9. Name binding constructs

In Section 8.1 we saw how to bind names globally, at top level, by use of `define`. In this chapter we will study local name binding constructs - `let` constructs - and we will se how they are made by means of lambda expressions.

9.1. The `let` name binding expression

Lecture 3 - slide 2

Let us first define what we mean by name binding constructs.

A *name binding expression* establishes a number of *local name bindings* in order to ease the evaluation of a body expression

In a name binding construct a number of names are bound to values. The name bindings can be used in the body, which must be an expression when we are working in the functional paradigm. There are a number of variations in the way the names can refer to each other mutually. We will meet some of them on the following pages.

The syntax a let form follows.

```
(let ((n1 e1)
      ...
      (nk ek))
  body-expr)
```

Syntax 9.1 *The names n_1 ... n_k are bound to the respective values e_1 ... e_k, and the body expression is evaluated relative to these name bindings. Free names in the body expression are bound to names defined in the surround of the let construct.*

- Characteristics of a `let` construct:
  - In `body-expr` n_1 refers to the value of e_1, ..., and n_k refers to the value of e_k
  - Syntactic sugar for an immediate call of a lambda expression
    - To be illustrated on the next page
  - As a consequence, the names are bound *simultaneously* relative to the name bindings in effect in the context of the `let` construct.

The idea of *simultaneous name binding* is especially important to understand. Take a close look at the second example of Table 9.1 If you understand the result of the let expression in this example, you probably understand simultaneous name binding.

9.2. The equivalent meaning of `let`

Lecture 3 - slide 3

A `let` construct can be defined by use of the name binding features of a lambda expression. In the rest of this section, we will see how it is done.

We will here understand the underlying, equivalent form of a `let` name binding construct

Below we show a syntactic equivalence. The let form in Syntax 9.2 is fully equivalent with the lambda expression in Syntax 9.3

Whenever a form like Syntax 9.2 is encountered it is transformed to the equivalent, but more basic form of Syntax 9.3. The syntactic transformation is done by a Scheme macro.

```
(let ((n1 e1)
      ...
      (nk ek))
  body-expr)
```

Syntax 9.2

```
((lambda (n1 ... nk) body-expr)
   e1 ... ek)
```

Syntax 9.3

9.3. Examples with `let` name binding

Lecture 3 - slide 4

We provide a couple of examples of name binding with let. The examples are drawn from the web domain.

Expression	**Value after rendering**

`(let ((anchor "An anchor text")`
`      (url "http://www.cs.auc.dk")`
`      (tag a)`
`     )`
`  (tag 'href url anchor))`
	An anchor text

`(let ((f b))`
`  (let ((f em)`
`        (g f))`
`    (p (f "Text 1") (g "Text 2"))))`
	Text 1 **Text 2**

`(let ((phrase-elements`	• *foo*

`        (list em strong dfn code samp`
`              kbd var cite abbr acronym))`
`     )`
`  (ul`
`   (map`
`    (lambda (f) (li (f "foo")))`
`    phrase-elements)))`
	• **foo**
	• *foo*
	• `foo`
	• `foo`
	• `foo`
	• *foo*
	• *foo*
	• foo
	• foo

Table 9.1 *Examples of namebindings with* `let`*. The first example shows that all constituents of a function call can be bound to local names - in the example both the function object referred to by a, and two string parameters. The second example illustrates that alternative names, aliases, can be defined for a couple of functions. Notice in particular that g is bound to b (the bold face function), not em (the emphasis function). This can also be seen in the second column. The third example is a little more advanced, and it can first be understood fully on the ground of the material in the lecture about higher-order functions. We bind the name* `phrase-elements` *to a list of ten functions. Via mapping, we apply each function to* `foo`*, and we present the results in an ul list.*

9.4. The `let*` name binding construct

Lecture 3 - slide 5

It is often useful to use a sequential alternative to simultaneous name binding, ala let. In this section we will study let*, which provides for sequential name binding.

It is often useful to be able to use previous name bindings in a let construct, which binds several names

The syntax of `let*`, as shown in Syntax 9.4 is very close to the syntax of `let`, which we saw in Syntax 9.1.

```
(let* ((n_1 e_1)
       ...
       (n_i-1 e_i-1)
       (n_i e_i)
       ...
       (n_k e_k))
  body-expr)
```

Syntax 9.4

- Characteristics of `let*`:
    - It is possible to refer to n_1 through n_{i-1} from the expression e_i
    - Syntactic sugar for k nested `let` name bindings

Take a moment to understand the last item above. Thus, try to understand that it is possible to obtain the effect of sequential name bindings by nesting a number of ordinary `let` constructs. In that way, we can devise a rewriting of a `let*` construct to a construct with nested lambda expressions.

9.5. An example with `let*`

Lecture 3 - slide 6

In the example on this page we show a function from the LAML time library. There is access to this library from the web material, cf. [timelib].

```scheme
(define (how-many-days-hours-minutes-seconds n)
  (let* ((days     (quotient n seconds-in-a-day))
         (n-rest-1 (modulo n seconds-in-a-day))
         (hours    (quotient n-rest-1 seconds-in-an-hour))
         (n-rest-2 (modulo n-rest-1 seconds-in-an-hour))
         (minutes  (quotient n-rest-2 60))
         (seconds  (modulo n-rest-2 60))
        )
    (list days hours minutes seconds)))
```

Program 9.1 *A typical example using sequential name binding. The task is to calculate the number of days, hours, minutes, and seconds given a number of seconds. We subsequently calculate a number of quotients and rest. While doing so we find the desired results. In this example we would not be able to use let; let* is essential because a given calculation depends on previous name bindings. The full example, including the definition of the constants, can be found in the accompanying elucidative program. The function is part of the LAML time library in lib/time.scm of the LAML distribution. The time library is used whenever we need to display time information, such as 'the time of generation' of some HTML files.*

In the web version of the material we provide a link to an elucidator which explains the basic time calculations in LAML. Please refer to the web version to get access to this resource.

Examples that illustrate uses of the LAML time functions are given later in the material, in Section 9.7.

9.6. The `letrec` namebinding construct

Lecture 3 - slide 7

There exists a third local name binding form, called letrec. It is used for local definition of mutually recursive functions, as sketched in Program 9.2.

The `letrec` name binding construct allows for definition of mutually recursive functions

```
(letrec ((n₁ e₁)
         ...
         (nₖ eₖ))
  body-expr)
```

Syntax 9.5

```
(letrec ((f1 (lambda (...) ... (f2 ...)))
         (f2 (lambda (...) ... (f1 ...)))
        )
  body-expr)
```

Program 9.2 *An schematic example of a typical application of letrec for local definition of two mutually recursive functions.*

- Characteristics of `letrec`
  - Each of the name bindings have effect in the entire `letrec` construct, including e_1 to e_k

9.7. LAML time functions

Lecture 3 - slide 8

In section Section 9.5 we discussed the function `how-many-days-hours-minutes-second`. We will now illustrate some other useful LAML time functions.

Expression	Value
`(current-time)`	`999789132`
`(time-decode 1000000000)`	`(2001 9 9 3 46 40)`
`(time-decode 0)`	`(1970 1 1 2 0 0)`
`(time-interval 1000000000)`	`(31 8 2 5 1 46 40)`
`(weekday (current-time))`	`Thursday`
`(danish-week-number (current-time))`	`36`

Table 9.2 *Example use of some of the LAML time library functions*

Strictly speaking, the abstractions which are applied in the example above, are not functions. They all depend on some state, which is updated every second due to the fact that time does not stand still.

9.8. References

[timelib]	Manual of the LAML time library http://www.cs.auc.dk/~normark/scheme/distribution/laml/lib/man/time.html
[-]	Foldoc: let http://wombat.doc.ic.ac.uk/foldoc/foldoc.cgi?query=let
[-]	R5RS: Binding Constructs (let, let*, letrec) http://www.cs.auc.dk/~normark/prog3-03/external-material/r5rs/r5rs-html/r5rs_33.html

10. Conditional expressions

In this chapter we will be interested in conditional expressions a la `if` and `cond`.

10.1. Conditional expressions

Lecture 3 - slide 10

In this section we introduce `if` and `cond` , both at the syntactic level (through Syntax 10.1 and Syntax 10.2) and at the semantic level (below the syntax boxes).

`if` and `cond` are special forms which evaluate their expressions according to the value of one or more boolean selectors

`if` and `cond` are not control structures when applied in the functional paradigm

Control structures belong to the imperative paradigm. In the functional paradigm, `if` and `cond` are used in conditional expressions. By that we mean expressions, of which subexpressions are selected for evaluation based on one or or more boolean selectors.

```
(if bool-expr expr₁ expr₂)
```

Syntax 10.1

```
(cond (bool-expr₁ expr₁)
      ...
      (bool-exprₖ exprₖ)
      (else exprₖ₊₁))
```

Syntax 10.2

- `if` evaluates $expr_1$ if `bool-expr` is true, and $expr_2$ if `bool-expr` is false
- `cond` evaluates the first expression $expr_i$ whose guarding $bool\text{-}expr_i$ is true. If $bool\text{-}expr_1$, ..., $bool\text{-}expr_k$ are all false, the value of `cond` becomes the value of $expr_{k+1}$

Exercise 3.1. *HTML Header functions*

This is a small exercise that aims at construction of slightly different header functions than those provided by the native header functions `h1`, ..., `h6`.

Define a function `(header level)` which takes a parameter `level`. The header function should return the similar basic header function provided that n is between one and six. If n is outside this interval, we want header to return the *identity function* of one parameter.

It means that ((header 3) "Header text") is equal to (h3 "Header text") and that ((h 0) "Header text") is just "Header text".

Hint: Arrange the header functions in a list, and let `header` select the appropriate header function from this list.

Define a variant of `header` which returns a native header function if it receives a single parameter (level), and which returns the value, such as, ((header 3) "Header text"), if it receives both a level parameter and a header text string.

10.2. Examples with `if`

Lecture 3 - slide 11

The examples in the table below gives web-related examples of `if`.

Expression	Value

`(body`
`  (if (string=? (weekday (current-time)) "Wednesday")`
`      (p (em "Remember the Thursday meeting tomorrow!"))`
`      '( ))`
 	
`  (h1 "Schedule")`
 	
`  (p "..."))`
	Remember the Thursday meeting tomorrow!
	Schedule
	...

`(body`
`  (p (if (string=? (weekday (current-time)) "Wednesday")`
`         (em "Remember the Thursday meeting tomorrow!")`
`         '( )))`
 	
`  (h1 "Schedule")`
 	
`  (p "..."))`
	Remember the Thursday meeting tomorrow!
	Schedule
	...

Table 10.1 *Examples using an if conditional expression on a Wednesday. In both examples we extract the weekday (a string) from the current time. If it is a Wednesday we emit a paragraph which serves as a reminder of a meeting the following day. If not executed on a Wednesday, we do not want any special text. We achieve this by returning the empty list, which is spliced into the the body context (in the first example) and into the paragraph context (in the second example). The splicing is a result of the handling of lists by the HTML mirror functions in LAML. The two examples differ slightly. In the first example the `if` is placed on the outer level, feeding information to `body`. In the second row, the `if` is placed at an inner level, feeding information to the `p` function. The two examples also give slightly different results. Can you characterize the results?*

10.3. Example with `cond`: `leap-year?`

Lecture 3 - slide 12

The leap year function is a good example of a function, which calls for use of a `cond` conditional. It would, of course, also be possible to program the function with nested `if` expressions.

```
(define (leap-year? y)
  (cond ((= (modulo y 400) 0) #t)
        ((= (modulo y 100) 0) #f)
        ((= (modulo y 4) 0) #t)
        (else #f)))
```

Program 10.1 *The function `leap-year?`. The function returns whether a year y is a leap year. For clarity we have programmed the function with a conditional. In this case, we can express the leap year condition as a simple boolean expression using `and` and `or`. We refer to this variation below, and we leave it to you to decide which version you prefer.*

It is also possible to program the leap year function with simple, boolean arithmetic. This is shown below. It is probably easier for most of us to understand the version in Program 10.1 because it is closer to the way we use to formulate the leap year rules.

```
(define (leap-year? y)
  (or (= (modulo y 400) 0)
      (and (= (modulo y 4) 0)
           (not (= (modulo y 100) 0)))))
```

Program 10.2 *The function leap year programmed without a conditional.*

In the web version of this material we provide a link to the same elucidator as already discussed in Section 9.5. The elucidator shows the leap year function in a larger context.

10.4. Example with cond: `american-time`

Lecture 3 - slide 13

In this section we will study an extended example of the use of cond. We carry out a calculation of 'American time', such as 2:30PM given the input of 14 30 00. There are several different cases to consider, as it appears in Program 10.3.

```
(define (american-time h m s)
  (cond ((< h 0)
           (laml-error "Cannot handle this hour:" h))

        ((and (= h 12) (= m 0) (= s 0))
             "noon")

        ((< h 12)
           (string-append
             (format-hour-minutes-seconds h m s)
             " " "am"))

        ((= h 12)
           (string-append
             (format-hour-minutes-seconds h m s)
             " " "pm"))

        ((and (= h 24) (= m 0) (= s 0))
             "midnight")

        ((<= h 24)
           (string-append
             (format-hour-minutes-seconds (- h 12) m s)
             " " "pm"))

        (else
           (laml-error "Cannot handle this hour:" h))))
```

Program 10.3 *The function `american-time`. The function returns a string displaying the 'am/pm/noon' time given hour h, minute m, and seconds s.*

In the web version of the material - slide or annotated slide view - we include a version of the program which includes the helping functions `format-hour-minutes-seconds` and `zero-pad-string`.

10.5. Example with `cond`: `as-string`

Lecture 3 - slide 14

As a final example with cond, we show `as-string`, which is a function from the general LAML library. Given an almost arbitrary piece of data the function will attempt to convert it to a string. Similar functions named `as-number`, `as-symbol`, and `as-boolean` exist in the library, cf. [generallib].

```
(define (as-string x)
  (cond ((number? x) (number->string x))
        ((symbol? x) (symbol->string x))
        ((string? x) x)
        ((boolean? x)
            (if x "true" "false"))   ; consider "#t" and "#f" as alternatives
        ((char? x) (char->string x))
        ((list? x)
            (string-append "("
               (string-merge (map as-string x) (make-list (- (length x) 1) " "))
               ")"))
        ((vector? x)
          (let ((lst (vector->list x)))
            (string-append "#("
               (string-merge (map as-string lst) (make-list (- (length lst) 1) "
"))
               ")")))
        ((pair? x)
            (string-append "("
               (apply string-append
                  (map (lambda (y) (string-append (as-string y) " ")) (proper-
part x))
               )
               " . " (as-string (first-improper-part x))
               ")"))
        (else "??")))
```

Program 10.4 *The function as-string converts a variety of Scheme data types to a string. This function makes use of the fact that any kind of data can be passed to the function, without intervening static type check. At run time we dispatch on the type of x. The function string-merge is discussed later in this section, cf. the reference from this page. The function* `as-string`*, and its sibling functions* `as-number`, `as-char`, `as-symbol`*, and* `as-list` *are used heavily in all LAML software. The functions are convenient because they do not need to know the type of the input data. In functional languages with static type checking, we cannot program these functions as shown above. In these language we could overload the function name* `as-string`*, and underneath define a number of individual functions each taking a particular type of input.*

10.6. References

[generallib]	Manual of the LAML general library http://www.cs.auc.dk/~normark/scheme/distribution/laml/lib/man/general.html
[-]	R5RS: cond http://www.cs.auc.dk/~normark/prog3-03/external-material/r5rs/r5rs-html/r5rs_32.html
[-]	R5RS: if http://www.cs.auc.dk/~normark/prog3-03/external-material/r5rs/r5rs-html/r5rs_29.html

11. Recursion and iteration

Recursion plays an important role for non-trivial functional programs. One of the reasons is that recursive data structures are used heavily in functional programs. Just take, as example, linear lists, cf. Chapter 6.

As another reason, most non-trivial needs some kinds of repeating structures - iteration. In our style of Scheme programming, we use recursive functions for iterative purposes. In this chapter we will see how this can be done without excessive use of memory resources.

And as before we attempt to illustrate also this topic with examples from the web domain.

11.1. Recursion

Lecture 3 - slide 16

In this section we characterize the basic ideas of recursion, and the kinds of problem solving which are aided by recursion.

Recursive functions are indispensable for the processing of recursive data structures

Recursion is an algorithmic program solving idea that involves solution of subproblems of the same nature as the overall problem

- Given a problem P
  - If P is trivial then solve it immediately
  - If P is non-trivial, divide P into subproblems P_1 ,..., P_n
  - Observe if P_i (i=1..n) are of the same nature as P
  - If so, use the overall problem solving idea on each of P_1 ... P_n
  - Combine solutions of the subproblems P_1 ... P_n to a solution of the overall problem P

The problem solving technique sketched here is called *divide and conquer*. It is not all *divide and conquer* problems that involve recursion. But many do in fact. Recursion comes into play when the subproblems P_1 ... P_n turn out to be of the same nature as the overall problem, and as such they can be solved by the same 'medicine' as used for the overall problem P.

We would like to refer the reader to an ECIU material on recursion, which in a more careful way discusses and illustrates the ideas [eciu-recursion]. Notice that there is some overlap with the ECIU material and the material you are reading now.

11.2. List processing

Lecture 3 - slide 17

We have already discussed lists as a recursive data type in Section 6.1. In this section we will give an extended LAML related example of recursive list processing in Scheme.

A list is a recursive data structure

As a consequence list processing is done via recursive functions

We illustrate list processing by extracting attribute values from a LAML attribute *property list*

The function `find-href-attribute` in Program 11.1 extracts the href attribute value from an attribute property list. Property lists have already been discussed in Section 6.6.

Notice the recursive nature of the function find-href-attribute. The recursive call is highlighted with red color.

It happens to be the case that the function in Program 11.1 is tail recursive, cf. the discussion in Section 11.5.

```
; Return the href attribute value from a property list
; Return #f if no href attribute is found.
; Pre-condition: attr-p-list is a property list -
; of even length.
(define (find-href-attribute attr-p-list)
 (if (null? attr-p-list)
     #f
     (let ((attr-name (car attr-p-list))
           (attr-value (cadr attr-p-list))
           (attr-rest (cddr attr-p-list)))
      (if (eq? attr-name 'href)
          attr-value
          (find-href-attribute attr-rest)))))
```

Program 11.1 *A function for extraction of the href attribute from a property list.*

To stay concrete, we show an example of using the function `find-href-attribute` in Program 11.2.

```
1> (define a-clause
    (a 'id "myid" 'class "myclass" 'href "http://www.cs.auc.dk"))

2> a-clause
(ast "a" ()
  (id "myid" class "myclass" href "http://www.cs.auc.dk" shape "rect")
  double xhtml10-strict)
```

```
3> (render a-clause)
"<a id = \"myid\" class = \"myclass\" href = \"http://www.cs.auc.dk\"></a>"

4> (define attr-list (ast-attributes a-clause))

5> attr-list
(id "myid" class "myclass" href "http://www.cs.auc.dk" shape "rect")

6> (find-href-attribute attr-list)
"http://www.cs.auc.dk"
>
```

Program 11.2 *An example with property lists that represent HTML attributes in LAML. As the last interaction, we see the function find-href-attribute in play.*

11.3. Tree processing (1)

Lecture 3 - slide 18

Trees are another classical example of recursive data types.

In this section we show a web document and its internal structure. In Section 11.4 we show how to traverse this document, by means of tree traversal, with the purpose of extracting and collecting all URLs from href attributes of anchor elements in the document.

A tree is a recursive data structure
We illustrate how to extract information from an HTML syntax tree

The LAML document in Program 11.3 shows a web document, in which we have highlighted all the anchor elements - the a elements. The tree structure in Figure 11.1 shows the hierarchical composition of the document, in terms of HTML elements. In the web version of the material - slide or annotated slide view - you can also access the actual abstract syntax tree - AST - which is the internal document representation of LAML. We do not include it in this version of the material because it is relatively long.

```
(load (string-append laml-dir "laml.scm"))
(laml-style "simple-xhtml1.0-strict-validating")

(write-html 'raw
(html
 (head (title "Demo Links"))
 (body
  (p "ACM has a useful" (a 'href "http://www.acm.org/dl" "digital library") )

  (p "The following places are also of
     interest:")


  (ul
   (li (a 'href "http://www.ieee.org/ieeexplore/" "The IEEE"))
   (li "The" (a 'href "http://www.w3c.org" "W3C") "for web standards")
   (li (a 'href "http://link.springer.de/link/service/series/0558/"
           "Lecture Notes in Computer Science")))

  (p "Kurt Nørmark" (br)
     (a 'href "http://www.cs.auc.dk" "Department of Computer Science") (br)
     (a 'href "http://www.auc.dk" "Aalborg University")))))

(end-laml)
```

Program 11.3   *A sample web document with a number of links. The link forms - represented by a elements - are highlighted.*

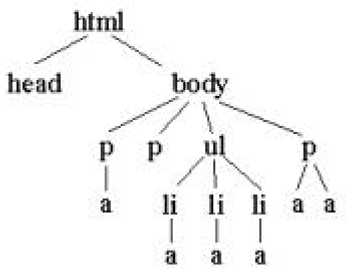

Figure 11.1   *The syntax tree of the web document with the root made up by the `html` element.*

11.4. Tree processing (2)

Lecture 3 - slide 19

We continue the example from Section 11.3 .

In Program 11.4 we show the function `extract-links` . The function is indirectly recursive via the function `extract-links-ast-list` .

```
; Return a list of URLS as located in the a elements of ast.
(define (extract-links ast)
 (if (ast? ast)
     (let ((name (ast-element-name ast))
           (subtrees (ast-subtrees ast))
          )
       (if (equal? name "a")
           (let ((href-attr-value
                  (find-href-attribute (ast-attributes ast))))
             (if href-attr-value (list href-attr-value) '()))
           (extract-links-ast-list subtrees)))
     '()))

; Return a list of URLS as located in the a elements of
; the list of ast's as passed in ast-list.
(define (extract-links-ast-list ast-list)
 (if (null? ast-list)
     '()
     (append
      (extract-links (car ast-list))
      (extract-links-ast-list (cdr ast-list)))))
```

Program 11.4 *The link extraction functions.*

The `extract-links` function above traverses the internal AST structure of a web document. When an anchor element is encountered, when `(equal? name "a")` becomes true, we collect the `href` attribute by means of the function `find-href-attribute`, which we described in Section 11.2, see Program 11.1. In the cases where we do not encounter an anchor element, the call `(extract-links-ast-list subtrees)` causes traversal of the list of subtrees.

In the dialogue shown below we illustrate how to extract the URLs from a demo document, which we assume is identical with the document in Program 11.3.

```
1> (define doc-ast
    (html
     (head (title "Demo Links"))
     (body
       ...)))

2 > (extract-links doc-ast)
("http://www.acm.org/dl" "http://www.ieee.org/ieeexplore/" "http://www.w3c.org"
 "http://link.springer.de/link/service/series/0558/" "http://www.cs.auc.dk"
 "http://www.auc.dk")
```

Program 11.5 *A link extraction dialogue.*

Exercise 3.2. *The function outline-copy*

Program a function `outline-copy` which makes a deep copy of a list structure. Non-list data in the list should all be translated to a symbol, such as '-. You should be able to handle both proper lists and improper lists.

As an example:

```
(outline-copy '((a b c) (d e . f) (h i))) =>
   ((- - -) (- - . -) (- -))
```

11.5. Recursion versus iteration

Lecture 3 - slide 20

The purpose of this section is to introduce and not least motivate the idea of tail recursion.

Recursive functions are - modulo use of memory resources - sufficient for any iterative need

Tail recursive functions in Scheme are memory efficient for programming of any iterative process

Tail recursion is a variant of recursion in which the recursive call takes place without contextual, surrounding calculations in the recursive function.

A tail call is the last 'thing' to be done before the function returns. Therefore there is no need to maintain any activation record of such a recursive call - we can reuse the callers activation record.

The main source of insight to understand tail recursiveness is a series of images, which are available in the web version of the material (slide view). You should definitively consult this before you go on in this material.

11.6. Example of recursion: `number-interval`

Lecture 3 - slide 21

We provide an example of a recursive function, namely `number-interval`.

The function `number-interval` returns a list of integers from a lower bound to an upper bound

The version of `number-interval` shown in Program 11.6 is not tail recursive. The rewriting of the function in Program 11.7 is tail recursive however. Notice that the function in Program 11.7 needs a helping function, `number-interval-iter-help`, with an appropriate parameter profile.

```
(define (number-interval f t)
 (if (<= f t)
     (cons f (number-interval (+ f 1) t))
    '()))
```

Program 11.6 *The function* ***`number-interval`*** *from the general LAML library. This function returns a list of* t-f+1 *numbers from* ***`f`*** *to* ***`t`*** *.Try it out!.*

```
(define (number-interval-iter f t)
  (reverse (number-interval-iter-help f t '())))


(define (number-interval-iter-help f t res)
  (if (<= f t)
      (number-interval-iter-help (+ f 1) t (cons f res))
      res))
```

Program 11.7 *The function* ***`number-interval-iter`*** *is an iterative, tail recursive variant of* ***`number-interval`***.

We show below a couple of concrete applications of the functions in Program 11.6 and Program 11.7.

```
1> (number-interval 1 10)
(1 2 3 4 5 6 7 8 9 10)

2> (number-interval-iter 10 20)
(10 11 12 13 14 15 16 17 18 19 20)

3> (number-interval-iter 20 10)
()
```

Program 11.8 *A sample dialogue with the number interval functions.*

Exercise 3.3. *The append function*

The function `append`, which is a standard Scheme function, concatenates two or more lists. Let us here show a version which appends two lists:

```
  (define (my-append lst1 lst2)
       (cond ((null? lst1) lst2)
             (else (cons (car lst1) (my-append (cdr lst1) lst2)))))
```

We will now challenge ourselves by programming an iterative solution, by means of tail recursion. We start with the standard setup:

```
(define (my-next-append lst1 lst2)
  (my-next-append-1 lst1 lst2 ...))
```

where `my-next-append-1` is going to be the tail recursive function:

```
(define (my-next-append-1 lst1 lst2 res)
  (cond ((null? lst1) ...)
        (else (my-next-append-1 (cdr lst1) lst2 ...))))
```

Fill out the details, and try out your solution.

Most likely, you will encounter a couple of problems! Now, do your best to work around these problems, maybe by changing aspects of the templates I have given above.

One common problem with iterative solutions and tail recursive functions is that lists will be built in the wrong order. This is due to our use of `cons` to construct lists, and the fact that `cons` operates on the front end of the list. The common medicine is to reverse a list, using the function `reverse`, either on of the input, or on the output.

Exercise 3.4. *A list replication function*

Write a tail recursive function called { t replicate-to-length}, which in a cyclic way (if necessary) replicates the elements in a list until the resulting list is of certain exact length. The following serves as an example:

```
(replicate-to-length '(a b c) 8) =>
(a b c a b c a b)

(replicate-to-length '(a b c) 2) =>
(a b)
```

In other words, in `(replicate-to-length lst n)`, take elements out of `lst`, cyclically if necessary, until you reach `n` elements.

11.7. Examples of recursion: string-merge

Lecture 3 - slide 22

This section and the next give yet other examples of recursive functions. We start with `string-merge`.

The function `string-merge` zips two lists of strings to a single string. The lists are not necessarily of equal lengths

```
(define (string-merge str-list-1 str-list-2)
 (cond ((null? str-list-1) (apply string-append str-list-2))
       ((null? str-list-2) (apply string-append str-list-1))
       (else (string-append
                 (car str-list-1) (car str-list-2)
                 (string-merge (cdr str-list-1) (cdr str-list-2))))))
```

Program 11.9 *The recursive function string-merge. Notice that this function is a general recursive function. The recursive call, emphasized above, is not in a tail position, because of the embedding in string-append.*

The function in Program 11.9 not tail recursive. To remedy this weakness, we make another version which is. It is shown in Program 11.10.

As it is characteristic for all tail recursive functions, the state of the iteration needs to be represented in the parameter list, here in the helping function called `merge-iter-helper`. The necessary state for string merging purpose is reduced to the resulting, merged string - the `res` parameter.

```
(define (string-merge-iter str-list-1 str-list-2)
 (merge-iter-helper  str-list-1 str-list-2 ""))

(define (merge-iter-helper str-list-1 str-list-2 res)
 (cond ((null? str-list-1)
          (string-append res (apply string-append str-list-2)))
       ((null? str-list-2)
          (string-append res (apply string-append str-list-1)))
       (else (merge-iter-helper
                (cdr str-list-1)
                (cdr str-list-2)
                (string-append
                    res (car str-list-1) (car str-list-2))))))
```

Program 11.10 *A tail recursive version of string-merge.*

In the LAML software, the function `string-merge` is used in several contexts. One of them is in the function `list-to-string` , which we show in Program 11.11 We could in fact have applied `list-to-string` in the function `as-string`, which we discussed in Program 10.4.

```
(define (list-to-string str-lst separator)
  (string-merge
     str-lst
     (make-list (- (length str-lst) 1) separator)))
```

Program 11.11 *An application of string-merge which converts a list of strings to a string with a given separator. This is a typical task in a web program, where a list of elements needs to be aggregated for HTML presentation purposes. Notice the merging of a list of n elements with a list of length n-1. The function make-list is another LAML function; (make-list n el) makes a list of n occurrences of el.*

11.8. Examples with recursion: `string-of-char-list?`

Lecture 3 - slide 23

The last example in this chapter is a boolean function that can check if a string is formed by the characters from a given alphabet.

The function `string-of-char-list?` is a predicate (a boolean function) that finds out if a string is formed exclusively by characters from a particular alphabet.

```
(define (string-of-char-list? str char-list)
  (string-of-char-list-1? str char-list 0 (string-length str)))

(define (string-of-char-list-1? str char-list i lgt)
  (if (= i lgt)
      #t
      (and (memv (string-ref str i) char-list)
           (string-of-char-list-1? str char-list (+ i 1) lgt))))
```

Program 11.12 *The function string-of-char-list? which relies on the tail recursive function string-of-char-list-1?. The function string-of-char-list-1? iterates through the characters in str, via the controlling parameters i and lst.*

The predicates `blank-string?` and `numeric-string?` in Program 11.13 are very useful for many practical purposes. The first function checks if a string represents white space only. The latter function checks if a string represents a decimal integer.

```
;; A list of characters considered as blank space characters
(define white-space-char-list
   (list #\space (as-char 13) (as-char 10) #\tab))

;; Is the string str empty or blank (consists of white space)
(define (blank-string? str)
  (or (empty-string? str)
      (string-of-char-list? str white-space-char-list)))

;; Returns if the string str is numeric.
;; More specifically, does str consist exclusively of the
;; ciffers 0 through 9.
(define (numeric-string? str)
  (string-of-char-list? str
    (list #\0 #\1 #\2 #\3 #\4 #\5 #\6 #\7 #\8 #\9 )))
```

Program 11.13 *Applications of string-of-char-list?. The function blank-string? determines if a string is formed entirely of white space characters. The function numeric-string? is a predicate that returns true if the string consists exclusively of decimal digits. This is, for instance, useful to check the form input of dates and time in some server-based web applications. The version of numeric-string? in the lib/general.scm of LAML is slightly more general than the version shown above (it allows + or - signs as well, depending on an optional parameter).*

Exercise 3.5. *Sublists of a list*

In this exercise we will program a function `front-sublist` which returns the first n elements of a list. The signature (the head) of the function should be `(front-sublist lst n)` where `lst` is a list and `n` is a number. As a precondition it can be assumed that `lst` is a proper list and that `n` is a non-negative integer. As a postcondition we want to guarantee that the length of the result is `n`.

As an example

```
(front-sublist '(a b c d e) 3)  =>
(a b c)

(front-sublist '(a b c d e) 6)  =>
ERROR
```

First, identify the extreme, border cases and be sure that you know how to handle these. Next, program the function with due respect to both the precondition and the postcondition. Next, test the function carefully in a dialogue with your Scheme system.

Given the function `front-sublist` we will program the function `sublists`, which breaks a proper list into a list of sublists of some given size. As an example

```
(sublists '(a b c d e f) 3) =>
((a b c) (d e f))
```

Program the function `sublists` with use of `front-sublist`. Be careful to prepare you solution by recursive thinking. It means that you should be able to break the overall problem into a smaller problem of the same nature, as the original problem. You are free to formulate both preconditions and postconditions of the function sublists, such that existing function front-sublist fits well.

Hint: The Scheme function `list-tail` is probably useful when you program the function `sublists`.

A table can be represented as a list of rows. This is, in fact, the way tables are represented in HTML. The `tr` tag is used to mark each row; the `td` tag is used to mark each cell. The `table` tag is used to mark the overall table. Thus, the list of rows `((a b c) (d e f))` will be marked up as:

```
<table>
  <tr> <td>a</td> <td>b</td> <td>c</td> </tr>
  <tr> <td>d</td> <td>e</td> <td>f</td> </tr>
</table>
```

Write a Scheme function called `table-render` that takes a list of rows as input (as returned by the function `sublists`, which we programmed above) and returns the appropriate HTML rendering of the rows. Use the LAML mirror functions `table`, `tr`, and `td`. Be sure to call the LAML function `xml-render` to see the textual HTML rendering of the result, as opposed to

LAML's internal representation.

Notice: During the course we will see better and better ways to program `table-render`. Nevertheless, it is a good idea already now to program a first version of it.

11.9. References

[-]	Foldoc: iteration http://wombat.doc.ic.ac.uk/foldoc/foldoc.cgi?query=iteration
[-]	R5RS: Proper tail recursion http://www.cs.auc.dk/~normark/prog3-03/external-material/r5rs/r5rs-html/r5rs_22.html
[-]	The HTML version of the web document that illustrates tree traversal external-material/ast-example.html
[-]	AST functions in LAML http://www.cs.auc.dk/~normark/scheme/distribution/laml/lib/xml-in-laml/man/xml-in-laml.html#SECTION4
[eciu-recursion]	Recursion - an ECIU material http://www.cs.auc.dk/~normark/eciu-recursion/html/recit.html
[-]	Foldoc: recursion http://wombat.doc.ic.ac.uk/foldoc/foldoc.cgi?query=recursion

12. Example of recursion: Hilbert Curves

In this chapter we will give examples of recursive curves. The examples are taken from the ECIU material on recursion [eciu-recursion] which we have mentioned earlier on.

The primary value of this chapter is the animations, which show the building of the Hilbert Curves. These animations must be approached in the web version of the material.

In this paper version of the material we only give a shallow and superficial coverage. You are referred to the web version to get the real outcome.

12.1. Hilbert Curves

Lecture 3 - slide 26

The *Hilbert Curve* is a space filling curve that visits every point in a square grid

At this spot in the web version of the material you will find a Hilbert curve of order 5, i.e, a quite complicated curve.

The path taken by a Hilbert Curve appears as a sequence - or a certain iteration - of **up**, **down**, **left**, and **right**.

12.2. Building Hilbert Curves of order 1

Lecture 3 - slide 27

Here we will study the recursive composition of the most simple Hilbert Curve.

This section is only meaningful in the web version of the material - please take a look at it.

12.3. Building Hilbert Curves of order 2

Lecture 3 - slide 28

Here will study the recursive composition of Hilbert Curves in additional details.

This section is only meaningful in the web version of the material - please take a look at it.

12.4. Building Hilbert Curves of order 3

Lecture 3 - slide 29

In the same way we made a Hilbert Curve of order 2, we will here see how a Hilbert Curve of order 3 is made.

This section is only meaningful in the web version of the material - please take a look at it.

12.5. Building Hilbert Curves of order 4

Lecture 3 - slide 30

In the same way we made a Hilbert Curve of order 3, we will here see how a Hilbert Curve of order 4 is made. This is the final development along these lines in this material.

This section is only meaningful in the web version of the material - please take a look at it.

12.6. A program making Hilbert Curves

Lecture 3 - slide 31

Given our understanding of Hilbert Curves obtained from the previous pages, we will now study a computer program that generates Hilbert Curves of order n, where n is any non-negative number.

We will here discuss a concrete program which draws Hilbert Curves of order n

The program below, Program 12.1 shows the `hilbert` function, which returns a rendering of Hilbert Curves.

```
(define (hilbert n turn)
 (cond ((= n 0) (empty-hilbert-curve))
       ((> n 0)
         (cond
             ((eq? turn 'up)
               (concat-path
                 (hilbert (- n 1) 'right)
                 (up-line)
                 (hilbert (- n 1) 'up)
                 (right-line)
                 (hilbert (- n 1) 'up)
                 (down-line)
                 (hilbert (- n 1) 'left) ))

             ((eq? turn 'left)
               (concat-path
                 (hilbert (- n 1) 'down)
                 (left-line)
                 (hilbert (- n 1) 'left)
                 (down-line)
                 (hilbert (- n 1) 'left)
                 (right-line)
                 (hilbert (- n 1) 'up)))

             ((eq? turn 'right)
               (concat-path
                 (hilbert (- n 1) 'up)
                 (right-line)
                 (hilbert (- n 1) 'right)
                 (up-line)
                 (hilbert (- n 1) 'right)
                 (left-line)
                 (hilbert (- n 1) 'down)))

             ((eq? turn 'down)
               (concat-path
                 (hilbert (- n 1) 'left)
                 (down-line)
                 (hilbert (- n 1) 'down)
                 (left-line)
                 (hilbert (- n 1) 'down)
                 (up-line)
                 (hilbert (- n 1) 'right))) ))))
```

Program 12.1 *The function `hilbert` programmed in Scheme as a functional program. The function returns the path of the Hilbert Curver of order n. The parameter `turn` determines the rotation of the curve. In the top level call we ask for an upward Hilbert Curve: As an example, `(hilbert 3 'up)` produces an upward Hilbert Curve of order 3. The red fragments are responsible for all line drawing. The blue fragments represent all the recursive calls of the `hilbert` function. Finally, the green fragment represent the level 0 'basis' case. The level 0 case returns the empty Hilbert Curve, which is literally empty (no drawing at all - no contribution to the resulting path). What does it mean that the the program is a functional program? Well, it basically means that `hilbert` returns a value which can be rendered somehow by another function or procedure. The value returned is a path, composed by concat-path. The `hilbert` function does not carry out any action itself.*

The actual rendering of a Hilbert Curve is done by use of SVG stuff [svg]. SVG is a W3C standard for Scalable Vector Graphics. In case you want to get started with SVG we will recommend that you start with an excellent tutorial made by Ivan Herman, F.R.A. Hopgood, and D.A. Duce [svg-tutorial].

In the web version of the material - in slide or annotated slide view - you will have access to the additional implementation details of the primitives used in Program 12.1.

13. Continuations

Continuations represent one of the advanced concepts in Scheme. In this section we will introduce continuations, and we will show some examples of their use within the functional programming paradigm.

13.1. Introduction and motivation

Lecture 3 - slide 33

We start by motivating our interest in continuations. One part of the story is the usefulness of a mechanism that allows us to 'jump out of a deep subexpression'. Another part is the possibility of controlling and manipulating the 'remaining part of the calculation' relative to some given control point.

It is sometimes necessary to escape from a deep expression, for instance in an exceptional case

We are interested in a primitive which allows us to control the remaining part of a calculation - a so-called *continuation.*

- Exit or exception mechanism:
    - The need to abandon some deep evaluation
- Continuation
    - Capturing of continuations
    - Exploring new control mechanisms by use of continuations

Scheme support first class continuations dressed as functions

Both needs mentioned above are handled by first class continuations in Scheme.

13.2. The catch and throw idea

Lecture 3 - slide 34

In this section, and section Section 13.3 we explore a catch and throw escape mechanism. This mechanism is used in Common Lisp, but it is not directly available in Scheme. As we will see in Section 13.8 first class continuations can easily play the role of catch and throw. A similar Scheme-base example is given in Section 13.9.

`Catch` and `throw` provides for an intuitively simple escape mechanism on functional ground

We introduce an imaginary syntax of catch and throw, see Syntax 13.1 and Syntax 13.2. The meaning is intended to be that catch identifies an expression, `catch-expr` with an id. id is a symbol. Within the expression, or within a function called directly or indirectly in catch-expr, we may encounter a throw form, which mention the id of the catch. The value of the thrown expression, throw-expr, is passed back along the chain of calls to the catcher, and it becomes the return value of the catch form. If no throw form with an appropriate id is met during the evaluation of the catch form, the value of the catch form just becomes the value of `catch-expr`.

```
(catch id catch-expr)
```

Syntax 13.1

```
(throw id throw-expression)
```

Syntax 13.2

Scheme does not support `catch` and `throw`

Rather Scheme supports a much more powerful mechanisms based on continuations

In case you are interested in more precise details of catch and throw in Common Lisp, you should consult the book about Common Lisp, [cltl], (full text on the web). More specifically you should consult the chapter about dynamic non-local exists [cltl-non-local-exists].

13.3. A catch and throw example

Lecture 3 - slide 35

We now study an concrete, real-life example of catch and throw. This is not a Scheme example.

Exit from a list length function in case it discovers a non-empty tail of the list

The function `list-length` returns the length of the list. The function counts the cons cells. If we encounter an improper list (a list without an empty list in the end of the cdr chain, see Section 6.2) we wish to return the symbol `improper-list`. In order to provide for this we set of a catcher around the a local function, list-length1, which does the real job. The function list-length calls list-length1. If list-length1 encounters an improper termination of the list, it throws the symbol improper-list to the catcher, which returns it. If not, it just returns the count, which also is returned by catch via the letrec form.

```
(define (list-length lst)
  (catch 'exit
    (letrec ((list-length1
               (lambda (lst)
                 (cond ((null? lst) 0)
                   ((pair? lst) (+ 1 (list-length1 (cdr lst))))
                   (else (throw 'exit 'improper-list))))))
       (list-length1 lst))))
```

Program 13.1 *An example using catch and throw. Please notice that the example is not a proper Scheme program. Catch and throw are not defined in Scheme.*

13.4. The intuition behind continuations

Lecture 3 - slide 36

The rest of this chapter is about concepts that are fully supported in Scheme.

We start with an overall definition of a continuations. Then follows some intuitive examples of continuations of given expressions within given contexts (surrounding expressions).

A *continuation* of the evaluation of an expression E in a surrounding context C represents the future of the computation, which waits for, and depends on, the value of E

It may very well be difficult to grasp the intuition of continuations. We hope the following table will help you. It is intended to explain the intuitive understanding of the continuations of the blue, emphasized expressions in the leftmost column.

Context C and expression E	**Intuitive continuation of E in C**
`(+ 5 (* 4 3))`	*The adding of 5 to the value of E*
`(cons 1 (cons 2 (cons 3 '())))`	*The consing of 3, 2 and 1 to the value of E*

`(define x 5)`
`(if (= 0 x)`
`    'undefined`
`    (remainder (* (+ x 1) (- x 1)) x))`
	The multiplication of E by x - 1 followed by a division by x

Table 13.1 *An intuitive understanding of continuations of an expression in some context.*

13.5. Being more precise

Lecture 3 - slide 37

Instead of relying of an informal understanding of continuations we will now introduce lambda expressions that represent the continuations.

We can form a lambda expression that to some degree represents a continuation

The continuation of the expression `(* 3 4)` within `(+ 5 (* 3 4))` is a function that adds 5. Written precisely, it is the function `(lambda (e) (+ 5 e))`. The other two examples of Table 13.2 (corresponding to the second and third rows) are similar.

Context C and expression E	**The continuation of E in C**
`(+ 5 (* 4 3))`	`(lambda (e) (+ 5 e))`
`(cons 1 (cons 2 (cons 3 '())))`	`(lambda (e) (cons 1 (cons 2 (cons 3 e))))`

`(define x 5)`
`(if (= 0 x)`
`    'undefined`
`    (remainder (* (+ x 1) (- x 1)) x))`
	`(lambda (e) (remainder (* e (- x 1)) x))`

Table 13.2 *A more precise notation of the continuations of E*

The representation of continuations with lambda expressions is part of the truth, but not the whole truth. The problem is that if we activate the continuation, by calling the function that represents it, it will return the normal way, and its calling context will finish the evaluation the normal way. We do not want that. Therefore a mechanism known as *escape functions* are invented and used. An escape function ignores its context in every call. We will not go into the technical details of escape functions in this text. The interested reader should consult [Springer89].

13.6. The capturing of continuations

Lecture 3 - slide 38

It is now time to introduce the Scheme primitive that allows us to capture a continuation.

Scheme provides a primitive that captures a continuation of an expression E in a context C

The primitive is called `call-with-current-continuation`, or `call/cc` as a short alias

`call/cc` takes a parameter, which is a function of one parameter.

The parameter of the function is bound to the continuation, and the body of the function is E

We will use the brief form call/cc in our examples.

Context C and the capturing
`(+ 5 (call/cc (lambda (e) (* 4 3)) ))`
`(cons 1 (cons 2 (cons 3 (call/cc (lambda (e) '()) ))))`

`(define x 5)`
`(if (= 0 x)`
`    'undefined`
`    (remainder (* (call/cc (lambda (e) (+ x 1)) ) (- x 1))`
`x))`

Table 13.3 *Use of call/cc and capturing of continuations.*

We elaborate the examples from Table 13.1 and Table 13.2. In the first line of Table 13.3 we capture the continuation of `(* 4 3)` in `(+ 5 (* 4 3))`. In the second line we capture the continuation of `'()` in `(cons 1 (cons 2 (cons 3 '())))`. And in the third line we capture the continuation of `(+ x 1)` in the if expression. This is the same as the continuation of the `(+ x 1)` in the `remainder` expression.

One thing is capturing continuations. Another is to make good use of them. Table 13.3 does not illustrate the latter aspect at all. This is seen by the fact the the continuations, bound to the names `e` in all three examples, are not used.

It should be noticed that a captured continuation is dressed like a function. Somehow we can think of a continuation as a 'wolf in sheep's clothing'. A continuation is activated in the same way as a function is called. However, the continuation is defined (captured) differently than the way functions are defined. Notice also that continuations inherit their first class status from functions, see Section 8.6.

13.7. Capturing, storing, and applying continuations

Lecture 3 - slide 39

In this section we will illustrate applications of the captured continuations. Once captured, we assign the continuations to a global variable `cont-remember`. We assume that `cont-remember` has been defined before any of the expressions in table Table 13.4 are evaluated. Use of assignments is of course not functional programming, but it provides an easy way to illustrate the working and the nature the captured continuations. Later in this section we will show uses of continuations in functional programming. For a brief review of imperative programming in Scheme the reader is referred to Chapter 29.

We here show capturing, imperative assignment, and a subsequent application of a continuation

In table Chapter 29 below we show the context expression C, its value, the application of the captured continuation that we have stored in the variable `cont-remember`, and the value of the application. We explain the rightmost column below the table.

Context C and expression E	**Value of C**	**Application of continuation**	**Value**

`(+ 5`
` (call/cc`
`  (lambda (e)`
`   (set! cont-remember e)`
`   (* 4 3))))`
	`17`	`(cont-remember 3)`	`8`

`(cons 1`
` (cons 2`
`  (cons 3`
`   (call/cc`
`    (lambda (e)`
`     (set! cont-remember e)`
`     '()))))))`
	`(1 2 3)`	`(cont-remember '(7 8))`	`(1 2 3 7 8)`

`(define x 5)`
`(if (= 0 x)`
`    'undefined`
`    (remainder`
`     (* (call/cc`
`         (lambda (e)`
`          (set! cont-remember`
`e)`
`          (+ x 1) ))`
`        (- x 1))`
`     x))`
	`4`	`(cont-remember 3)`	`2`

Table 13.4 *Capturing and applying continuations. The captured continuations are stored in a global variable. The application of the continuation is shown in the third column, and the result of the application of the continuation is shown in the fourth column.*

First we explain the first row in the table. The application `(cont-remember 3)` passes 3 to the continuation e. It means that we fuel the expression `(+ 5 X)` with an `X` which is 3. The result is 8.

In the second row, `(cont-remember '(7 8))` passes the list `(7 8)` into the innermost point `Y` of `(cons 1 (cons 2 (cons 3 Y)))`. The result is the list `(1 2 3 7 8)`.

In the last row, we activate `(cont-remember 3)`. It implies that 3 is passed into the `Z` of `(remainder (* Z (- x 1)) x)`, where `x` is 5. The value is `(remainder 12 5)` = `2`. Notice in particular that the `if` has made the choice of the 'else part'. `If` is not a function, but a special form with special evaluation rules. Once it the choice of the `if` is made, there is no trace left of it in the continuation. For more details of the evaluation of `if` special forms see Chapter 19 to Chapter 21, and in particular Section 20.10.

13.8. Use of continuations for escaping purposes

Lecture 3 - slide 40

In this section we will illustrate how to apply the captured continuations for escaping purposes.

We here illustrate applications of the continuations for escaping purposes

In Table 13.5 we basically show the same expressions as in Table 13.1, Table 13.2, Table 13.3, and Table 13.4. In the light blue fragments, of the form `(e X)` we send a value `X` to the continuation, which is bound to `e`. In the first row we send 5 to the addition, and the value of the context becomes 15. In the second row we send the symbol `x` to the continuation, whereby the value of the context is the pair `(1 . x)`. (Notice that the second example captures a continuation at a more outer level than in the other tables). In the second row we send the integer *111* to the else part of the `if` form, and hereby the value of the context becomes *111*.

Context C, capturing, and escape call	**Value**
`(+ 5 (call/cc (lambda (e) (* 4 (e 10)))) )`	`15`
`(cons 1 (call/cc (lambda (e) (cons 2 (cons 3 (e 'x))))) )`	`(1 . x)`
`(define x 5)` `(if (= 0 x) 'undefined (call/cc (lambda (e) (remainder (* (+ x 1) (- x (e 111))) x))) )`	`111`

Table 13.5 *Capturing and use of a continuation for escaping purposes*

13.9. Practical example: Length of an improper list

Lecture 3 - slide 41

Now that we have seen how to capture and use continuations for escaping purposes we will study a number of *real examples*. The first is similar to the catch throw example in Program 13.1. Like the examples in Section 13.8 we also deal with escape values in this example.

Recall from Program 13.1 that we are about to program a list length function. If we, during the element counting, realize that we deal with an improper list (a list not terminated by the empty list) we want some special result, namely the symbol `improper-list`.

> The length of an improper list is undefined
>
> We chose to return the symbol `improper-list` if `list-length` encounters an improper list
>
> This example is similar to the `catch` and `throw` example shown earlier in this section

It is easy to program the escaping version of `list-length` with continuations, see Program 13.2. At the outer level we capture the continuation that immediately returns from `list-length`. We can freely name the continuation, and we chose the name `do-exit`. Within the scope of the continuation we define a local helping function `list-length1`, which does the real counting job. We follow the cdr chain in the recursion of `list-length1`. If we encounter a data object which is not a cons pair or the empty list we have identified an improper list. In this situation we send the symbol `improper-list` to `do-exit`. The effect is that we immediately return this symbol, and the count of of cons pairs is not used.

```
(define (list-length l)
  (call-with-current-continuation
   (lambda (do-exit)
     (letrec ((list-length1
                (lambda (l)
                   (cond ((null? l) 0)
                         ((pair? l) (+ 1 (list-length1 (cdr l))))
                         (else (do-exit 'improper-list))))))
       (list-length1 l)))  ))
```

Program 13.2 *The function list-length, which returns the symbol 'improper-list in case it encounters an improper list.*

13.10. Practical example: Searching a binary tree

Lecture 3 - slide 42

The next example is about traversal of a tree, with the purpose of finding a subtree which satisfy a given predicate.

> Searching a binary tree involves a recursively defined tree traversal
>
> If we find the node we are looking for it is convenient to throw the out of the tree traversal

The function `find-in-tree` is shown in Program 13.3. As in the list length example in Program 13.2 we set up a continuation at the outer level of `find-in-tree`. The continuation is called `found`. This is not a continuation used for an exceptional value, but for the expected 'normal' value of the function.

The local function `find-in-tree1` is a recursive pre-order tree traversal function. In case the predicate holds on a subtree, it is passed to the continuation `found`. If not, the subtrees are searched recursively. The recursion stops when we reach the leaves, on which `(subtree-list ...)` returns the empty list. In case we finish the traversal without ever finding a subtree that satisfies `pred` we drop through the if form. In that case we will have to return `#f`. Notice that this is a rare example of having two expression in sequence in the body of a functional abstraction.

```
(define (find-in-tree tree pred)
 (call-with-current-continuation
  (lambda (found)
   (letrec
    ((find-in-tree1
       (lambda (tree pred)
            (if (pred tree)
                (found tree)
                (let ((subtrees (subtree-list tree)))
                   (for-each
                      (lambda (subtree) (find-in-tree1 subtree pred))
                      subtrees)))
            #f)))
    (find-in-tree1 tree pred)))  ))
```

Program 13.3 *A tree search function which uses a continuation found if we find what we search for. Notice that this examples requires the function subtree-list, in order to work. The function returns #f in case we do not find node we are looking for. Notice that it makes sense in this example to have both the if expression and the #f value in sequence!*

13.11. References

[cltl-non-local-exists]	Dynamic Non-local exists (Common Lisp) http://www.ida.liu.se/imported/cltl/clm/node96.html
[cltl]	Common Lisp the Language, 2nd Edition. http://www.ida.liu.se/imported/cltl/cltl2.html
[springer89]	George Springer and Daniel P. Friedman, *Scheme and the art of programming*. The MIT Press and McGraw-Hill Book Company, 1989.