 18.7.

The idea is to generate a boolean 'less than or equal' (leq) function based on an explicit enumeration order, which is given as input to the function `generate-leq`. A number of technicalities are involved. You should read the details in Program 18.7 to grasp these details.

In some contexts we wish to specify a number of clauses in an arbitrary order For presentational clarity, we often want to ensure that the clauses are presented in a particular order Here we want to generate a leq predicate from an enumeration of the desired order

```
;; Generate a less than or equal predicate from the
;; enumeration-order. If p is the generated predicate,
;; (p x y) is true if and only if (selector x) comes before
;; (or at the same position) as (selector y) in the
;; enumeration-order. Thus, (selector x) is assumed to give a
;; value in enumeration-order. Comparison with elements in the
;; enumeration-list is done with eq?
(define (generate-leq enumeration-order selector)
  (lambda (x y)
     ; x and y supposed to be elements in enumeration order
     (let ((x-index (list-index (selector x) enumeration-order))
           (y-index (list-index (selector y) enumeration-order)))
       (<= x-index y-index))))

; A helping function of generate-leq.
; Return the position of e in lst. First is 1
; compare with eq?
; if e is not member of lst return (+ 1 (length lst))
(define (list-index e lst)
 (cond ((null? lst) 1)
       ((eq? (car lst) e) 1)
       (else (+ 1 (list-index e (cdr lst))))))
```

Program 18.7 *The functions `generate-leq` and the helping function `list-index`.*

The table below shows a very simple example, in which we use `simple-leq?`, which is generated by the higher-order function `generate-leq` from Program 18.7.

Expression	Value
`(define simple-leq? (generate-leq '(z a c b y x) id-1))` `(sort-list '(a x y z c c b a) simple-leq?)`	`(z a a c c b y x)`

Table 18.5 *A simple example of an application of generate-leq.*

The fragment in Program 18.8 gives a more realistic example of the use of generated 'less than or equal' functions. In Program 18.9 we show how the desired sorting of `manual-page` subelements is achieved.

```
(manual-page
 (form '(show-table rows))
 (title "show-table")
 (description "Presents the table, in terms of rows")
 (parameter "row" "a list of elements")
 (pre-condition "Must be placed before the begin-notes clause")
 (misc "Internally, sets the variable lecture-id")
 (result "returns an HTML string")
)
```

Program 18.8 *A hypothetical manual page clause. Before we present the clauses of the manual page we want to ensure, that they appear in a particular order, say title, form, description, pre-condition, result, and misc. In this example we will illustrate how to obtain such an ordering in an elegant manner.*

```
(define (syntactic-form name)
  (lambda subclauses (cons name subclauses)))

(define form (syntactic-form 'form))
(define title (syntactic-form 'title))
(define description (syntactic-form 'description))
(define parameter (syntactic-form 'parameter))
(define pre-condition (syntactic-form 'pre-condition))
(define misc (syntactic-form 'misc))
(define result (syntactic-form 'result))

(define (manual-page . clauses)
 (let ((clause-leq?
         (generate-leq
           '(title form description
             pre-condition result misc)
           first))
      )
  (let ((sorted-clauses (sort-list clauses clause-leq?)))
    (present-clauses sorted-clauses))))
```

Program 18.9 *An application of* `generate-leq` *which sorts the manual clauses.*

18.9. References

[-] The XHTML1.0 frameset validating mirror
http://www.cs.auc.dk/~normark/scheme/distribution/laml/lib/xml-in-laml/mirrors/man/xhtml10-frameset-mirror.html

[-] The XHTML1.0 transitional validating mirror
http://www.cs.auc.dk/~normark/scheme/distribution/laml/lib/xml-in-laml/mirrors/man/xhtml10-transitional-mirror.html

[-] The XHTML1.0 strict validating mirror
http://www.cs.auc.dk/~normark/scheme/distribution/laml/lib/xml-in-laml/mirrors/man/xhtml10-strict-mirror.html

[-] The HTML4.01 transitional validating mirror
http://www.cs.auc.dk/~normark/scheme/distribution/laml/lib/html4.01-transitional-validating/man/surface.html