Functional Programming in Scheme

With Web Programming Examples

September 2003

Kurt Nørmark ©
Department of Computer Science, Aalborg University, Denmark

Abstract

This is a teaching material about functional programming in Scheme. The current version of the material is a 2nd edition. Compared the first edition from the Fall 2002, the title has been slightly changed. Scheme is a pragmatic choice of programming language in the functional programming paradigm, with unique flexibility due to the membership of the Lisp family of languages. Furthermore, Scheme is both very powerful and rather small compared to other Lisp languages.

As the title indicates, we are concerned with the functional subset of Scheme. As a unique aspect of this material, we illustrate the functional paradigm in Scheme with examples from the area of web programming and web authoring. This is done on the ground of LAML - the Lisp Abstracted Markup Language.

We hypothesize that an approach with HTML and XML examples is more motivating than many other approaches, which often tend to illustrate Scheme programming via examples of only little practical relevance.

The sound track of the cs.auc.dk version of this material depends on Real Player (www.real.com). The animations depend on SVG and a SVG plugin, such as the one from Adobe (http://www.adobe.com/svg/viewer/install/).

A number of exercises are integrated with this material. One star exercises are relatively easy, two star exercises of medium difficulty, and the three star exercises are the most demanding. The numbering of exercises follows the numbering of lectures. Figures, programs, tables etc. are numbered within the thematic section, where they appear.

The material is indexed, and the special index icon on most pages leads to the alphabetic index of the material.

When new concepts are introduced we often refer to The free Online Dictionary of Computing - FOLDOC. You must have an Internet connection to make use of the FOLDOC references.

In relation to Scheme, we provide references to the *Revised(5) Report on the Algorithmic Language Scheme* - R5RS . The Scheme Report is bundled with the CD version of this material.

Colophon: This material has been authored using the LENO language. LENO is an XML language, which at the grammatical level is defined using an XML DTD. We use LENO in the context of LAML, which allows us to author LENO material in the Scheme programming language. Thus, all source material is written in Scheme, using the mirrors of the XML LENO language, and a mirror of HTML. The primary LENO language is used to produced annotated slide pages, in a number of different views. A secondary source - the thematic view - can be derived, which is enriched with additional text. The present material is a particular 'printable' rendering of the thematic view. The printable thematic view has been edited slightly in MS Word for the sake of appropriate page breaking, and appropriate aggregations of theme chapters have been formed. From MS Word, PDF files have been generated using Adobe Acrobat.

Contents

1. Programming Paradigms

Before we start on the functional programming paradigm we give a broad introduction to programming paradigms in general.

In this section we will discuss the meaning of the word 'paradigm', and we will enumerate the main programming paradigms, as we see them.

In Chapter 2 we will discuss each of the main programming paradigms in some details. Be aware, however, that this material is about the functional programming paradigm. The two first chapters of the material mainly serve as background, and as contrast for an enhanced understanding of the functional school.

1.1. Paradigm

Lecture 1 - slide 2

It is interesting and useful to investigate the meaning of the word 'paradigm'

We will here look at the meaning of the word 'paradigm', as it appears in 'The American Heritage Dictionary of the English Language, Third Edition':

> *"An example that serves as pattern or model."*

Another and slightly more complicated explanation stems from the 'The Merriam-Webster's Collegiate dictionary':

> *"A philosophical and theoretical framework of a scientific school or discipline within which theories, laws, and generalizations and the experiments performed in support of them are formulated"*

Below we will first describe the meaning of the word 'paradigm' as we adopt it in this course. After that we describe related concepts, namely 'programming technique', 'programming style', and 'programming culture'.

- Programming paradigm (in this course)
  - A pattern that serves as a *school of thoughts* for programming of computers
- Programming technique
  - Related to an algorithmic idea for solving a particular class of problems
  - Examples: 'Divide and conquer' and 'program development by stepwise refinement'
- Programming style

- The way we express ourselves in a computer program
- Related to elegance or lack of elegance

- Programming culture
  - The totality of programming behavior, which often is tightly related to a family of programming languages
  - The sum of a main paradigm, programming styles, and certain programming techniques.

1.2. The main programming paradigms

Lecture 1 - slide 3

In this section we will enumerate the four main programming paradigms which will be treated in additional details in Chapter 2. It may very well be a matter of taste if some of the additional programming paradigms, which we also mention below, should be considered as main programming paradigms as well.

We identify four main programming paradigms and a number of minor programming paradigms

In the concept definition below, we characterize a main programming paradigm in terms of an *idea* and a *basic discipline*.

A *main programming paradigm* stems an *idea* within some *basic discipline* which is relevant for performing computations

- Main programming paradigms
  - The imperative paradigm
  - The functional paradigm
  - The logical paradigm
  - The object-oriented paradigm
- Other possible programming paradigms
  - The visual paradigm
  - One of the parallel paradigms
  - The constraint based paradigm

In Chapter 2 we will characterize the four main programming paradigms mentioned above. We will in particular attempt to trace the idea and basic discipline behind the four main programming paradigms. We do not go into the other programming paradigms, mentioned above, in this material.

1.3. References

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

2. Overview of the four main programming paradigms

In this section we will characterize the four main programming paradigms, as identified in Section 1.2.

As the main contribution of this exposition, we attempt to trace the *basic discipline* and the *idea* behind each of the main programming paradigms.

With this introduction to the material, we will also be able to see how the functional programming paradigm corresponds to the other main programming paradigms.

2.1. Overview of the imperative paradigm

Lecture 1 - slide 5

The word 'imperative' can be used both as an adjective and as a noun. As an adjective it means 'expressing a command or plea'. In other words, asking for something to be done. As a noun, an imperative is a command or an order. Some programming languages, such as the object oriented language Beta, uses the word 'imperative' for commands in the language.

First **do this** and next **do that**

The 'first do this, next do that' is a short phrase which really in a nutshell describes the spirit of the imperative paradigm. The basic idea is the command, which has a measurable effect on the program state. The phrase also reflects that the order to the commands is important. 'First do that, then do this' would be different from 'first do this, then do that'.

In the itemized list below we describe the main properties of the imperative paradigm.

- Characteristics:
  - Discipline and idea
    - Digital hardware technology and the ideas of Von Neumann
  - Incremental *change of the program state* as a function of *time*.
  - Execution of computational steps in an order governed by *control structures*
    - We call the steps for *commands*
  - Straightforward abstractions of the way a traditional Von Neumann computer works
  - Similar to descriptions of everyday routines, such as food recipes and car repair
  - Typical commands offered by imperative languages
    - Assignment, IO, procedure calls
  - Language representatives
    - Fortran, Algol, Pascal, Basic, C

- The natural abstraction is the procedure
  - Abstracts one or more actions to a procedure, which can be called as a single command.
  - "Procedural programming"

We use several names for the computational steps in an imperative language. The word *statement* is often used with the special computer science meaning 'a elementary instruction in a source language'. The word *instruction* is another possibility; We prefer to devote this word the computational steps performed at the machine level. We will use the word 'command' for the imperatives in a high level imperative programming language.

A procedure abstracts one or more actions to a procedure, which can be activated as a single action.

2.2. Overview of the functional paradigm

Lecture 1 - slide 6

We here introduce the functional paradigm at the same level as imperative programming was introduced in Section 2.1.

Functional programming is in many respects a simpler and more clean programming paradigm than the imperative one. The reason is that the paradigm originates from a purely mathematical discipline: the theory of functions. As described in Section 2.1, the imperative paradigm is rooted in the key technological ideas of the digital computer, which are more complicated, and less 'clean' than mathematical function theory.

Below we characterize the most important, overall properties of the functional programming paradigm. Needless to say, we will come back to most of them in the remaining chapters of this material.

Evaluate an expression and use the resulting value for something

- Characteristics:
  - Discipline and idea
    - Mathematics and the theory of functions
  - The values produced are *non-mutable*
    - Impossible to change any constituent of a composite value
    - As a remedy, it is possible to make a revised copy of composite value
  - Atemporal

- Abstracts a single expression to a function which can be evaluated as an expression
- Functions are first class values
  - Functions are full-fledged data just like numbers, lists, ...
- Fits well with computations driven by needs
  - Opens a new world of possibilities

2.3. Overview of the logic paradigm

Lecture 1 - slide 7

The logic paradigm is dramatically different from the other three main programming paradigms. The logic paradigm fits extremely well when applied in problem domains that deal with the extraction of knowledge from basic facts and relations. The logical paradigm seems less natural in the more general areas of computation.

Answer a question via search for a solution

Below we briefly characterize the main properties of the logic programming paradigm.

- Characteristics:
  - Discipline and idea
    - Automatic proofs within artificial intelligence
  - Based on axioms, inference rules, and queries.
  - Program execution becomes a systematic search in a set of facts, making use of a set of inference rules

2.4. Overview of the object-oriented paradigm

Lecture 1 - slide 8

The object-oriented paradigm has gained great popularity in the recent decade. The primary and most direct reason is undoubtedly the strong support of encapsulation and the logical grouping of program aspects. These properties are very important when programs become larger and larger.

The underlying, and somewhat deeper reason to the success of the object-oriented paradigm is probably the conceptual anchoring of the paradigm. An object-oriented program is constructed with the outset in concepts, which are important in the problem domain of interest. In that way, all the necessary technicalities of programming come in second row.

Send messages between objects to simulate the temporal evolution of a set of real world phenomena

As for the other main programming paradigms, we will now describe the most important properties of object-oriented programming, seen as a school of thought in the area of computer programming.

- Characteristics:
  - Discipline and idea
    - The theory of concepts, and models of human interaction with real world phenomena
  - Data as well as operations are encapsulated in objects
  - Information hiding is used to protect internal properties of an object
  - Objects interact by means of message passing
    - A metaphor for applying an operation on an object
  - In most object-oriented languages objects are grouped in classes
    - Objects in classes are similar enough to allow programming of the classes, as opposed to programming of the individual objects
    - Classes represent concepts whereas objects represent phenomena
  - Classes are organized in inheritance hierarchies
    - Provides for class extension or specialization

This ends the overview of the four main programming paradigms. From now on the main focus will be functional programming in Scheme, with special emphasis on examples drawn from the domain of web program development.

2.5. References

[-] Foldoc: object-oriented programming
http://wombat.doc.ic.ac.uk/foldoc/foldoc.cgi?query=object-oriented+programming

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

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

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

3. Lisp and Scheme

We use the programming language Scheme in this material. Therefore it is natural to start with a brief discussion of the family of languages, to which Scheme belongs. This is the Lisp family of languages.

3.1. Lisp

Lecture 2 - slide 2

Lisp was invented by John McCarthy in the late fifties. In these days the dominating use of computers was for numeric purposes. One of the purposes of Lisp was to support symbolic computation.

As an example of symbolic computation, let us mention the calculation of differentiated mathematical functions. The symbolic derivation of the function $f(x) = x * x$ is the function $g(x) = 2 * x$. The numeric derivation of f will never deliver the function g on source form. The best we can hope for is some sort of numeric approximation to g, which can be applied to numbers.

It is worth noticing that transformation and compilation of programs also can be considered as symbolic computations. In fact it turns out, that the computer is better suited to do symbolic computations than numeric computations, because the former always can be done exactly, whereas the latter often are inexact.

Today, many Lisp languages are not in use any more. Lisp 1.5 and Interlisp are two of these. 'Lisp' is today used as a family name of all 'Lisp languages', which includes such languages and Emacs Lisp, Common Lisp, and Scheme.

Lisp is the next oldest programming language - only Fortran is older.

In the past, thousands of programming languages have been invented and tried out. Naturally, many of these are not in active use any more. It is interesting to notice that both Lisp and Fortran are still in widespread use for many different purposes.

Below we will summarize the main characteristics of Lisp.

- Lisp characteristics:
  - Invented for symbolic computations
  - Superficially inspired by mathematical function theory
  - Is syntactically and uniformly based on parenthesized prefix notation
    - Parsing a Lisp program is trivial
  - Programming goes hand in hand with language development
  - It is easy to access and manipulate programs from programs
    - Calls for tool making in Lisp

One of the most remarkable facts of Lisp is that the primary data structure in the language - lists - is used for the representation of programs. This is the reason why we use all these parentheses in a Lisp program! Originally, this characteristic program representation was only thought as an intermediate representation, not to be used by the human programmer. It turned out, eventually, that the representation had some very useful properties. Therefore the following 'equation' is an important characteristic of all Lisp languages.

Program = Data = Lists

3.2. Scheme

Lecture 2 - slide 3

Scheme is a programming language in the Lisp family. Scheme is formally defined in the Scheme report [Abelson98], which is revised from time to time. Currently, the fifth revision is the most current one. This explains the abbreviation R5RS, which goes something like 'The fifth Revised Report on the Algorithmic Language Scheme'.

Scheme is a small, yet powerful language in the Lisp family

- Scheme characteristics:
  - Supports functional programming - but not on an exclusive basis
  - Functions are first class data objects
  - Uses static binding of free names in procedures and functions
  - Types are checked and handled at run time - no static type checking
  - Parameters are evaluated before being passed - no lazyness

Many people encounter Lisp programming in Emacs Lisp [fsf02] , [fsf02a], because of the need of customizing Emacs in non-trivial ways. Emacs Lisp is an old and primitive dialect of Lisp. Hard core Lisp programmers are also likely to meet Common Lisp, which is much bigger than Scheme. The statement below compares very briefly Common Lisp and Emacs Lisp with Scheme.

Scheme is an attractive alternative to **Common Lisp** (a big monster) and **Emacs Lisp** (the rather primitive extension language of the Emacs text editor).

Exercise 2.1. *Getting started with Scheme and LAML*

The purpose of this exercises is learn the most important practical details of using a Scheme system on Unix. In case you insist to use Windows we will assume that you install the necessary software in your spare time. There is no time available to do that during the course exercises. Further details on installation of Scheme and LAML on Windows.

You will have to choose between DrScheme and MzScheme.

DrScheme is a user friendly environment for creating and running Scheme programs, with lots of menus and lots of help. However, it is somewhat awkward to use DrScheme with LAML. Only use DrScheme in this course if you cannot use Emacs, or if you are afraid of textually, command based tools. Follow this link for further details.

MzScheme is the underlying engine of DrScheme. MzScheme is a simple read-eval-print loop, which let you enter an expression, evaluate and print the result. MzScheme is not very good for debugging and error tracing. MzScheme works well together with Emacs, and there is a nice connection between MzScheme and LAML. MzScheme used with Emacs is preferred on this course. Please go through the following steps:

1. Insert the following line in your .emacs file in your home dir, and then restart Emacs:

```
(load "/pack/laml/emacs-support/dot-emacs-contribution.el")
```

2. Have a session with a naked Scheme system by issuing the following command in Emacs: `M-x run-scheme-interactively`
   - Define a couple of simple functions (`odd` and `even`, for instance) and call them.
   - Split the window in two parts with `C-x 2` and make a buffer in the topmost one named `sources.scm` (`C-x b`). Bring the Scheme interpreter started above into the lower part of the window. The buffer with the Scheme process is called `*inferior-lisp*`. Put the `sources.scm` buffer in Scheme mode (`M-x scheme-mode`). Define the functions `odd` and `even` in the buffer and use the Scheme menu (or the keyboard shortcuts) to define them in the running Scheme process.
3. Have a similar session with a Scheme+LAML system by issuing the following command in Emacs: `M-x run-laml-interactively` (You may have to confirm that a previously started Scheme process is allowed to be killed).
   - All you did in item 2 can also be done here.
   - Evaluate a simple HTML expression, such as

```
(html (head (title "A title")) (body (p "A body")))
```

   - Use the function `xml-render` to make a textual rendering of the HTML expression.
   - Make a deliberate grammatical error in the LAML expression and find out what happens.
4. Make a file 'try.laml'.
   - Control that Emacs brings the buffer in Laml mode. Issue a `M-x laml-mode`

explicitly, if necessary.

- Use the menu 'Laml > Insert LAML template' to insert an XHTML template.
- Fill in some details in the head and body.
- Process the file via the LAML menu in Emacs: Process asynchronously. The file try.html will be defined.
- Play with simple changes to the HTML expression, and re-process. You can just hit C-o on the keyboard for processing.
- You can get good inspiration from the tutorial Getting started with LAML at this point.

3.3. References

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

[-] The Scheme Language Report
http://www.cs.auc.dk/~normark/prog3-03/external-material/r5rs/r5rs-html/r5rs_1.html

[-] Schemers.org home page
http://www.schemers.org/

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

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

[abelson98] Richard Kelsey, William Clinger and Jonathan Rees, "Revised^5 Report on the Algorithmic Language Scheme", *Higher-Order and Symbolic Computation*, Vol. 11, No. 1, August 1998, pp. 7--105.

[fsf02] Free Software Foundation, "Programming in Emacs Lisp (Second Edition)", January 2002.

[fsf02a] *GNU Emacs Lisp Reference Manual.* The Free Software Fundation Inc, May 2002.

4. Expressions and values

The notion of *expression* is of central importance in the functional programming paradigm. In some sense, expressions is the only computational building block of the functional programming paradigm. As a contrast, the imperative paradigm makes use of both commands and expressions. In the imperative paradigm commands are executed with the purpose of modifying the state of the program, as it is being executed. Expressions are executed - or evaluated - with the purpose of producing a value. Values of expressions can be used as parts of surrounding expressions, in an evaluation process. Ultimately, the value of an expression is presented to the person who runs a functional program. The value serves as the 'result' of the computation.

4.1. Expressions, values, and types

Lecture 2 - slide 5

We will now describe and characterize the important concepts of expressions, values and types.

The relationship between the three key concepts is as stated below.

Evaluation of an *expression* yields a *value* which belongs to a *type*

In the itemized list below we will describe the most important properties of expressions, values, and types. The coverage given here is only a brief appetizer. We have much more to say about all three of them later in the material.

- Expressions
  - Written by the programmer
  - Will typically involve one or more function calls
  - A Function - as written by the programmer - is itself an expression
- Values
  - Primitive as well as composite
  - A function expression is evaluated to a function object
- Types
  - A set of values with common properties
  - Type checking can be used to validate a program for certain errors before it is executed

Expressions are part of the source program, as written by the programmer.

A function expression is called a lambda expression. We will encounter these important expressions later in this material.

The primitive values are those which cannot be decomposed into more primitive parts. Some of the important primitive values are numbers, the two boolean values (*true* and *false*), and the characters of some character set. Primitive values stand as a contrast to composite values, such as lists and arrays, which are aggregations of parts, each of which are compositive or primitive values themselves.

4.2. Examples of expressions and their values

Lecture 2 - slide 6

Before we go into additional details, we will give examples of important kinds of expressions. This includes simple expressions, such as the well-known arithmetic expressions. Next we give an example of a conditional expression, which somehow corresponds to selective control structures in the imperative paradigm. Lambda expressions generate functions, and as such they are of primary importance for the functional programming paradigm. Finally, HTML expressions are of interest to the approach taken in the running example used in this material - an example from the domain of web program development. Wherever possible we wish to illustrate the use of functional programming in the web domain. In this domain, expressions that involve mirrors of HTML and XML elements are the key constituents.

- *Let us assume that x has the value 3*
- **Simple expressions**
  - `7` has the value 7
  - `(+ x 5)` has the value 8
- **Conditional expressions**
  - `(if (even? x) 7 (+ x 5))` has the value 8
- **Lambda expressions**
  - `(lambda (x) (+ x 1))` has the value 'the function that adds one to its parameter'
- **HTML mirror expressions**

```
(html
 (head
  (title "PP"))
 (body
  (p "A body paragraph"))
)
```

  - The value of this expression can be rendered as a string in HTML which can be presented in an Internet browser.

The conditional expression is evaluated in two steps. First the boolean expression `(even? x)` is evaluated. If `x` is even, the boolean expression `(even? x)` evaluates to true and the trivial expression 7 is evaluated. Because `x` is 3 and therefore odd, the other expression `(+ x 5)` is evaluated, giving us the final value 8. It is important to realize that an if form does not evaluate all three constituent expressions at the outset. It first evaluates the boolean expression, and based on

the outcome, it either evaluates the 'then part' or the 'else part'. Not both! We have much more to say about the order of evaluation of an `if` form in a later part of this material

Regarding the lambda expression, the `x` in parentheses after lambda is the formal parameter of the function. the expression `(+ x 1)` is the body. In functions, the body is an expression - not a command.

The HTML mirror expressions stem from the LAML libraries.

The functions `html`, `body`, `title`, `head`, and `p` correspond to the HTML elements of the same names. In the LAML software, the HTML elements are mirrored as functions in Scheme.

The evaluation order of the constituents in a conditional expression is discussed in details in Section 20.10. Conditional expression is a theme we will study in much more detail in Chapter 10. HTML mirror expressions are discussed in additional details in Chapter 26.

4.3. Evaluation of parenthesized expressions

Lecture 2 - slide 7

Parentheses in Scheme are used to denote lists. Program pieces - expressions - are represented as lists. Evaluation of parenthesized expressions in Scheme follows some simple rules, which we discuss below.

How is the form `(a b c d e)` evaluated in Scheme?

The form `(a b c d e)` appears as a pair of parentheses with a number of entities inside. The question is how the parenthesized expression is evaluated, and which constraints apply to the evaluation.

- Evaluation rules
  - The evaluation of the empty pair of parentheses `( )` is in principle an error
  - If `a` is the name of a special form, such as `lambda`, `if`, `cond`, or `define` special rules apply
  - In all other cases:
    - Evaluate all subforms uniformly and recursively.
    - The value of the first constituent `a` ***must*** be a function object. The function object is called with the values of `b`, `c`, `d`, and `e` as actual parameters.

The evaluation of the empty pair of parentheses `( )` is often - in concrete Scheme systems - considered as the same as `'( )`, which returns the empty list. However, you should always quote the empty pair of parentheses to denote the empty list.

Having reached the case where a function is called on some given data, which are passed as parameters, like in the call (a b c d), the next natural question is how to call a function. We will explore this in detail in Chapter 20, more specifically, Section 20.3where we see that the call should be replaced by the body of the called function, in which the formal parameters should be replaced by the actual parameters.

4.4. Arithmetic expressions

Lecture 2 - slide 8

It is natural to start our more detailed study of expressions by reviewing the well-known arithmetic expressions. The important thing to notice is the use of fully parenthesized prefix notation.

Scheme uses fully parenthesized arithmetic expressions with prefix notation

Prefix notation is defined in the following way:

Using *prefix notation* the operator is given before the operands

Prefix notation stands as a contrast to infix and postfix notation. Infix notation is 'standard notation' in which the operand is found in between the operands.

Below we give examples of evaluation of a number of different arithmetic expressions. Notice in particular the support of rational numbers in Scheme, and the possibility to use arbitrarily large numbers.

Expression	Value
`(+ 4 (* 5 6))`	`34`
`(define x 6)` `(+ (* 5 x x) (* 4 x) 3)`	`207`
`(/ 21 5)`	`21/5`
`(/ 21.0 5)`	`4.2`

`(define (fak n)`
`  (if (= n 0) 1 (* n (fak (- n 1)))))`
 	
`(fak 50)`
	`30414093201713378043612608166064768844377641568960512000000000000`

Table 4.1 *Examples of arithmetic expressions. The prefix notation can be seen to the left, and the values of the expressions appear to the right.*

There is no need for priorities - operator precedence rules - of operators in fully parenthesized expressions

The observation about priorities of operators stands as a contrast to most other languages. In Lisp and Scheme, the use of parentheses makes the programmer's structural intentions explicit. There is no need for special rules for solving the parsing problem of arithmetic expressions. Thus, 1+2*3 is written (+ 1 (* 2 3)), and it is therefore clear that we want to multiply before the addition is carried out.

4.5. Equality in Scheme

Lecture 2 - slide 9

Equality is relevant and important in most programming paradigms and in most programming languages. Interestingly, equality distinctions are not that central in the functional programming paradigm. Two objects o1 and o2 which are equal in a weak sense (structurally equal) cannot really be distinguished in the functional paradigm. One of them can be substituted by the other without causing any difference or harm.

When we encounter imperative aspects of the Scheme language, the different notions of equality becomes more important. In the imperative programming paradigm we may mutate existing objects. By changing one of the two structurally equal objects, `o1` and `o2`, it is revealed if `o1` and `o2` are also equal in a stronger sense. If the mutation of `o1` also affects `o2` we can conclude that `o1` and `o2` are identical `(eq? o1 o2)`. If a mutation of `o1` does not affect `o2`, then `(not (eq? o1 o2))`.

As most other programming languages, Scheme supports a number of different equivalence predicates

In Scheme we have the following important equivalence predicates:

- The most discriminating
  - `eq?`
- The least discriminating - structural equivalence
  - `equal?`
- Exact numerical equality
  - `=`
- Others
  - `eqv?` is close to `eq?`
  - `string=?` is structural equivalence on strings

An equivalence predicate divides a number of objects into equivalence classes (disjoint subsets). The objects in a certain class are all equal. The most discriminating equivalence predicate is the one forming most equivalence classes.

To stay concrete, we show a number some examples of equality expressions in a dialogue with a Scheme system. You should consider to try out other examples yourself.

```
1> (eq? (list 'a 'b) (list 'a 'b))
#f

2> (eqv? (list 'a 'b) (list 'a 'b))
#f

3> (equal? (list 'a 'b) (list 'a 'b))
#t

4> (= (list 'a 'b) (list 'a 'b))
=: expects type <number> as 2nd argument, given: (a b); other arguments were: (a
b)

5> (string=? "abe" "abe")
#t

6> (equal? "abe" "abe")
#t

7> (eq? "abe" "abe")
#f

8> (eqv? "abe" "abe")
#f
```

Program 4.1 *A sample interaction using and demonstrating the equivalence functions in Scheme.*

4.6. The read-eval-print loop

Lecture 2 - slide 10

It is possible to interact directly with the Scheme interpreter. At a fine grained level, expressions are read and evaluated, and the resulting value is printed. This is a contrast to many other language processors, which require much more composite and coarse grained fragments for processing purposes.

The tool which let us interact with the Scheme interpreter is called a 'read-eval-print loop', sometimes referred to via the abbreviation 'REPL'.

> The 'read-eval-print loop' allows you to interact with a Scheme system in terms of evaluation of individual expressions

We show the interaction with a Scheme REPL below. The interaction is quite similar to the exposition in Table 4.1. In this as well as in future presentations of REPL interaction, we often put a number in front of the prompt. This simple convenience to allow us to refer to a single interaction in our discussions.

```
1> (+ 4 (* 5 6))
34

2> (define x 6)

3> (+ (* 5 x x) (* 4 x) 3)
207

4> (/ 21 5)
21/5

5> (/ 21.0 5)
4.2

6> (define (fak n) (if (= n 0) 1 (* n (fak (- n 1)))))

7> (fak 50)
30414093201713378043612608166064768844377641568960512000000000000
```

Program 4.2 *A sample session with Scheme using a read eval print loop.*

4.7. References

[-] Equivalence predicates in R5RS
http://www.cs.auc.dk/~normark/prog3-03/external-material/r5rs/r5rs-html/r5rs_48.html#SEC50

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

[-] R5RS: Numerical operations
http://www.cs.auc.dk/~normark/prog3-03/external-material/r5rs/r5rs-html/r5rs_54.html

[-] R5RS: Numbers in Scheme
http://www.cs.auc.dk/~normark/prog3-03/external-material/r5rs/r5rs-html/r5rs_49.html#SEC51

[-] R5RS: Procedure calls
http://www.cs.auc.dk/~normark/prog3-03/external-material/r5rs/r5rs-html/r5rs_28.html

5. Types

It is hard to justify a text about functional programming without a fair treatment of types. In this chapter we will go over the most important concepts of types, as relevant in the functional programming paradigm.

5.1. Types

Lecture 2 - slide 12

Types plays an essential role in any programming language and any programming paradigm. In many languages types are used in the program text as constraints on variables and parameters. C, Pascal, and Java are such languages. In others, the types are inferred (somehow extracted from use of variables, parameters etc. relative to the ways the variables and parameters are used in operators and functions). ML is such a language. Yet in other languages, types are solely used to classify the values (data objects) at run time. Scheme is such a language. Thus, in Scheme we do not encounter types in the source program, but only at run time.

In general, we see three advantages of dealing with types:

> The notion of type is used to make programs *more readable*, make them run *more efficient*, and to *detect certain errors* before they cause errors in the calculation

Let us now make a more detailed characteristics of these advantages in the following itemized list.

- Readability
  - Explicitly typed variables, parameters and function serve as important *documentation*, which enhances the program understanding.
- Efficiency
  - Knowledge of the properties of data makes it possible to generate more efficient code
- Correctness
  - Explicit information about types in a program is a kind of *redundancy* against which it is possible to check expressions and values
  - Programmers usually wish to identify type errors as early as possible in the development process

The correctness quality is often brought into focus. In the next few sections we will therefore discuss type checking issues.

5.2. Type checking

Lecture 2 - slide 13

As already mentioned the use of types in source programs makes it possible to deal with program correctness - at least in some simple sense. In this context, correctness is not relative to the overall intention or specification of the program. Rather, it is in relation to the legal use of values as input to operators and functions.

> Type checking is the processes of identifying errors in a program based on explicitly or implicitly stated type information

Below we will identify three kinds of 'typing', which are related to three different approaches to type checking.

- **Weak typing**
  - Type errors can lead to erroneous calculations
- **Strong typing**
  - Type errors cannot cause erroneous calculations
  - The type check is done at compile time or run time
- **Static typing**
  - The types of all expressions are determined before the program is executed
  - The type check is typically carried out in an early phase of the compilation
  - Comes in two flavors: explicit type decoration and implicit type inference
  - Static typing implies strong typing

According to section 1.1 the Scheme Report (R5RS) 'Scheme has latent as opposed to manifest types. Types are associated with values (also called objects) rather than with variables.' In our categorization, Scheme is strongly typed and types are dealt with at run time (on values) as a contrast to compile time (on variables).

It is worth noticing that we classify Scheme as supporting strong typing. Many programmers will probably be surprised by this categorization, because the 'typing' in Scheme is experienced to be relatively 'weak' and 'dynamic'. However, type errors in Scheme do not cause erroneous calculations. Type errors are discovered at a low and basic level. As such we find it justifiable to classify the typing in Scheme as being strong. Notice however, that there is no trace of static type checking in Scheme. Static type checking is the rule in most modern programming languages today, and static type checking is also an absolutely key aspect in functional programming languages such as ML [Harper88] and Haskell [Hudak92].

5.3. Static type checking

Lecture 2 - slide 14

We here make the distinction between explicit type decoration and implicit type inference, and explain the principled difference.

There are two main kinds of static type checking: explicit type decoration and implicit type inference

For the sake of the discussion we will involve the following example:

Let us study the expression `(+ x (string-length y))`

- Explicit type decoration
  - Variables, parameters, and others are explicitly declared of a given type in the source program
  - It is checked that `y` is a string and that `x` is a number
- Implicit type inference
  - Variables and parameters are not decorated with type information
  - By studying the body of `string-length` it is concluded that `y` must be a string and that the type of `(string-length y)` has to be an integer
  - Because `+` adds numbers, it is concluded that `x` must be a number

Explicit type decoration is well-known to most computer science students.

If you want to study additional details about implicit type inference you should consult a textbook of ML or Haskell programming.

5.4. An example of type checking

Lecture 2 - slide 15

We will now discuss type checking relative to the three kinds of 'typing', which we identified in Section 5.2.

Is the expression `(+ 1 (if (even? x) 5 "five"))` correct with respect to types?

The example shows an arithmetic expression that will cause a type error with most type checkers. However, if x is even the sum can be evaluated to 6. If x is odd, we encounter a type error because we cannot add the integer 1 to the string "five".

- Weak typing
  - It is not realized that the expression `(+ 1 "five")` is illegal.
  - We can imagine that it returns the erroneous value 47
- Strong typing
  - If, for instance, `x` is 2, the expression `(+ 1 (if (even? x) 5 "five"))` is OK, and has the value 6
  - If `x` is odd, it is necessary to identify this as a problem which must be reported before an evaluation of the expression is attempted
- Static typing
  - `(+ 1 (if (even x) 5 "five"))` fails to pass the type check, because the type of the expression cannot be statically determined
  - Static type checking is rather *conservative*

When we use the word *conservative* for static type checking, we refer to the caution of the type checker. Independent of branching, and in 'worst cases scenarios', the type constraints should be guaranteed to hold.

5.5. Types in functional programming languages

Lecture 2 - slide 16

Before we proceed we will compare the handling of types in Scheme with the handling of types in other functional programming languages. Specifically, we compare with Haskell and ML.

Scheme is not representative for the handling of types in most contemporary functional programming languages

- ML and Haskell
  - Uses static typing ala implicit type inference
  - Some meaningful programs cannot make their way through the type checker
  - There will be no type related surprises at run time
- Scheme
  - Is strongly typed with late reporting of errors
  - Type errors in branches of the program, which are never executed, do not prevent program execution
  - There may be corners of the program which eventually causes type problems

Due to the handling of types, Scheme and Lisp are elastic and flexible compared with ML, Haskell, and other similar language which are quite stiff and rigid.

5.6. References

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

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

[-] R5RS: Semantics (Types in Scheme)
http://www.cs.auc.dk/~normark/prog3-03/external-material/r5rs/r5rs-html/r5rs_5.html

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

[harper88] Robert Harper, Robin Milner and Mads Tofte, "The Definition of Standard ML, Version 2", No. ECS-LFCS-88-62, University of Edinburgh, August 1988, .

[hudak92] Paul Hudak and Joseph H. Fasel, "A Gentle Introduction to Haskell", *ACM Sigplan Notices*, Vol. 27, No. 5, May 1992.

6. Lists

The list data type is the characteristic composite data type in all Lisp languages, and as such also in Scheme. Interesting enough, the surface form of a Lisp program is a list itself. This is an important practical observation. Below, we will study the list data type of Lisp and Scheme.

6.1. Proper lists

Lecture 2 - slide 18

Lists are recursively composed. We start with the main points regarding the recursive construction of lists.

A list is recursively composed of a *head* and a *tail*, which is a (possibly empty) list itself

The building blocks of lists are the *cons cells*

Every such cell is allocated by an activation of the `cons` function

Below we illustrate how the list `(d c b a)` is built. The web version of the material gives the best impression of the construction process, via animation (refresh the web presentation to restart the animation). The paper version of the material only shows the end result of the construction.

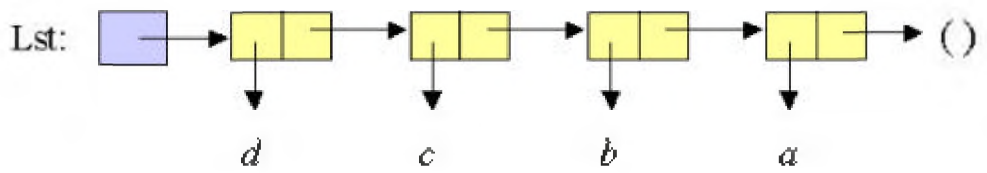

Figure 6.1 *A list (d c b a) constructed by evaluating the expression* `(cons 'd (cons 'c (cons 'b (cons 'a '()))))`.

In the items below we emphasize the decomposition of the cons cell made by `(cons e f)`, where `e` is an arbitrary expression and `f` is a list. Notice that we assume that the variable `x` is bound to `(cons e f)`.

- Construction of the list structure which we here call x
  - `(cons e f)`
- Selection:
  - `(car x) => e`
  - `(cdr x) => f`

The constructor function `cons` takes an element e and a list f and constructs a new list. As illustrated above cons makes exactly one new cons cell, and no kind of list copying is involved at all.

The selector `car` returns the first element of the list. A better name of car would be `head` .

The selector `cdr` returns the list consisting of all but the first element in the list. A better name of cdr would be `tail` .

A *proper list* is always terminated by the empty list

In Scheme the empty list is denoted '(). When we in this context talk about the termination of the list we mean the value we get by following the `cdr` references to the end of the list structure.

6.2. Symbolic expressions and improper lists

Lecture 2 - slide 19

As illustrated above, the `cons` function can be used to construct linear linked lists. It should not come as a surprise, however, that `cons` can be used to make binary tree structures as well. The reason is that each cons cell can refer to two other cons cells.

The `cons` function is suitable for definition of binary trees with 'data' in the leaves

In Figure 6.2 we show a binary tree structure made by use of the the cons function. The light blue box labeled Sexpr is a variable, the value of which is the binary tree structure. In Exercise 2.2 you are encouraged to construct the tree.

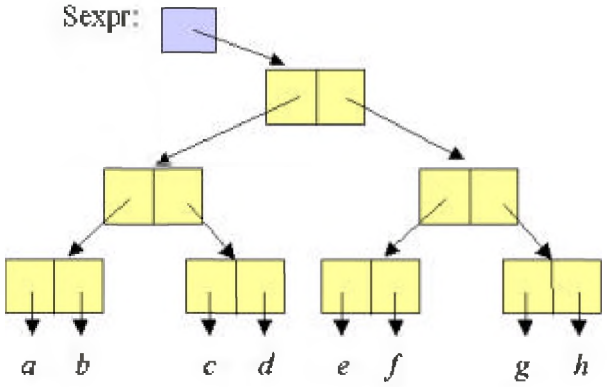

Figure 6.2 *A symbolic expression which illustrates the most general form it can take - a binary tree*

In Figure 6.3 we show the exact same structure in a slightly different layout, and with another coloring. This layout emphasizes the understanding of the structure as an improper list. The first element is the green tree, the second is the brown tree, the third is the symbol g, and the improper list element is the symbol h.

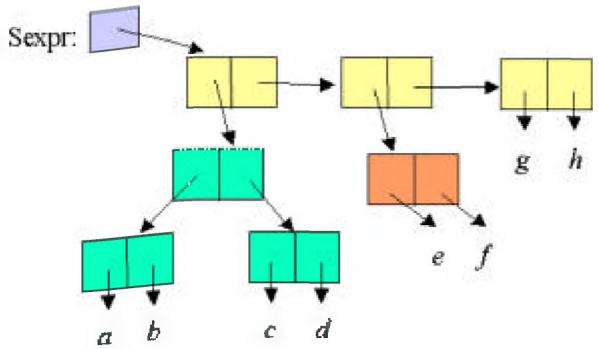

Figure 6.3 *The same symbolic expression laid out as a list. The expressions is a proper list if and only if h is the empty list. If h is not the empty list, the symbolic expression is an improper list.*

As a matter of terminology, we use the name symbolic expressions, or S-expressions, for the structures we have shown above.

Exercise 2.2. *Construction of symbolic expressions*

Construct the symbolic expressions illustrated on this page via the `cons` primitive in Scheme. The entities named *a* through *h* should be symbols. As a help, the rightmost part of the structure is made by `(cons 'g 'h)`. `'g` is equivalent to `(quote g)`, meaning that `g` is not evaluated, but taken for face value.

Experiment with *h* being the empty list. Try to use a proper list function, such as `length`, on your structures.

6.3. Practical list construction

Lecture 2 - slide 20

On this page we address topics related to 'practical list construction'. Often, `cons` is too low level for construction of lists. Instead we use the `list` function or quoted expressions. A quoted expression is taken for face value; The `quote` form `'(e)` prevents evaluation. The `quasiquote` form `` `(e) `` is a variant of `quote`, which allows non constant constituents in a `quote` form. Please notice the use of 'normal quote' and 'back quote' before the parentheses. For details of the `quasiquote` special form you should consult section 4.2.6 in the Scheme report [Abelson98] .

`cons` is the basic list constructor function - but it can be applied through a number of other means as well

- List and S-expression construction:
  - Deep `cons` expressions
  - Using the `list` function
  - Using `quote` or `quasiquote` also known as *backquote*

Expression	Value
`(cons 1 (cons 2 (cons (+ 1 2) '())))`	`(1 2 3)`
`(list 1 2 (+ 1 2))`	`(1 2 3)`
`(quote (1 2 (+ 1 2)))`	`(1 2 (+ 1 2))`
`'(1 2 (+ 1 2))`	`(1 2 (+ 1 2))`
`(quasiquote (1 2 (unquote (+ 1 2))))`	`(1 2 3)`
`` `(1 2 ,(+ 1 2)) ``	`(1 2 3)`

Table 6.1 *Examples of list construction by use of* `cons`, `list` *and quoted list expressions.*

Exercise 2.3. *Every second element of a list*

Write a function, `every-second-element`, that returns every second element of a list. As examples

```
(every-second-element '(a b c)) => (a c)
(every-second-element '(a b c d)) => (a c)
```

It is recommended that you formulate a recursive solution. Be sure to consider the basis case(s) carefully.

It is often worthwhile to go for a *more general solution* than actually needed. Sometimes, in fact, the general solution is simpler than one of the more specialized solutions. Discuss possible generalizations of `every-second-element`, and implement the one you find most appropriate.

6.4. List functions

Lecture 2 - slide 21

On this page we will review a number of important list functions, which are part of Scheme and described in section 6.3.2 of the Scheme report [Abelson98] .

There exists a number of important List functions in Scheme, and we often write other such functions ourselves

- `(null? lst)` A predicate that returns whether lst is empty
- `(list? lst)` A predicate that returns whether lst is a proper list
- `(length lst)` Returns the number of elements in the proper list lst
- `(append lst1 lst2)` Concatenates the elements of two or more lists
- `(reverse lst)` Returns the elements in lst in reverse order
- `(list-ref lst k)` Accesses element number k of the list lst
- `(list-tail lst k)` Returns the k'th tail of the list lst

It should be noticed that the first element is designated as element number 0. Thus `(list-ref '(a b c) 1)` returns `b`

6.5. Association lists

Lecture 2 - slide 22

Association lists are often used to associate two pieces of data. Association lists in Lisp and Scheme correspond to a particular implementation of associative arrays, cf. [knoopnotes]

An association list is a list of `cons` pairs

Association lists are used in the same way as associative arrays

In the table below we shows simple examples and applications of association lists. Try them out yourself!

Expression	Value
`(define computer-prefs '((peter . windows) (lars . mac) (paw . linux) (kurt . unix)))`	
`(assq 'lars computer-prefs)`	`(lars . mac)`
`(assq 'kurt computer-prefs)`	`(kurt . unix)`
`(define computer-prefs-1 (cons (cons 'lene 'windows) computer-prefs))`	
`computer-prefs-1`	`((lene . windows) (peter . windows) (lars . mac) (paw . linux) (kurt . unix))`

Table 6.2 *Examples of association lists. The function assq uses `eq?` to compare the first parameter with the first element - the key element - in the pairs. As an alternative, we could use the function `assoc`, which uses `equal?` for comparison. A better and more general solution would be to pass the comparison function as parameter. Notice in this context, that both `assq` and `assoc` are 'traditional Lisp functions' and part of Scheme, as defined in the language report.*

Exercise 2.4. *Creation of association lists*

Program a function `pair-up` that constructs an association list from a list of keys and a list of values. As an example

```
(pair-up '(a b c) (list 1 2 3))
```

should return

```
((a . 1) (b . 2) (c . 3))
```

Think of a reasonable solution in case the length of the key list is different from the length of the value list.

Exercise 2.5. *Association list and property lists*

Association lists have been introduced at this page. An association list is a list of keyword-value pairs (a list of cons cells).

Property lists are closely related to association lists. A property list is a 'flat list' of even length with alternating keys and values.

The property list corresponding to the following association list

```
((a . 1) (b . 2) (c . 3))
```

is

```
(a 1 b 2 c 3)
```

Program a function that converts an association list to a property list. Next, program the function that converts a property list to an association list.

6.6. Property lists

Lecture 2 - slide 23

Property lists are closely related to association lists. On this page - in Table 6.3 - we compare the two kinds of lists with each other. In Program 6.1 we give examples of property lists from LAML, which uses property lists for attributes in HTML, XML, and CSS.

A property list is a flat, even length list of associations

The HTML/XML/CSS attributes are represented as property lists in LAML documents

Association list	Property list
((peter . "windows") (lars . "mac") (paw . "linux") (kurt . "unix"))	(peter "windows" lars "mac" paw "linux" kurt "unix")

Table 6.3 *A comparison between association lists and property lists. In this example we associate keys (represented as symbols) to string values.*

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


(write-html 'raw
 (html 'xmlns "http://www.w3.org/1999/xhtml"
  (head
   (meta 'http-equiv "Content-Type"
         'content "text/html; charset=iso-8859-1")
   (title "Attribute Demo"))
  (body 'id "KN" 'class "generic"

    (p "Here comes a camouflaged link:")

    (p (a 'href "http://www.cs.auc.dk" 'css:text-decoration "none"
          'target "main" "Link to the CS Department"))

    (p "End of document."))))


(end-laml)
```

Program 6.1 *A simple LAML document with emphasis on the attributes, represented as property lists. There are four attribute lists (property lists, each with its own color). Notice the CSS attribute* `css:text-decoration`*, given inline in the document .*

In the LAML general library there are functions (`alist-to-propertylist` and `propertylist-to-alist`) that convert between association lists and property lists

You should consult Section 26.2 if you want to learn more about the handling of attributes in LAML.

6.7. Tables as lists of rows

Lecture 2 - slide 24

In this material we are especially interested in studying examples from the web domain. In HTML, tables are represented as collections of rows. It is therefore obvious to use lists of lists as a concrete Lisp representation of tables. In Table 6.4 we show such a table, tab1, its rendering, and a number of manipulations of the table (transpositions, row eliminations, and column eliminations). The table operations will be studied in further details in Exercise 4.4 .

It is natural to represent tables as lists of rows, and to represent a row as a list

Tables play an important roles in many web documents

LAML has a strong support of tables

Expression	Value

`tab1`	`(("This" "is" "first" "row")`
	` ("This" "is" "second" "row")`
	` ("This" "is" "third" "row")`
	` ("This" "is" "fourth" "row")`
	`)`

`(show-table tab1)`	This is first row This is second row This is third row This is fourth row
`(show-table (transpose-1 tab1))`	This This This This is is is is first second third fourth row row row row
`(show-table (eliminate-row 2 tab1))`	This is first row This is third row This is fourth row
`(show-table (eliminate-column 4 tab1))`	This is first This is second This is third This is fourth

Table 6.4 *Examples of table transposing, row elimination, and column elimination. We will program and illustrate these functions in a later exercise of this material. The function* `show-table` *is similar to* `table-0` *from a LAML convenience library. Using higher-order functions it is rather easy to program the show-table function. We will come back to this later in these notes.*

In the program below we show a possible implementation of the show-table function, which we used in Table 6.4 The function table-1 is one of the LAML convenience functions, which we have used in the past. There are others and more interesting ways to deal with tables in LAML. You should consult Program 18.6 for details.

```
(define (show-table rows)
 (let ((row-lgt (length (first rows))))
   (table-1
      0
      (make-list row-lgt 50)
      (make-list row-lgt green1)
      rows)))
```

Program 6.2 *The function show-table, implemented in terms of a LAML table function. There are several different ways to implement and deal with the table functions. In the chapter about higher-order functions we describe another simple table function.*

6.8. Programs represented as lists

Lecture 2 - slide 25

The purpose of this section is to remind you that Scheme programs are themselves list structures. At this point of the material, it should not be a big surprise to the readers.

> It is a unique property of Lisp that programs are represented as data, using the main data structure of the language: the list

A sample Scheme program from the LAML library:

```
(define (as-number x)
  (cond ((string? x) (string->number x))
        ((number? x) x)
        ((char? x) (char->integer x))
        ((boolean? x) (if x 1 0))  ; false -> 0, true -> 1
        (else
         (error
          (string-append "Cannot convert to number " (as-string x))))
```

Program 6.3 *The function from the general library that converts different kinds of data to a number.*

Is it possible to access the list source program of a Scheme definition? In other words, is it possible to introspect and reflect about a Scheme program from another Scheme program. Or even more interesting perhaps, is it possible for a function to introspect and affect its own source code? Using the standard Scheme functions the answers are 'no'. However, some Scheme systems allow it nevertheless, through use of non-standard functions. Using more traditional Lisp languages, the answers are 'yes'.

> In Scheme it is not intended that the program source should be introspected by the running program
>
> But in other Lisp systems there is easy access to self reflection

6.9. References

[-] R5RS: Vectors
http://www.cs.auc.dk/~normark/prog3-03/external-material/r5rs/r5rs-html/r5rs_40.html

[-] Table functions in the HTML4.0 convenience library
http://www.cs.auc.dk/~normark/scheme/distribution/laml/lib/html4.0-loose/man/convenience.html#SECTION5

[-] Manual entry of alist-to-propertylist
http://www.cs.auc.dk/~normark/scheme/distribution/laml/lib/man/general.html#alist-to-propertylist

[-] Manual entry of propertylist-to-alist
http://www.cs.auc.dk/~normark/scheme/distribution/laml/lib/man/general.html#propertylist-to-alist

[-] The HTML document that illustrates the property list representation of attributes
external-material/laml-doc-proplist.html

[knoopnotes] Associative arrays - OOP (in Danish)
http://www.cs.auc.dk/~normark/prog1-01/html/noter/arrays-lister-note-associative-arrays.html

[-] List functions in the general LAML library
http://www.cs.auc.dk/~normark/scheme/distribution/laml/lib/man/general.html#SECTION6

[-] R5RS: Quasiquotation
http://www.cs.auc.dk/~normark/prog3-03/external-material/r5rs/r5rs-html/r5rs_37.html#SEC39

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

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

[-] R5RS: List and pair functions in Scheme
http://www.cs.auc.dk/~normark/prog3-03/external-material/r5rs/r5rs-html/r5rs_58.html#SEC60

[abelson98] Richard Kelsey, William Clinger and Jonathan Rees, "Revised^5 Report on the Algorithmic Language Scheme", *Higher-Order and Symbolic Computation*, Vol. 11, No. 1, August 1998, pp. 7--105.

7. Other Data Types

There are other kinds of data than lists and numbers. In this chapter we will - relatively briefly - review booleans, characters, symbols, vectors, and strings in Scheme.

7.1. Other simple types

Lecture 2 - slide 27

As most other programming languages Scheme supports the simple types of booleans and characters. As a slightly more specialized type, Scheme also supports symbols.

You can get the details of these data types by reading in the Scheme Report [Abelson98]. Section 6.3.1 in the report covers the boolean type. Section 6.3.4 is about characters. Symbols are described in section 6.3.3. From the slide and the annotated slide view of this material, there are direct links to these sections of the Scheme Report.

Besides numbers, Scheme also supports booleans, characters, and symbols

- Booleans
  - *True* is denoted by `#t` and *false* by `#f`
  - Every non-*false* values count as true in `if` and `cond`
- Characters
  - Characters are denoted as #\a, #\b, ...
  - Some characters have symbolic names, such as #\space, #\newline
- Symbols
  - Symbols are denoted by quoting their names: `'a` , `'symbol` , ...
  - Two symbols are identical in the sense of `eqv?` if and only if their names are spelled the same way

The equivalence function `eqv?` is similar to `eq?`. See section 6.1 of [Abelson98] for details.

7.2. Vectors

Lecture 2 - slide 28

There are a number of superficial similarities between vectors and list, as supported by Scheme. However, at the conceptual level vectors are arrays, and lists are linearly linked structures. As such, they represent quite different structures.

The most basic and fundamental difference between lists and vectors is that lists can be changed and extended in a very flexible way (due to the use of dynamically allocated cons cells). A vector is of fixed and constant size once allocated.

Vectors are treated in section 6.3.6 of [Abelson98].

Vectors in Scheme are heterogeneous array-like data structures of a fixed size

- Vectors are denoted in a similar way as list
  - Example: `#(0 a (1 2 3))`
  - Vectors must be quoted in the same way as list when their external representations are used directly
- The function `vector` is similar to the function `list`
- There are functions that convert a vector to a list and vice versa
  - `vector->list`
  - `list->vector`

The main difference between lists and vectors is the mode of access and the mode of construction

There is direct access to the elements of a vector. List elements are accessed by traversing a chain of references. This reflects the basic differences between arrays and linked lists.

The mode of construction for list is recursive, using the `cons` function. Lists are created incrementally: New elements can be created when needed, and prepended to the list. Vectors are allocated in one chunck, and cannot be enlarged or decreased incrementally.

7.3. Strings

Lecture 2 - slide 29

There are no big surprises in the way Scheme handles and supports strings. Please see section 6.3.5 of [Abelson98] for details.

String is an array-like data structure of fixed size with elements of type character.

- The string and vector types have many similar functions
- A number of functions allow lexicographic comparisons of strings:
  - `string=?`, `string<?`, `string<=?`, ...
  - There are case-independent, `ci`, versions of the comparison functions.
- The `substring` function extracts a substring of a string

Like lists, strings are important for many practical purposes, and it is therefore important to familiarize yourself with the string functions in Scheme

7.4. References

[-] Other LAML String functions
http://www.cs.auc.dk/~normark/scheme/distribution/laml/lib/man/general.html#SECTION9

[-] LAML String predicates
http://www.cs.auc.dk/~normark/scheme/distribution/laml/lib/man/general.html#SECTION8

[-] R5RS: Strings
http://www.cs.auc.dk/~normark/prog3-03/external-material/r5rs/r5rs-html/r5rs_61.html

[-] R5RS: Vectors
http://www.cs.auc.dk/~normark/prog3-03/external-material/r5rs/r5rs-html/r5rs_62.html

[-] R5RS: Equivalence predicates
http://www.cs.auc.dk/~normark/prog3-03/external-material/r5rs/r5rs-html/???

[-] R5RS: Symbols
http://www.cs.auc.dk/~normark/prog3-03/external-material/r5rs/r5rs-html/r5rs_59.html#SEC61

[-] R5RS: Characters
http://www.cs.auc.dk/~normark/prog3-03/external-material/r5rs/r5rs-html/r5rs_60.html#SEC62

[-] R5RS: Booleans
http://www.cs.auc.dk/~normark/prog3-03/external-material/r5rs/r5rs-html/r5rs_57.html#SEC59

[abelson98] Richard Kelsey, William Clinger and Jonathan Rees, "Revised^5 Report on the Algorithmic Language Scheme", *Higher-Order and Symbolic Computation*, Vol. 11, No. 1, August 1998, pp. 7--105.

8. Functions

We have now reached the most central concept in this material, namely functions. Functions play a key role in the functional programming paradigm.

Before we look at the function concept as such, we will take a look at definitions.

8.1. Definitions

Lecture 2 - slide 31

A definition binds a value to a name. The name is often referred to as a *variable*. The value bound to a name may be a function value (function object/closure), but it may also be another kind of value, such as a number or a list.

A *definition* binds a name to a value

Below we show the syntactic form of a definition in Scheme.

```
(define name expression)
```

Syntax 8.1 *A name is first introduced and the name is bound to the value of the expression*

- About Scheme `define` forms
  - Appears normally at *top level* in a program
  - Creates a new location named `name` and binds the value of `expression` to that location
  - In case the location already exists we have *redefinition*, and the `define` form is equivalent to the assignment `(set! name expr)`
  - Does not allow for imperative programming, because `define` cannot appear in selections, iterations, etc.
  - Can also appear at certain positions in bodies, but only as syntactic sugar for local binding forms (`letrec`)

In Section 8.12 we discuss definition of functions, and a particular variation of `define` which applies only for function definition.

As it is stated in the first item, define forms appear normally, but not necessarily at top level of a Scheme program. By top level we mean 'at the outer level' of a program - not nested into other constructs.

It is, however, possible to have `define` forms at certain other locations in a Scheme program. The Scheme Report [Abelson98] explains this in section 5.2. Later in this material, in Section 28.3, where we discuss simulation of object-oriented concepts in Scheme, we will use nested `define` forms.

8.2. The function concept

Lecture 2 - slide 33

We start our coverage of functions with the observation that there is both a conceptual and a notational starting point.

> The *conceptual starting point* is the well-known mathematical concept of functions
>
> The *notational starting point* is lambda calculus

The conceptual starting point is well-known for most readers, due to the common knowledge of the mathematical meaning of functions.

The notational starting point is probably not familiar to very many readers. It happens to be the case that the notational inspiration of lambda calculus is quite superficial, as applied in Scheme and Lisp.

- **The mathematical function concept**
  - A mapping from a domain to a range
  - A function transfers values from the domain to values in the range
    - A value in the domain has at most a single corresponding value in the range
  - *Totally* or *partially* defined functions
  - *Extensionally* or *intensionally* defined functions
- **Lambda calculus**
  - A very terse notation of functions and function application

An extensionally defined function is defined by a set of pairs, enumerating corresponding elements in the domain and range. Notice that this causes practical problems if there are many different values in the domain of the function. An intensionally defined function is based on an algorithm that describes how to bring a value from the domain to the similar value in the range. This is a much more effective technique to definition of most the functions, we program in the functional paradigm.

Before we continue the conceptual and programming-related discussion of functions, we will in Section 8.3 take a closer look at the notational starting point.

8.3. Lambda calculus

Lecture 2 - slide 34

We will here introduce the notation of the lambda calculus, mainly in order to understand the inspiration which led to the concept of lambda expressions in Lisp and Scheme.

Lambda calculus is a more dense notation than the similar Scheme notation

	Lambda calculus	**Scheme**
Abstraction	? v . E	(lambda (v) E)
Combination	E1 E2	(E1 E2)

Table 8.1 *A comparison of the notations of abstraction and combination (application) in the lambda calculus and Lisp. In some variants of lambda calculus there are more parentheses than shown here: (? v . E). However, mathematicians tend to like ultra brief notation, and they often eliminate the parentheses. This stands as a contrast to Lisp and Scheme programmers.*

8.4. Functions in Scheme

Lecture 2 - slide 35

On this page we introduce the crucial distinction between a lambda expression and a function object. Lambda expressions are part of a source program. Lambda expressions can be evaluated as all other Scheme expressions. The value of a lambda expression is a function object.

Functions are represented as *lambda expressions* in a source program

At run time, functions are represented as first class *function objects*

Below we show a sample dialogue with a Scheme system. In the dialogue we define functions, and we play with them in order to illustrate some of the basic properties of function in relation to function definition and application. Please play with these elements yourself!

```
> (define x 6)

> (lambda (x) (+ x 1))
#<procedure>

> (define inc (lambda (x) (+ x 1)))

> inc
#<procedure:inc>

> (if (even? x) inc fac)
#<procedure:inc>
```

```
> ((if (even? x) inc fac) 5)
6
```

Program 8.1 *A sample read-eval-print session with lambda expressions and function objects. In a context where we define* `x` *to the number 6 we first evaluate a lambda expression. Scheme acknowledges this by returning the function object, which prints like* `' #<procedure> '`*. As a contrast to numbers, lists, and other simple values, there is no good surface representation of function values (function objects). Next we bind the name* `inc` *to the same function object. More about name binding in a later part of this material. The expression* `(if (even? x) inc fac)` *returns* `inc` *because the value of* `x` *is 6, and as such it is even. Therefore the value of* `((if (even? x) inc fac) 5)` *is the same as the value of* `(inc 5)`*, namely 6.*

8.5. Function objects

Lecture 2 - slide 36

Let us now define the concepts of *function objects* and *closures*.

A *function object* represents a function at run time. A function object is created as the value of a lambda expression

A function object is also known as a *closure*.

A function object is a first class value at run time, in the same way as numbers, lists and other data are values. This is different from more traditional programming languages, where procedural and functional abstractions have another status than ordinary data.

The name 'closure' is related to the interpretation of free names in the body expression of the function. Free names are used, but not defined in the body. In a function object (or closure) the free names are bound in the context of the lambda expression. This is a contrast to the case where the free names are bound in the context of the application of the function.

- Characteristics of function objects:
  - First class objects
  - Does not necessarily have a name
  - A function object can be bound to a name in a definition
  - Functions as closures:
    - Capturing of free names in the context of the lambda expression
    - Static binding of free names
    - A closure is represented as a pair of *function syntax* and *values of free names*
  - A function object can be applied on actual parameters, passed as a parameter to a function, returned as the result from another function, and organized as a constituent of a data structure

The first characteristics of functions, as mentioned in the itemized lists above, is 'the first class status'. We will consolidate our understanding of first class 'citizens' in Section 8.6 .

8.6. Functions as first class values

Lecture 2 - slide 37

As it is discussed in this section, first class entities in a language can be passed as parameters, returned as results, and organized in data structures.

We are used to the first class status of numbers and lists. But with a background from imperative programming, we are not used to organize functions and procedures in data structures, and we are not used to the possibility of returning procedures and functions from other abstractions.

Notice that objects, as known from the object-oriented paradigm, are of first class.

Here is our definition of being 'of first class'.

A *first class citizen* is an entity which can be passed as parameter to functions, returned as a result from a function, and organized as parts of data structures

A function object is a first class citizen

In Program 8.2 we show an interaction with a Scheme system, which illustrates that functions can be used as elements in data structures.

```
1> (define toplevel-html-elements (list html frameset))

2> overall-html-elements
(#<procedure> #<procedure>)

3> ((cadr toplevel-html-elements) (frame 'src "sss"))
(ast "frameset" ((ast "frame" () (src "sss") single)) () double)

4> (xml-render ((cadr toplevel-html-elements) (frame 'src "sss")))
"<frameset><frame src = \"sss\"></frameset>"
```

Program 8.2 *A few interactions which illustrate the first class properties of function objects. We bind the variable* `toplevel-html-elements` *to the list of the two functions* `html` *and* `frameset`*. Both are HTML mirror functions defined in the LAML general library. We illustrate next that the value of the variable indeed is a list of two functions. Thus, we have seen that we can organized functions as elements in lists. The function* `cadr` *returns the second element of a list. It is equivalent to* `(compose car cdr)`*, where* `compose` *is functional composition. In the third evaluation we apply the mirror function* `frameset` *on a single frame. The last interaction shows the HTML rendering of the this.* `xml-render` *is a function defined in the LAML general library.*

8.7. Anonymous functions

Lecture 2 - slide 38

The reader may believe that a function name is a necessary constituent of a function. This understanding is not correct, however. We can chose to associate a name with a function by using an enclosing define form, as explained in Section 8.1. But the function itself is not named.

> A function object does not have a name, and a function object is not necessarily bound to a name

The interactions below illustrate the use of anonymous functions, i.e., functions without names.

```
1> ((lambda(x) (+ x 1)) 3)
4

2> (define fu-lst (list (lambda (x) (+ x 1)) (lambda (x) (* x 5))))

3> fu-lst
(#<procedure> #<procedure>)

4> ((second fu-lst) 6)
30
```

Program 8.3 *An illustration of anonymous functions. The function* `(lambda(x) (+ x 1))` *is the function that adds one (to its parameter). It is organized in a list side by side with the function that multiplies by five. Notice in this context that none of these two functions are named. In the last interaction we apply the latter to the number 6.*

8.8. Lambda expressions in Scheme

Lecture 2 - slide 39

The syntax definitions in Syntax 8.2 and Syntax 8.3 below show the two possible forms of lambda expressions.

Each of the formal parameters in a formal parameter list are *binding name occurrences*. It means that a formal parameter introduces a new name with a new role. The new name can be used and referred from other parts of the program. We can talk about the *scope* of the name as the area of the program in the which the binding is in effect. The scope of a formal parameter is - quite naturally - the body expression of the lambda form.

In the first syntax definition, `formal-parameter-list` is a list of formal parameters. The formal parameter list may be improper, such as `(a b . c)`. In this case all actual parameters after the second one is bound to `c`.

```
(lambda (formal-parameter-list) expression)
```

Syntax 8.2

In the second case, the list of actual parameters is simply bound to the name `formal-parameters-name`.

Be sure to understand the correspondence between formal parameters (in the two forms) and the actual parameters. Use Exercise 2.6 to strengthen your understanding.

```
(lambda formal-parameters-name expression)
```

Syntax 8.3

- Lambda expression characteristics in Scheme:
  - No type declaration of formal parameter names
  - Call by value parameters
    - In reality passing of references to lists and other structures
  - Positional and required parameters
    - `(lambda (x y z) expr)` accepts exactly three parameters
  - Required and rest parameters
    - `(lambda (x y z . r) expr)` accepts three or more parameters
  - Rest parameters only
    - `(lambda r expr)` accepts an arbitrary number of parameters

Exercise 2.6. *Parameter passing in Scheme*

Familiarize yourself with the parameter passing rules of Scheme by trying out the following calls:

```
((lambda (x y z) (list x y z)) 1 2 3)
((lambda (x y z) (list x y z)) 1 2)
((lambda (x y z) (list x y z)) 1 2 3 4)
((lambda (x y z . r) (list x y z r)) 1 2 3)
((lambda (x y z . r) (list x y z r)) 1 2)
((lambda (x y z . r) (list x y z r)) 1 2 3 4)
((lambda r r) 1 2 3)
((lambda r r) 1 2)
((lambda r r) 1 2 3 4)
```

Be sure that you can explain all the results

8.9. Optional parameters of Scheme functions (1)

Lecture 2 - slide 40

In LAML software we use a particular pattern to deal with optional parameters. This pattern is built on top of the rest parameter mechanism discussed in Section 8.8. The pattern also involves a function `optional-parameter`, defined in the LAML general library, as an important brick.

When we use optional parameters of a function, the caller may chose not to pass a value. In that case, the parameter is bound to a *default value*, which is defined as part of the function.

> It is often useful to pass one or more optional parameters to a function
>
> In case an optional parameter is not passed explicitly, a default value should apply

The example in Program 8.4 illustrates how to define a function which requires one parameter `rp`, and up to three optional parameters `op1`, `op2`, and `op3`.

```
(define (f rp . optional-parameter-list)
 (let ((op1 (optional-parameter 1 optional-parameter-list 1))
       (op2 (optional-parameter 2 optional-parameter-list "a"))
       (op3 (optional-parameter 3 optional-parameter-list #f)))
  (list rp op1 op2 op3)))
```

Program 8.4 *A example of a function f that accepts optional-parameters. Besides the required parameter rp, the function accepts an arbitrary number of additional parameters, the list of which are bound to the formal parameter optional-parameter-list. The function optional-parameter from the LAML general library accesses information from optional-parameter-list. In case an optional parameter is not passed, the default value (the last parameter of optional-parameter) applies.*

The following dialogue with a Scheme system shows optional parameters in play.

```
0>
(define (f rp . optional-parameter-list)
  (let ((op1 (optional-parameter 1 optional-parameter-list 1))
        (op2 (optional-parameter 2 optional-parameter-list "a"))
        (op3 (optional-parameter 3 optional-parameter-list #f)))
    (list rp op1 op2 op3)))

1> (f 7)
(7 1 "a" #f)

2> (f 7 "c")
(7 "c" "a" #f)

3> (f 7 8)
(7 8 "a" #f)

4> (f 7 8 9)
(7 8 9 #f)

5> (f 7 8 9 10)
(f 7 8 9 10)
```

```
6> (f 7 8 9 10 11)
(7 8 9 10)
```

Program 8.5 *A number of calls of the function f. For clarity we define f as the first interaction.*

In the next section we will discuss a major shortcoming of the optional parameter mechanism.

8.10. Optional parameters of Scheme functions (2)

Lecture 2 - slide 41

Optional parameters, as discussed in Section 8.9 is not a perfect solution in all respects. On this page we will discuss a major weakness.

- Observations about optional parameters:
  - The function `optional-parameter` is a LAML function from the general library
  - The optional parameter idea works well if there is a natural ordering of the relevance of the parameters
    - If parameter *n* is passed, it is also natural to pass parameter *1* to *n-1*
  - The idea does not work well if we need to pass optional parameter number *n*, but not number *1* .. *n-1*

Keyword parameters is a good alternative to optional parameter lists in case many, 'unordered' parameters need to passed to a function

We have demonstrated how we simulate optional parameter via the 'rest parameter list' mechanism in Scheme. It is also possible to simulate a keyword parameter mechanism. In a LAML context, this is done with respect to the passing of attributes to the HTML mirror functions.

For more information about the simulation of keyword parameters in HTML and XML mirror functions, please consult Section 26.2.

8.11. Closures

Lecture 2 - slide 42

Function objects are also called *closures*. In this section we will see why.

Functions capture the free names in the context of the lambda expression

The illustration below shows a closure. For a better visualization, you should visit the web version of the page, which uses animation to illustrate the capturing of free names.

When we talk about a *free name* it is always relative to a given construct, such as a lambda expression. A free name in the construct is used, but not bound in the construct. The formal parameter names of a lambda expression are 'binding positions'. Thus, names in the body of a lambda expression, which correspond to formal parameter names of the lambda expressions, are not free names.

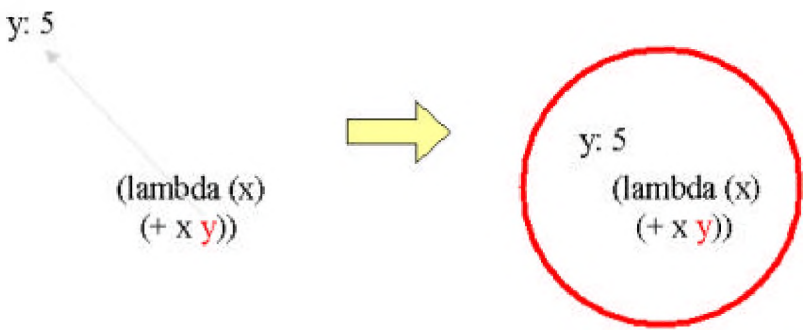

Figure 8.1 *A lambda expression with a free name y. The name y is bound outside the lambda expression. A closure is formed by associating the lambda expressions (the syntactic form) with the binding of the free names.*

In the table below we illustrate free names and closures. Notice that in the inner lambda expression, `(lambda (txt) ...)`, both `p` and `b` are free names, whereas in the lambda expression bound to `f` only `p` is a free name.

Expression	Value

`(define f`
`(let ((b (lambda (x)`
`(string-append x ":" x))))`
`(lambda (txt) (p (b txt)))))`
 	
`(f "A text")`
	A text:A text

`(b "A text")`	**A text**
`(f (b "A text"))`	**A text:A text**

Table 8.2 *Examples of the closuring effect. In the first example b is locally bound to a function which replicates its parameter with a colon in between. f is bound to a function (the inner lambda) in which b refers to the string replicating function. Notice that outside this this context, b is the HTML mirror function that renders a text in bold face.*

8.12. Function definition in Scheme

Lecture 2 - slide 43

In Section 8.1 we studied definitions in general. In a definition we associate a name with a value through the evaluation of an expression. As already discussed there, we can define functions in the same way we associate names with other types of values.

In this section we will study a particular twist on function definitions.

A function object can be bound to a name via `define` like any other kind of value.

But we often use a slightly different, equivalent syntax for function definitions, where the '`lambda`' is implicitly specified

In Syntax 8.4 we show the ordinary way of defining a function. With this, f is bound to a function object.

```
(define f (lambda (p1 p2) ...))
```

Syntax 8.4 *The ordinary way to bind f to the value of a lambda expressions*

In Syntax 8.5 we show an alternative way of defining a function. The second element of the define form is a list, which corresponds to the calling profile of the function. The two definitions are fully equivalent, and it is a matter of style and personal preference which one to use.

I typically use the form in Syntax 8.5 because it is a little more shallow (with respect to parentheses) than the one in Syntax 8.4. As another reason, it is nice to have the calling form as a constituent of the definition. It is often convenient to copy it out of the definition to some context, in which the function is to be called.

```
(define (f p1 p2) ...)
```

Syntax 8.5 *An equivalent syntactic sugaring with a more shallow parenthesis structure. Whenever Scheme identifies a list at the 'name place' in a define form, it carries out the transformation* `(define (f p1 p2) ...) => (define f (lambda (p1 p2) ...))`. *Some Scheme programmers like the form* `(define (f p1 p2) ...)` *because the calling form* `(f p1 p2)` *is a constituent of the form* `(define (f p1 p2) ...)`

8.13. Simple web-related functions (1)

Lecture 2 - slide 44

We will here give a simple example of a web-related function. Much more interesting examples will appear later in the material.

The programs in Program 8.6 and Program 8.7 show the definition and a call of a www-document function, which abstracts the outer HTML elements. In the web version of the material you will, in addition, find an illustration with all the LAML details necessary to execute the example.

```
(define (www-document the-title . body-forms)
 (html
  (head (title the-title))
  (body body-forms)))
```

Program 8.6 *The definition of a* `www-document` *function. The* `www-document` *function is useful if you want to abstract the HTML envelope formed by the elements html, head, title, and body. If you need to pass attributes to html or body the proposed function is not adequate.*

```
(www-document
  "This is the document title"
  (h1 "Document title")

  (p "Here is the first paragraph of the document")

  (p "The second paragraph has an" (em "emphasized item")
     "and a" (em "bold face item")_"."))
```

Program 8.7 *A sample application of the function* `www-document`. *Notice the way we pass a number of body contributions, which - as a list - are bound to the formal parameter body-forms.*

8.14. Simple web-related functions (2)

Lecture 2 - slide 45

The example on this page shows an indent-pixel function, which indents a block of text a number of pixels to the right.

In Program 8.8 you find a version which is implemented in terms of tables.

```
(define (indent-pixels p . contents)
  (table 'border "0"
   (tbody
     (tr
      (td 'width (as-string p) "")
      (td 'width "*" contents)))))
```

Program 8.8 *The definition of the* `indent-pixel` *function. This is a function which we use in many web documents to indent the contents a number of pixels relative to its context. Here we implement the indentation by use of a table, in which the first column cell is empty. As we will se, other possibilities exist.*

In Program 8.9 we show an alternative version of indent-pixels, which is implement by use of Cascading Style Sheets (CSS) features.

```
(define (indent-pixels p . contents)
  (div 'css:margin-left (as-string p)
    contents))
```

Program 8.9 *An alternative version of the* `indent-pixel` *function. This version uses Cascading Style Sheets expressiveness. As it appears, this is a more compact, and more direct way of achieving our indentation goal.*

Below, in Program 8.10 we show a sample application of the `indent-pixels` function. The program in Program 8.10 is complete and self contained relative to the LAML libraries.

In the web version of the material (slide or annotated slide view) you will find references to the generated documents.

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

(define (indent-pixels p . contents)
  (div 'css:margin-left (as-string p)
    contents))

(write-html 'raw
 (html
  (head (title "Indent Pixel Example"))
  (body

    (p "Here is some initial text")

    (indent-pixels 45
       (p "First paragraph of indented text")
       (p "Second paragraph of indented text")
    )

    (p "Here is some final text"))))
```

Program 8.10 *A sample application of* `indent-pixel` *with some initial LAML context (software loading). Notice the use of the XHTML mirror.*

8.15. Function exercises

Lecture 2 - slide 46

In this last section of the chapter we provide a couple of extra exercises.

Exercise 2.7. *Colors in HTML*

In HTML we define colors as text strings of length 7:

```
"#rstuvw"
```

The symbols *r, s, t, u, v*, and *w* are all hexadecimal numbers between 0 and f (15). *rs* is in that way the hexadecimal representation for red, *tu* is the code for green, and *vw* is the code for blue.

As an example, the text string

```
"#ffffff"
```

represents white and

```
"#ff0000"
```

is red.

In Scheme we wish to represent a color as the list

```
(color r g b)
```

where color is a symbol, r is number between 0 and 255 which represents the amount of red, and g and b in a similar way the amount of green and blue in the color.

Write a Scheme function that transforms a Scheme color to a HTML color string.

It is a good training to program the function that converts decimal numbers to hexa decimal numbers. I suggest that you do that - I did it in fact in my solution! If you want to make life a little easier, the Scheme function `(number->string n radix)` is helpful (pass radix 16 as second parameter).

Exercise 2.8. *Letter case conversion*

In many web documents it is desirable to control the letter case of selected words. This allows us to present documents with consistent appearances. Therefore it is helpful to be able to capitalize a string, to transform a string to consist of upper case letters only, and to lower case letters only. Be sure to leave non-alphabetic characters untouched. Also, be sure to handle the Danish characters 'æ', 'ø', and 'å' (ASCII 230, 248, and 229 respectively). In addition, let us emphasize that we want functions that do not mutate the input string by any means. (It means that you are not allowed to

modify the strings passed as input to your functions).

Write functions `capitalize-a-string`, `upcase-a-string`, `downcase-a-string` for these purposes.

As examples of their use, please study the following:

```
(capitalize-a-string "monkey") => "Monkey"
(upcase-a-string "monkey") => "MONKEY"
(downcase-a-string "MONkey") => "monkey"
```

Hint: I suggest that you program the necessary functions yourself. Convert the string to a list of ASCII codes, do the necessary transformations on this list, and convert the list of modified ASCII codes back to a string. The Scheme functions `list->string` and `string->list` are useful.

Hint: If you want to make life a little easier (and learn less from this exercise...) you can use the Scheme functions `char-upcase` and `char-downcase`, which work on characters. But these functions do maybe not work on the Danish letters, so you you probably need some patches.

8.16. References

[-] The second version of the indent-pixels document
external-material/indent-pixels-2.html

[-] The first version of the indent-pixels document
external-material/indent-pixels-1.html

[-] Manual entry of optional-parameter
http://www.cs.auc.dk/~normark/scheme/distribution/laml/man/laml.html#optional-parameter

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

[-] R5RS: Procedures (Functions)
http://www.cs.auc.dk/~normark/prog3-03/external-material/r5rs/r5rs-html/r5rs_28.html

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

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

[-] R5RS: Definitions
http://www.cs.auc.dk/~normark/prog3-03/external-material/r5rs/r5rs-html/r5rs_43.html

[abelson98] Richard Kelsey, William Clinger and Jonathan Rees, "Revised^5 Report on the Algorithmic Language Scheme", *Higher-Order and Symbolic Computation*, Vol. 11, No. 1, August 1998, pp. 7--105.

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* ((n1 e1)
       ...
       (ni-1 ei-1)
       (ni ei)
       ...
       (nk ek))
  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].

```
(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 ala `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 $\texttt{expr}_1$ if `bool-expr` is true, and $\texttt{expr}_2$ if `bool-expr` is false
- `cond` evaluates the first expression $\texttt{expr}_i$ whose guarding $\texttt{bool-expr}_i$ is true. If $\texttt{bool-expr}_1$, ..., $\texttt{bool-expr}_k$ are all false, the value of `cond` becomes the value of $\texttt{expr}_{k+1}$

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.*

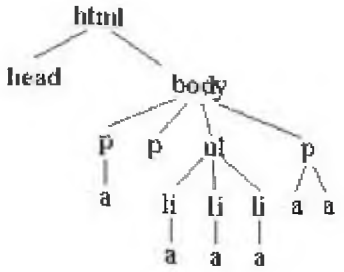

Figure 11.1 *The syntax tree of the web document with the root made up by the `html` element.*

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.

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.

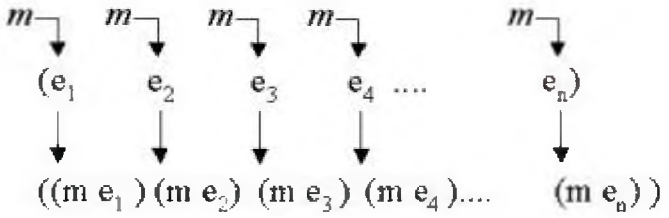

Figure 15.1 *Mapping a function m on a list. m is applied on every element, and the list of these applications is returned.*

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.

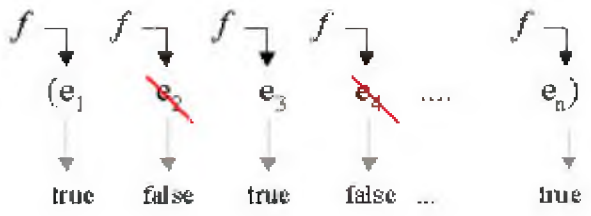

Figure 15.2 *Filtering a list with a predicate f. The resulting list is the subset of the elements which satisfy f (the elements on which f returns true).*

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.

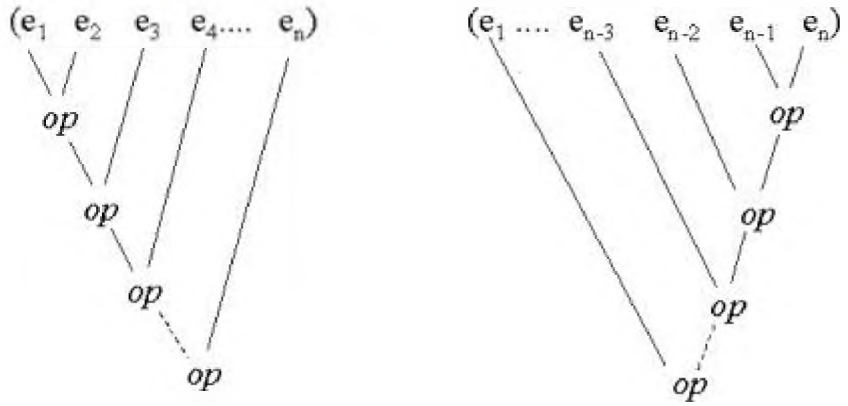

Figure 16.1 *Left and right reduction of a list. Left reduction is - quite naturally - shown to the left, and right reduction to the right.*

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

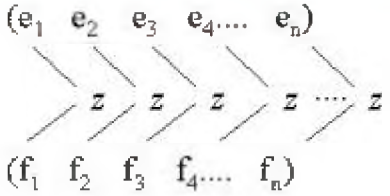

Figure 16.2 *Zipping two lists with the function z. The head of the resulting list is* `(z e`$_i$ `f`$_i$*), where the element* `e`$_i$ *comes from the first list, and* `f`$_i$ *comes from the other.*

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

19. Introduction to evaluation order

At this point in the material we start our coverage of *evaluation order*. We start by discussing the idea of referential transparency.

19.1. Referential transparency

Lecture 5 - slide 2

The main idea behind the concept of referential transparency is captured by the point below.

> Two equal expressions can substitute each other without affecting the meaning of a functional program

As formulated above, the possibility of substituting one expression by another depends on whether or not the two expressions are considered as being equal. As noticed in Section 4.5 there are a number of different interpretations of equality around in Scheme, as well as in other programming languages.

As we observe in the items below, we can use even the weakest form of equality, namely structural, deep equality, for our observations about referential transparency. In other words, if two structures are structurally equal, the expressions and values involved may substitute each other.

- Referential transparency
  - provides for easy equational reasoning about a program
  - does not rely on a particular notion of equality
    - Reference equality, shallow equality and deep equality cannot be distinguished by functional means
  - is a major contrast to imperative programming

The idea of referential transparency can be stated very briefly in the following way:

> *Equals can be replaced by equals*

19.2. An illustration of referential transparency

Lecture 5 - slide 3

Before we proceed we will illustrate some practical uses of referential transparency.

With referential transparency it is possible to perform natural program transformations without any concern of side effects

The two expressions in the top left and the top right boxes of Figure 19.1 may substitute each other, provided that the function F is a pure function (without side effects).

In the case where F is a value returning procedure, as illustrated in the bottom box of Figure 19.1 it is clear that it is important how many times F is actually evaluated. The reason is that an evaluation of F affects the value of the variable c, which is used in the top level expressions. (Thus, F is an imperative abstraction - a procedure).

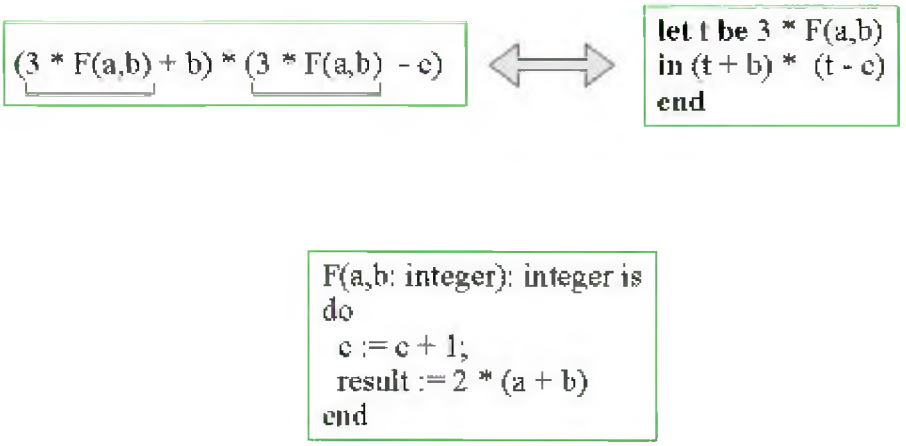

Figure 19.1 *It is possible to rewrite one of the expressions above to the other, provided that F is a function. Below, we have illustrated an example where F is of procedural nature. Notice that F assigns the variable c, such that it becomes critical to know how many times F is called.*

On the ground of this example it is worth observing that equational reasoning about functional programs is relatively straightforward. If procedures are involved, as F in the bottom box of Figure 19.1, it is much harder to reason about the expression, for instance with the purpose of simplifying it (as it is the case when substituting the expression to the left with the expression to the right in the top-part of Figure 19.1).

19.3. Arbitrary evaluation order - with some limits

Lecture 5 - slide 5

In this section we will discuss the order of subexpression evaluation in a composite expression.

In a functional program an expression is evaluated with the purpose of producing a value

An expression is composed of subexpressions

Take a look at one of the expressions in Figure 19.2. The underlying syntax trees are shown below the expressions in the figure. The question is which of the subexpressions to evaluate first, which comes next, and which one is to be the last.

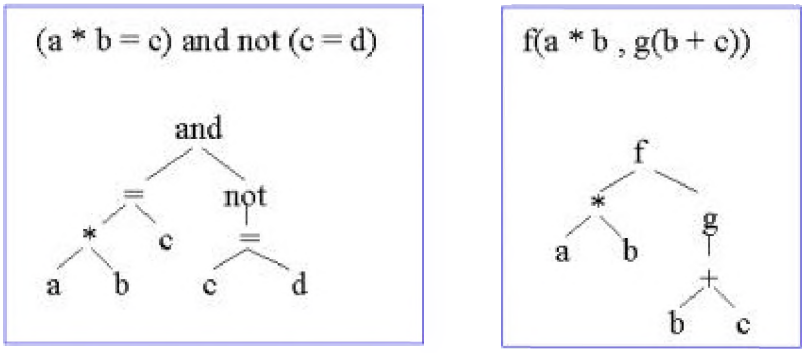

Figure 19.2 *An illustration of an expression with subexpressions.*

To be concrete, we can propose to start from the left leaf, from the right leaf, from the top, or perhaps from some location in the middle of the expression.

In the functional programming context, with expressions and pure functions, we will probably expect that an arbitrary evaluation order is possible. If we should devise a practical recipe we will probably start from one of leafs (say the leftmost leaf) and work our way to the expression in the root.

As noticed below, we can actually use an arbitrary evaluation order, provided that there are no errors in any of the subexpressions, and provided that all of the involved evaluations terminate.

- Subexpressions can be evaluated in an arbitrary order *provided that*
  - no errors occur in subexpressions
  - the evaluation of all subexpressions terminates
- It is possible, and without problems, to evaluate subexpressions in parallel

In the rest of this section, as well as in Chapter 20 we will study and understand the premises and the limits of 'arbitrary evaluation order'.

19.4. A motivating example

Lecture 5 - slide 6

It is valuable to understand the problems and the quirks of evaluation order by looking at a very simple program example.

What is the value of the following expression?

The lambda expression in Program 19.1 shows a pseudo application of the constant function, which returns 1 for every possible input `x`. The tricky part is, however, that we pass an actual parameter expression which never terminates.

```
((lambda (x) 1) some-infinite-calculation)
```

Program 19.1 *A constant function with an actual parameter expression, the evaluation of which never terminates.*

It is not difficult to write a concrete Scheme program, which behave in the same way as Program 19.1. Such a program is shown in Program 19.2. The parameter less function `infinite-calculation` just calls itself forever recursively.

```
(define (infinite-calculation)
  (infinite-calculation))

((lambda (x) 1) (infinite-calculation))
```

Program 19.2 *A more concrete version of the expression from above. The function `infinite-calculation` just calls itself without ever terminating.*

19.5. A motivating example - clarification

Lecture 5 - slide 7

As noticed below, it can be argued that the value of the expression in Program 19.1 is 1, due to the reasoning that the the result of the function `(lambda (x) 1)` is independent of the formal parameter `x`.

It can also be argued that an evaluation of the actual parameter expression `(infinite-calculation)` stalls the evaluation of the surrounding expression, such that the expression in Program 19.1 does not terminate. In Scheme, as well as in most other programming languages, this will be the outcome.

The items below summarizes these two possibilities, and they introduce two names of the two different function semantics, which are involved.

- Different evaluation orders give different 'results'
  - The number 1
  - A non-terminating calculation
- Two different semantics of function application are involved:
  - **Strict**: A function call is well-defined if and only if all actual parameters are well-defined
  - **Non-strict**: A function call can be well-defined even if one or more actual parameters cause an error or an infinite calculation

In most languages, functions are strict. This is also the case in Scheme. In some languages, however, such as Haskell and Miranda, functions are non-strict. As we will see in the following, languages with non-strict functions are very interesting, and they open up new computational possibilities.

19.6. References

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

20. Rewrite rules, reduction, and normal forms

At this point in the material it will be assumed that the reader is motivated to study evaluation order in some detail.

An evaluation of an expression can be understood as a transformation of the expressions which preserves its meaning. In this chapter we will see that transformations can be done incrementally in *rewriting steps*. A rewriting of an expression gives a new expression which is semantically equivalent to the original one. Usually, we go for rewritings which simplify an expression. In theory, however, we could also rewrite an expression to more complicated expression.

In this chapter we will formally characterize the *value of an expression*, using the concept of a normal form. We will see that the value of an expression is an expression in itself that cannot be rewritten to simpler forms by use of any rewriting rules.

As a key insight in this chapter, we will also see that an expression can be reduced to a value (a normal form) in many different ways. We will identify and name a couple of these, and we will discuss which of the evaluation strategies is the 'best'.

20.1. Rewrite rules

Lecture 5 - slide 9

This section gives an overview of the rewrite rules, we will study in the subsequent sections.

The *rewrite rules* define semantics preserving transformations of expressions

The goal of applying the rewrite rules is normally to reduce an expression to the simplest possible form, called a *normal form*.

- Overview of rewrite rules
  - **Alpha conversion:** rewrites a lambda expression
  - **Beta conversion:** rewrites general function calls
    - Expresses the idea of substitution, as described by *the substitution model*
  - **Eta conversion:** rewrites certain lambda expressions to a simpler form

The Beta conversion corresponds to the *substitution model* of function calls, which is explained in [Abelson96]. (See section of 1.1.5 of [Abelson96] for the details).

20.2. The alpha rewrite rule

Lecture 5 - slide 10

The first rewrite rule we encounter is called the alpha rewrite rule. From a practical point of view this rule is not very interesting, however. The rule tells under which circumstances it is possible to use other names of the formal parameters. Recall in this context that the formal parameter names are binding name occurrences, cf. Section 8.8.

An alpha conversion changes the names of lambda expression formal parameters

Here comes the formulation of the alpha rewrite rule.

Formal parameters of a lambda expression can be substituted by other names, which are not used as free names in the body

Recall in the context of this discussion that a free name in a construct is applied, but not bound (or defined) in the construct. See Section 8.11 for additional details about free names.

In Table 20.1 we see an example of a legal use of the alpha rewrite rule. The formal names `x` and `y` of the lambda expression are changed to `a` and `b`, respectively. It is fairly obvious that this causes no problems nor harm. The resulting lambda expression is fully equivalent with the original one.

Expression	Converted Expression
`(lambda (x y) (f x y))`	`(lambda (a b) (f a b))`

Table 20.1 *An example of an alpha rewriting. The name a replaces x and the name y replaces y.*

More interesting, we show an example of an illegal use of the alpha rewrite rule in Table 20.2. Again we change the name `x` to `a`. The name of the other formal parameter is changed to `f`. But `f` is a free name in the lambda expression. It is easy to see that the converted expression in Table 20.2 has changed its meaning. The name `f` is now bound in the formal parameter list. Thus, the rewriting in Table 20.2 is illegal.

Expression	Converted Expression
`(lambda (x y) (f x y))`	`(lambda (a f) (f a f))`

Table 20.2 *Examples of an illegal alpha conversion. f is a free name in the lambda expression. A free name is used, but not defined (bound) in the lambda expression. In case we rename one of the parameters to f, the free name will be bound, hereby causing an erroneous name binding.*

20.3. The beta rewrite rule

Lecture 5 - slide 11

The beta rewrite rule is the one to watch carefully, due to its central role in any evaluation process that involves the calling of functions.

A beta conversion tells how to evaluate a function call

The beta rewrite rules goes as follows.

An application of a function can be substituted by the function body, in which formal parameters are substituted by the corresponding actual parameters

It is worth noticing that there are no special conditions for the application of the beta rewrite rule. In that way the rule is different from both the alpha rewrite rule, which we studied in Section 20.2, and it is also different from the eta rewrite rule which we encounter in Section 20.4 below. All the examples in Table 20.3 are legal examples of beta rewritings.

Expression	Converted Expression
`((lambda(x) (f x)) a)`	`(f a)`
`((lambda(x y) (* x (+ x y))) (+ 3 4) 5)`	`(* 7 (+ 7 5))`
`((lambda(x y) (* x (+ x y))) (+ 3 4) 5)`	`(* (+ 3 4) (+ (+ 3 4) 5))`

Table 20.3 *Examples of beta conversions. In all the three examples the function calls are replaced by the bodies. In the bodies, the formal parameters are replaced by actual parameters.*

Be sure to understand that the beta rewrite rule tells us how to implement a function call, at least in a principled way. In a practical implementation, however, the substitution of formal parameters by (more or less evaluated) actual parameters is not efficient. Therefore, in reality, the bindings of the formal parameters are organized in name binding frames, in so-called environments. Thus, instead of name substitution (as called for in the beta rewrite rules), the formal names are looked up in a name binding environment, when they are needed in the body of the lambda expression.

The implementation of `eval` in a Scheme interpreter describes the details of a practical and real life use of the beta rewrite rule. See Section 24.3 for additional details.

20.4. The eta rewrite rule

Lecture 5 - slide 12

The eta rewrite rules transforms certain lambda expressions. As such the eta rewrite rule is similar to the alpha rewrite rule, but radically different from the beta rewrite rule.

An eta conversion lifts certain function calls out of a lambda convolute

In loose terms, the eta rewrite rule can be formulated in the following way. Be aware, however, that there is a condition associated with applications of the eta rewrite rule. The condition is described below.

A function f, which only passes its parameters on to another function e, can be substituted by e

Here is a slightly more formal - and more precise - description of the eta rewrite rule:

`(lambda(x) (e x)) <=> e` provided that `x` is not free in the expression `e`

In the same way as above for alpha conversions in Section 20.2 we will give examples of legal and illegal uses of the eta rule.

The example in Table 20.4 shows that the lambda expression around square is superfluous. In the eta-rewritten expression, the lambda surround of `square` is simply discarded.

Expression	Converted Expression
`(lambda (x) (square x))`	`square`

Table 20.4 *An example of an eta rewriting.*

It is slightly more complicated to illustrate an illegal use of the rule. In the expression of the left cell in Table 20.5 we are attempting to eliminate the outer lambda expression by use of the eta rewrite rule. Notice, however, that `x` is free in the inner blue lambda expression. Therefore the eta rewriting illustrated in Table 20.5 is not legal. By applying the rewriting rule on the left part of Table 20.5 anyway we loose the binding of `x`, and therefore the rewriting does not preserve the semantics of the left cell expression.

Expression	Converted Expression
`(lambda(x) ((lambda(y) (f x y)) x))`	`(lambda(y) (f x y))`

Table 20.5 *An example of an illegal eta conversion. The eta conversion rule says in general how 'e' is lifted out of the lambda expressions. In this example, e corresponds to the emphasized inner lambda expression (which is blue on a color medium.) However, x is a free name in the inner lambda expression, and therefore the application of the eta rewrite rule is illegal.*

This completes our discussion of rewriting rules, and we will now look at the concept of normal forms.

20.5. Normal forms

Lecture 5 - slide 13

As already mentioned above, the value v of an expression e is a particular simple expression which is semantically equivalent with e. The expression v is obtained from e by a number of rewriting steps.

Normal forms represent our intuition of the value of an expression

Here is the definition of a normal form.

An expressions is on *normal form* if it cannot be reduced further by use of beta and eta conversions

Notice in the definition that we talk about reduction. By this is meant application of the rewrite rules 'from left to right'.

- About normal forms
  - Alpha conversions can be used infinitely, and as such they do not play any role in the formulation of a normal form
  - A normal form is a particular simple expression, which is equivalent to the original expression, due to the application of the conversions

Normal forms are simple to understand. But there are a number of interesting and important questions that need to be addressed. One of them is formulated below.

Is a normal form always unique?

The answer to the question will be found in Section 20.9.

20.6. The ordering of reductions

Lecture 5 - slide 14

As discussed in Section 19.3 we can expect that the concrete order of evaluation steps will matter, especially in the cases where errors or infinite calculations are around in some of the subexpressions.

Evaluation steps are now understood as reductions with the beta or eta rewrite rule.

In this section we will identify and name a couple of evaluation strategies or plans. Such a strategy determines the order of use of the beta and eta reduction rules.

Given a complex expression, there are many different orderings of the applicable reductions

Using *normal-order reduction*, the first reduction to perform is the one at the outer level of the expression

Using *applicative-order reduction*, the first reduction to perform is the inner leftmost reduction

- Normal-order reduction represents *evaluation by need*
- Applicative-order reduction evaluates all constituent expressions, some of which are unnecessary or perhaps even harmful. As such, there is often a need to *control the evaluation* process with *special forms* that use a non-standard evaluation strategy

Let it be clear here, that many other evaluation strategies could be imagined. The practical relevance of additional strategies is another story, however.

Applicative-order reduction represents 'the usual' evaluation strategy, used for expressions in most programming languages. Normal-order reduction represents a new approach, which is used in a few contemporary functional programming languages.

In Section 20.10 we will discuss examples of the special forms mentioned in the item discussing the applicative-order reduction.

20.7. An example of normal versus applicative evaluation

Lecture 5 - slide 15

Let us illustrate the difference between normal-order reduction and applicative-order reduction via a concrete example.

Reduction of the expression `((lambda(x y) (+ (* x x) (* y y))) (fak 5) (fib 10))`

The example involves an application of the blue function `(lambda(x y) (+ (* x x) (* y y)))` on the actual parameters `(fak 5)` and `(fib 10)`. The functions `fak` and `fib` are shown in Program 20.1.

In Program 20.1 we show definitions of `fak` and `fib`, together with the example expression.

```
(define (fak n)
  (if (= n 0) 1 (* n (fak (- n 1)))))

(define (fib n)
  (cond ((= n 0) 0)
        ((= n 1) 1)
        (else (+ (fib (- n 1)) (fib (- n 2))))))

((lambda(x y) (+ (* x x) (* y y)))  (fak 5)  (fib 10))
```

Program 20.1 *The necessary Scheme stuff to evaluate the expression.*

In Figure 20.1 applicative-order reduction is outlined in the leftmost path of the graph. With applicative-order reduction we first evaluate the lambda expression, then `(fak 5)` and `(fib 10)`. The evaluation of the lambda expression gives a function object. Notice that the expensive calculations of `(fak 5)` and `(fib 10)` are only made once. The last step before the addition and the multiplications is a beta reduction, with which the function is called.

The normal order reduction is illustrated with the path to the right in Figure 20.1. The outer reduction is a beta reduction, in which we substitute the non-reduced parameter expressions `(fak 5)` and `(fib 10)`. Notice that the calculation of `(fak 5)` and `(fib 10)` are made twice.

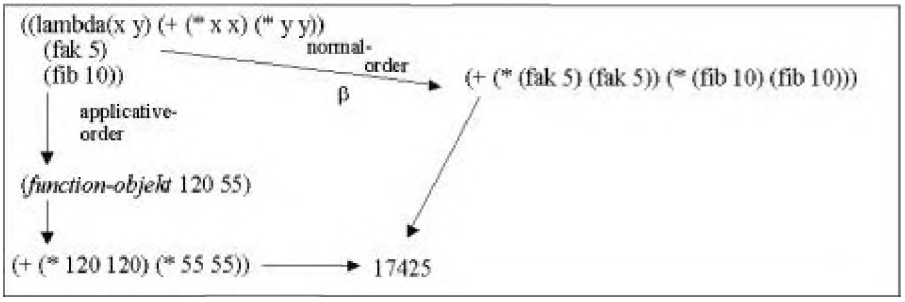

Figure 20.1 *Normal vs. applicative reduction of a Scheme expression*

As an immediate insight from the example we will emphasize the following:

> It appears to be the case that normal order reduction can lead to repeated evaluation of the same subexpression

20.8. Theoretical results

Lecture 5 - slide 16

We will now cite some theoretical results of great importance to the field.

> The theoretical results mentioned on this page assure some very satisfactory properties of functional programming

The results are based on a definition of *confluence*, which appears in the figure below.

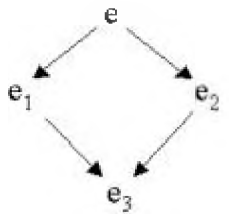

Figure 20.2 *The rewriting => is confluent if for all e, e_1 and e_2, for which e => e_1 and e => e_2, there exists an e_3 such that e_1 => e_3 and e_2 => e_3*

The results which we will use below are the following:

The first Church-Rosser theorem. Rewriting with beta and eta conversions are confluent.

The second Church-Rosser theorem. If e_0 => ... => e_1, and if e_1 is on normal form, then there exists a normal order reduction of e_0 to e_1

The practical consequences of the results will be discussed in the following section.

20.9. Practical implications

Lecture 5 - slide 17

We will here describe the practical consequences of the theoretical results mentioned on the previous page

- During the evaluation of an expression, it will never be necessary to backtrack the evaluation process in order to reach a normal form.
- An expression cannot be converted to two different normal forms (modulo alpha conversions, of course).
- If an expression *e* somehow can be reduced to *f* in one or more steps, *f* can be reached by normal order reduction - but not necessarily by applicative order reduction

Because rewriting with beta and eta reduction is confluent, according to the first Church-Rosser theorem in Section 20.8, we see that there can be no dead ends in an evaluation process. Assume there is, and you will get an immediate contradiction.

The middle item is of particular importance because it guaranties that a normal form is unique. Assume that two different normal forms exist, and get a contraction with the first of the theorems.

The last result is a direct consequence of the second Church-Rosser theorem. It says more or less that normal-order reduction is the most powerful evaluation strategy. Notice, however, the

efficiency penalties with are involved, due to repeated evaluation of expressions. This is the theme of Section 20.11.

We can summarize as follows.

Normal-order reduction is more powerful than the applicative-order reduction

Scheme and ML uses applicative-order reduction

Haskell is an example of a functional programming language with normal-order reduction

20.10. Conditionals and sequential boolean operators

Lecture 5 - slide 18

In languages with applicative-order reduction there is a need to control the evaluation process in order to avoid the traps of erroneous and infinite calculations. In this section we review a couple of widely used and important forms from Scheme and Lisp. The *evaluation control* of these should in particular be noticed.

There are functional language constructs - special forms - for which applicative order reduction would not make sense

- `(if b x y)`
  - Depending on the value of `b`, either `x` or `y` are evaluated
  - It would often be harmful to evaluate both `x` and `y` before the selection
    - `(define (fak n) (if (= n 0) 1 (* n (fak (- n 1)))))`

- `(and x y z)`
  - `and` evaluates its parameter from left to right
  - In case `x` is false, there is no need to evaluate `y` and `z`
  - Often, it would be harmful to evaluate `y` and `z`
    - `(and (not (= y 0)) (even? (quotient x y)))`

In the items above we discuss the general semantics of `if` and `and`. In the deepest items we give a concrete examples of `if` and `and` where the evaluation order matters.

20.11. Lazy evaluation

Lecture 5 - slide 19

Lazy evaluation is a particular implementation of normal-order reduction which takes care of the lurking multiple evaluations identified in Section 20.7.

We will now deal with a practical variant of normal-order reduction

Lazy evaluation is an implementation of normal-order reduction which avoids repeated calculation of subexpressions

In Figure 20.3 we show an evaluation idea which is based on normal-order reduction without multiple evaluation of parameters, which are used two or more times in the body of a function.

It is not our intention in this material to go deeper into the realization of an interpreter that supports lazy evaluation.

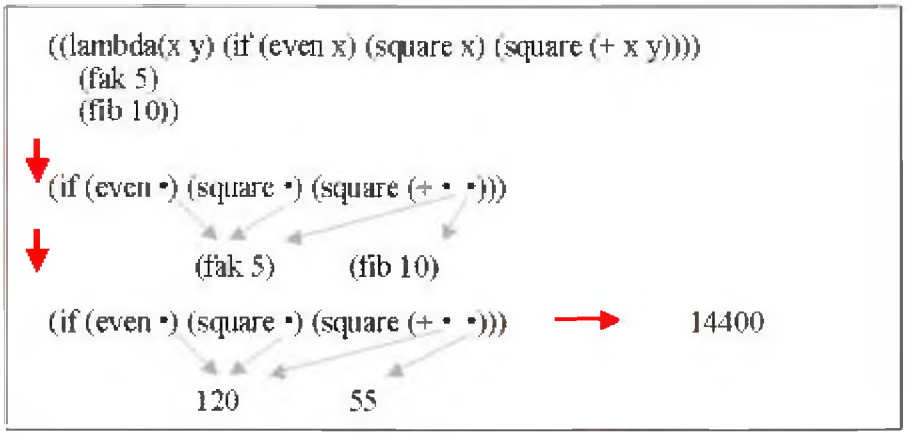

Figure 20.3 *An illustration of lazy evaluation of a Scheme expression. Notice, that Scheme does not evaluate the expression in this way. Scheme uses applicative-order reduction.*

This end the general coverage of evaluation order. In the next chapter we will see how to explore the insights from this chapter in Scheme, which is a language with traditional, applicative-order reduction.

20.12. References

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

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

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

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

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

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

[abelson96] Abelson, H., Sussman G.J. and Sussman J., *Structure and Interpretation of Computer Programs, second edition*. The MIT Press, 1996.

21. Delayed evaluation and infinite lists in Scheme

As noticed in Chapter 19 and Chapter 20 the evaluation strategy in Scheme is the one called *applicative order*, cf. Section 20.6.

In contrast to normal-order reduction and lazy evaluation - as described in Section 20.6 and Section 20.11 - we can think of Scheme as *eager* in the evaluation of the function parameters.

In this chapter we will see how to make use of a new evaluation idea in terms of explicitly delaying the evaluation of certain expressions. This is the topic of Section 21.1 and Section 21.2.

21.1. Delayed evaluation in Scheme

Lecture 5 - slide 21

The starting point of our discussion is now clear.

Scheme does not support normal-order reduction nor lazy evaluation

Scheme has an explicit primitive which delays an evaluation

The `delay` and `force` primitives are described in Syntax 21.1 and Syntax 21.2. The `delay` primitive returns a so-called *promise*, which can be redeemed by the `force` primitive. Thus, the composition of `delay` and `force` carry out a normal evaluation step.

```
(delay expr) => promise
```

Syntax 21.1

```
(force promise) => value
```

Syntax 21.2

In Program 21.1 we show simple implementations of `delay` and `force`. In Program 21.2 we show possible implementations of `delay` by means of Scheme macros.

```
(delay expr) ~ (lambda() expr)

(define (force promise) (promise))
```

Program 21.1 *A principled implementation of* `delay` *and* `force` *in Scheme.*

The thing to notice is the semantic idea behind the implementation of `delay`. The expression `(delay expr)` is equivalent to the expression `(lambda () expr)`. The first expression is supposed to replace the other expression at program source level. The value of the lambda expression is a closure, cf. Section 8.11, which captures free names in its context together with the syntactic form of `expr`. As it appears from the definition of the function `force` in Program 21.1 the promise returned by the `delay` form is redeemed by calling the parameter less function object. It is easy to see that this carries out the evaluation of `expr`.

Be sure to observe that `force` can be implemented by a function, whereas `delay` cannot. The reason is, of course, that we cannot allow a functional implementation of `delay` to evaluate the parameter of `delay`. The whole point of `delay` is to avoid such evaluation. This rules out an implementation of `delay` as a function. The `force` primitive, on the other hand, can be implemented by a function, because it works on the value of a lambda expression.

Please notice that other implementations of `delay` and `force` can easily be imagined. The Scheme Report describes language implementations of `delay` and `force`, which may use other means than described above to obtain the same semantic effect, cf. [delay-primitive] and [force-primitive].

```
; R5RS syntactic abstraction:
(define-syntax my-delay
  (syntax-rules ()
    ((delay expr)
     (lambda ()
        expr))))

; MzScheme syntactic abstraction:
(define-macro my-delay
  (lambda (expr)
    `(lambda () ,expr)))
```

Program 21.2 *Real implementations of delay. The first definition uses the R5RS macro facility, whereas the last one uses a more primitive macro facility, which happens to be supported in MzScheme.*

21.2. Examples of delayed evaluation

Lecture 5 - slide 22

Let us look at a few very simple examples of using `delay` and `force`. In the first line of the table below we delay the expression `(+ 5 6)`. The value is a promise that enables us to evaluate the sum when necessary, i.e, when we choose to force it. The next line shows that we cannot force a non-promise value. The last line shows an immediate forcing of the promise, which we bind to the name `delayed` in the `let` construct.

Expression	Value
`(delay (+ 5 6))`	`#<promise>`
`(force 11)`	*error*
`(let ((delayed (delay (+ 5 6)))) (force delayed))`	`11`

Table 21.1 *Examples of use of delay and force.*

21.3. Infinite lists in Scheme: Streams

Lecture 5 - slide 23

We are now done with the toy examples. It is time to use delayed evaluation in Scheme to something of real value. In this material we focus on *streams*. A stream is an infinite list. The inspiration to our coverage of streams comes directly from the book *Structure and Interpretation of Computer Programs* [Abelson98].

The crucial observation is the following.

> We can work with lists of infinite length by delaying the evaluation of every list tail using `delay`
>
> As an invariant, every list tail will be delayed

Every tail of a list is a promise. The promise covers an evaluation which gives a new cons cell, in which the tail contains another promise.

It is simple to define a vocabulary of stream functions. There is an obvious relationship between list functions (see Section 6.1) and the stream functions shown below in Program 21.3.

```
(cons-stream a b)    ~    (cons a (delay b))

(define head car)

(define (tail stream) (force (cdr stream)))


(define empty-stream? null?)

(define the-empty-stream '())
```

Program 21.3 *Stream primitives. Notice the way* `head` *is defined to be an alias of* `car`.

In that same way as we defined `delay` as a macro in Program 21.2 , we also need to define `cons-stream` as a macro. The reason is that we are not allowed to evaluate the second parameter; The second parameter of `cons-cell` is going to be delayed, and as such it must be passed unevaluated to `cons-stream`.

```
(define-macro cons-stream
  (lambda (a b)
    `(cons ,a (delay ,b))))
```

Program 21.4 *A MzScheme implementation of* `cons-stream`.

In the following sections we will study a number of interesting examples of streams from the numerical domain.

21.4. Example streams

Lecture 5 - slide 24

In the first example line in Table 21.2 we define a stream of ones. In other words, the name `ones` is bound to an infinite list of ones: `(1 1 1 ...)`.

Please notice the very direct use of recursion in the definition of ones. We are used to a conditional such as `cond` or `if` when we deal with recursion, in order to identify a basis case which stops the recursive evaluation process. We do not have such a construction here. The reason is that we never reach any basis (or terminating case) of the reduction. Due to the use of delayed evaluation we never attempt to expand the entire list. Instead, there is a promise in the end of the list which can deliver more elements if needed.

In the second row of the example we use the function `stream-section` to extract a certain prefix of the list (determined by the first parameter of `stream-section`). The function `stream-section` is defined in Program 21.5 together with another useful stream function called `add-streams` which adds elements of two numeric streams together.

In the third row we define a stream of all natural numbers, using the function `integers-starting-from`.

The fourth row shows an alternative definition of `nat-nums`. We use `add-streams` on `nat-nums` and `ones` to produce `nat-nums`. Please notice the recursion which is involved.

In the bottom row of the table we define the Fibonacci numbers, in a way similar to the definition of `nat-nums` just above. `fibs` is defined by adding `fibs` to its own tail. This works out because we provide enough staring numbers `(0 1)` to get the process started.

Expression	Value
`(define ones (cons-stream 1 ones))`	`(1 . #<promise>)`
`(stream-section 7 ones)`	`(1 1 1 1 1 1 1)`

`(define (integers-starting-from n)`
`  (cons-stream n`
`  (integers-starting-from (+ n 1))))`
 	
`(define nat-nums`
`  (integers-starting-from 1))`
 	
`(stream-section 10 nat-nums)`
	`(1 2 3 4 5 6 7 8 9 10)`

`(define nat-nums`
`  (cons-stream 1`
`  (add-streams ones nat-nums)))`
 	
`(stream-section 10 nat-nums)`
	`(1 2 3 4 5 6 7 8 9 10)`

`(define fibs`
`  (cons-stream 0`
`  (cons-stream 1`
`  (add-streams (tail fibs) fibs))))`
 	
`(stream-section 15 fibs)`
	`(0 1 1 2 3 5 8 13 21 34 55 89 144`
	`233 377)`

Table 21.2 *Examples of streams.* `ones` *is an infinite streams of the element 1.* `stream-section` *is a function that returns a finite section of a potentially infinite stream.* `nat-nums` *is stream of all the natural numbers, made by use of the recursive function* `integers-starting-from`*. The fourth row shows an alternative definition of* `nat-nums`*. Finally,* `fibs` *is the infinite stream of Fibonacci numbers.*

As mentioned above, the functions `stream-section` and `add-streams` in Program 21.5 are used in Table 21.2.

In the web version of the material (slide and annotated slide view) there is an additional program with all the necessary definitions which allow you to play with streams in MzScheme or DrScheme.

```
(define (stream-section n stream)
  (cond ((= n 0) '())
        (else
          (cons
            (head stream)
            (stream-section
             (- n 1)
             (tail stream))))))

(define (add-streams s1 s2)
 (let ((h1 (head s1))
       (h2 (head s2)))
   (cons-stream
     (+ h1 h2)
     (add-streams (tail s1) (tail s2)))))
```

Program 21.5 *The functions* `stream-section` *and* `add-streams`*.*

21.5. Stream example: The sieve of Eratosthenes

Lecture 5 - slide 25

Still with direct inspiration from the book *Structure and Interpretation of Computer Programs* [Abelson98] we will look at a slightly more complicated example, namely generation of the stream of prime numbers. This is an infinite list, because the set of prime numbers is not finite.

The algorithmic idea behind the generation of prime numbers, see Program 21.6 was originally conceived by Eratosthenes (a Greek mathematician, astronomer, and geographer who devised a map of the world and estimated the circumference of the earth and the distance to the moon and the sun - according to the American Heritage Dictionary of the English Language).

The input of the function `sieve` in Program 21.6 is the natural numbers starting from 2. See also the example in Table 21.3. The first element in the input is taken to be a prime number. Let us say the first such number is p. No number p*n, where n is a natural number greater than one, can then be a prime number. Program 21.6 sets up a sieve which disregards such numbers.

Recursively, the first number which comes out of the actual chain of sieves is a prime number, and it is used set up a new filter. This is due to the simple fact that the `sieve` function calls itself.

The Sieve of Eratosthenes is a more sophisticated example of the use of streams

```
(define (sieve stream)
   (cons-stream
     (head stream)
     (sieve
       (filter-stream
         (lambda (x) (not (divisible? x (head stream))))
         (tail stream)))))
```

Program 21.6 *The `sieve` stream function.*

Program 21.6 uses the functions `cons-stream`, `head` and `tail` from Program 21.3. The functions `filter-stream` and `divisible?` are defined in Program 21.7.

Figure Figure 21.1 shows a number of sieves, and it sketches the way the numbers `(2 3 4 ...)` are sieved. Notice that an infinite numbers of sieves are set up - on demand - when we in the end requests prime numbers.

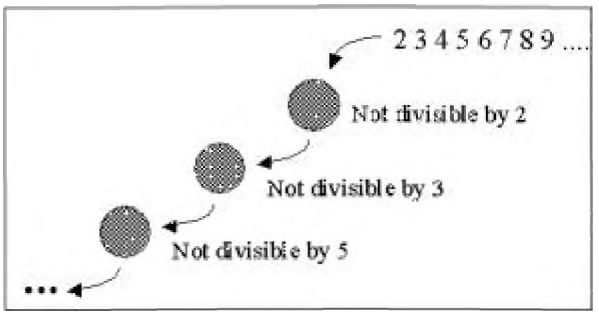

Figure 21.1 *An illustration of the generation of prime numbers in The Sieve of Eratosthenes*

21.6. Applications of The sieve of Eratosthenes

Lecture 5 - slide 26

In this section we show an example of prime number generation with the sieve function from Program 21.6.

Notice that the prime numbers are really generated on demand. In the call `(stream-section 25 primes)` we are requesting 25 prime numbers. This triggers generation of sufficient natural numbers via `(integers-starting-from 2)`, and it triggers the set up of sufficient sieves to produce the result.

We see that the evaluations are done *on demand*.

The sieve process produces the stream of all prime numbers

Expression	Value

`(define primes`
`  (sieve`
`  (integers-starting-from 2)))`
 	
`(stream-section 25 primes)`
	`(2 3 5 7 11 13 17 19 23 29 31 37 41`
	`43 47 53 59 61 67 71 73 79 83 89 97)`

Table 21.3 *The first 25 prime numbers made by sieving a sufficiently long prefix of the integers starting from 2.*

You can use the definitions in Program 21.7 to play with the `sieve` function. You should first load the stream stuff discussed in Section 21.4. More specifically, you should load the definitions on the last program clause in the slide view of Section 21.4. Then load the definitions in Program 21.7.

```
(define (sieve stream)
   (cons-stream
     (head stream)
     (sieve
       (filter-stream
         (lambda (x) (not (divisible? x (head stream))))
         (tail stream)))))

(define (divisible? x y)
  (= (remainder x y) 0))

(define (filter-stream p lst)
  (cond ((empty-stream? lst) the-empty-stream)
        ((p (head lst)) (cons-stream (head lst) (filter-stream p (tail lst))))
        (else (filter-stream p (tail lst)))))

(define (integers-starting-from n)
 (cons-stream n
  (integers-starting-from (+ n 1))))

(define primes (sieve (integers-starting-from 2)))
```

Program 21.7 *All the functions necessary to use the Sieve of Eratosthenes. In addition, however, you must load the Scheme stream stuff. The most remarkable function is* `filter-streams`*, which illustrates that it is necessary to rewrite all our classical higher order function to stream variants. This is clearly a drawback!*

21.7. References

[force-primitive] R5RS: force
http://www.cs.auc.dk/~normark/prog3-03/external-material/r5rs/r5rs-html/r5rs_63.html#SEC65

[delay-primitive] R5RS: delay
http://www.cs.auc.dk/~normark/prog3-03/external-material/r5rs/r5rs-html/r5rs_36.html#SEC38

[abelson98] Richard Kelsey, William Clinger and Jonathan Rees, "Revised^5 Report on the Algorithmic Language Scheme", *Higher-Order and Symbolic Computation*, Vol. 11, No. 1, August 1998, pp. 7--105.

22. Introduction to linguistic abstraction

This section if about linguistic abstraction in Scheme, with and without LAML. Linguistic abstraction is the act of making new languages. Thus, with linguistic abstraction we form new ways to express ourselves far beyond functional abstraction, as it has been studied until now.

22.1. Linguistic abstraction

Lecture 6 - slide 2

Linguistic abstraction can be defined very briefly, as follows.

Linguistic abstraction is the act of establishing a new language

In many contexts, linguistic abstraction is qualified with the word 'meta'. Thus we speak about metalinguistic abstraction, emphasizing that we enter a higher language level than the level we came from.

We introduce linguistic abstraction by comparing it with the more well-known discipline of making abstractions with functions (or for that sake, procedural abstraction).

- **Abstraction by means of functions**
  - Encapsulation, naming and parametrization of a potentially complex expression
  - Use of good abstractions makes a program easier to understand
  - Use of abstractions makes a program shorter, because the functions can be called from more than one context
- **Abstraction by means of languages**
  - Involves primitive means of expressions as well as the way the primitives can be meaningfully composed
  - A more global kind of abstraction than functional abstraction
  - Raises lexical, syntactical and - most important - semantical concerns
  - Specialized or general purpose language

Abstraction is a central discipline in all serious programming efforts. We have discussed functional abstraction in earlier parts of this material. Procedural abstraction is central in both the imperative paradigm and the object-oriented paradigm. Here we have contrasted functional abstraction with linguistic abstraction.

Problem solving by means of linguistic abstraction is a very powerful approach

The idea of defining and implementing a new language as part of a problem solving process is a very strong idea. Expert programmers tend to work in that way.

22.2. Linguistic abstraction in Lisp

Lecture 6 - slide 3

The primary language of interest in this material is Scheme, and with it the family of Lisp languages. Therefore we will at this point study linguistic abstraction in Lisp languages.

The are several possible approaches to linguistic abstractions in Lisp

Below we discuss an *incremental approach* to linguistic abstraction in Lisp. This approach is based on the point of view that definition of functions, procedures, and macros in Lisp contribute with new aspects of the Lisp language. This is discussed in additional detail in Section 22.3 and Section 22.4.

The contrast to an incremental approach will be called a *total approach*. As it appears from the items below, compilation and interpretation are considered as 'a total linguistic abstraction implementation technique'. Total linguistic abstraction is discussed in more details in Section 22.5 and Section 22.6.

- Incremental approaches
  - Each new construct is defined by a function or a macro
    - Macros are used for new surface syntax, and in cases where evaluation order issues prevent use of functions
  - *Fine grained linguistic abstraction*
- Total approaches
  - Writing an interpreter for the new language *or*
  - Translating the new language to an existing language (compilation)
  - *Coarse grained linguistic abstraction*

In some cases we *embed* the new language in an existing language, hereby combining the use of two or more languages in a single program or document

Language embedding is the issue of Chapter 23.

22.3. Fine grained linguistic abstraction in Lisp

Lecture 6 - slide 4

In this section, and in Section 22.4, we will discuss and give examples of fine grained linguistic abstraction in Lisp.

The main insight is that program contributions in terms of function definitions can be regarded as extensions of the Scheme language. The reason is that the status, use, and appearance of a new function is similar to both core language constructs and the pre-existing Scheme functions and procedures. In most other languages, there is a clear distinction of core language constructs and contributions made in terms of programs written in the language.

Due to the uniform notation of language constructs and functions, a set of Scheme functions can be seen as an extension of the Scheme language

- Incremental extension of the language
  - The functional paradigm is well-suited because application of functions can be nested into each other
  - The definitions of the functions implement the language
  - A possible approach:
    - the inner functions are more or less self evaluating
    - the outermost function is responsible for the realization of the language

A function call is an expression, which can be embedded into other function calls. In this way it is possible to build complex expressions by combination of programmed, functional abstractions.

As a contrast in the imperative programming paradigm, a procedure call is a command. A command can not normally be passed as a parameter to other commands. Thus, the combination of programmed abstractions is different, when we compare imperative and functional programming.

In section Section 22.4 we will see an example of the observations made above.

Programming in Lisp can be seen as incremental language development

22.4. An example of fine grained abstraction

Lecture 6 - slide 5

In this section we will study a simple formulation of a course home page, built by means of functions, and combined in the way discussed at the end of Section 22.3.

The `course-home-page` clause of Program 22.1 is to be processed somehow to create a set of course home pages. The functions used in Program 22.1 are defined in Program 22.2. As illustrated in Program 22.2 the subclauses of a `course-home-page` page are very simple, almost 'self-evaluating functions'. The `course-home-page` function itself is assumed to do the bulk part of the work - the real work so to say. This part of the program is only outlined in Program 22.2.

A fine grained implementation of a course home page language

Each of the forms in the language are implemented as a Scheme function

```
(course-home-page

  (name "Programming Paradigms")

  (number-of-lectures 15)

  (lecture-names
    "intr" "scheme" "higher-order-fn"
    "eval-order" "lisp-languages")

  (current-lecture 3)

  (links
    "schemers.org" "http://www.schemers.org/"
    "LAML" "http://www.cs.auc.dk/~normark/laml/"
    "Haskell" "http://haskell.org/"
  )
)
```

Program 22.1 *A sample document in a course home page language. The outer 'keyword' is* `course-home-page`. *Inside a* `course-home-page` *form there may be a number of subclauses. We see a* `name` *clause, a* `number-of-lectures` *clause etc. The important point of the example is that the expression is regarded as a clause in a new language, which we somehow want to implement with the purpose of 'solving some problem' - here to generate a set of coherent web pages for some activity.*

```
(define (course-home-page name-form number-form lecture-list current-form
                          link-form process-form)

  ; The real implementation of
  ; the course home page language

)



(define (name nm)
 (list 'name nm))

(define (number-of-lectures n)
 (list 'number-of-lectures n))

(define (lecture-names . name-lst)
 (cons 'lecture-names name-lst))

(define (current-lecture n)
 (list 'current-lecture n))

(define (links . lnk-list)
 (cons 'links lnk-list))
```

Program 22.2 *An almost empty outline of the functions that implement the course home page language. Each kind of subexpression is either implemented as a function or as a macro. In this simple example, macros are not used.*

The ideas in this section have been explored and developed in a LAML context. Early LAML languages, such as the 'Manual' language (for definition of library interface documentation) has been defined in the way illustrated above. More recent LAML-related language have been defined as XML-in-LAML language, and implemented as mirrors of an XML language in LAML. This approach is addressed in Chapter 24.

22.5. Coarse grained linguistic abstraction in Lisp

Lecture 6 - slide 6

As mentioned in Section 22.2 coarse grained linguistic abstraction is related to translation (compilation) and interpretation, as known from courses in compiler technology.

As a Scheme and Lisp topic related to transformation and interpretation, we notice on this page that parsing of an expression or program made in Lisp syntax (cf. parenthesized notation, Section 6.8), is very easy. The reason is that a generic parser can be written that translates a parenthesized string to a proper or improper list structure.

It is relatively easy and straightforward to establish a new language in Lisp syntax

- Establishing a new 'Lisp language'
  - Generic parsing can be done by the Lisp reader
  - It is possible to concentrate on the semantic issues
  - Language checking and error handling should not be forgotten

22.6. An example of coarse grained abstraction

Lecture 6 - slide 7

In this section we will discuss how to 'process' the course home page document (see Program 22.1), which we discussed earlier in Section 22.4.

Below we will assume that the course home page fragment of Program 22.1 is located on the file named "`new-document.lsp`".

In Program 22.3 we show how to open, read and close the file (in blue color). The processing of the parsed expression is shown in red color.

```
(let* ((port (open-input-file "new-document.lsp"))
       (new-document (read port))
      )

   ; new-document is a reference to the list structure
   ; representation of the new document.

   (process-document! new-document)

   (close-input-port port)
)
```

Program 22.3 *Reading the document as a list structure. We open a port to the document and use the read primitive in Scheme to read the list expression on the file. The procedure or function* `process-document` *is supposed to implement the processing semantics of the new language.*

In this material we will not go into any detail of the transformation. In Program 22.4 we limit ourselves to a superficial demo processing, in which we extract and print the keyword of each subform of the course home page form.

The important thing to notice is that it is very easy to come to the point where the semantic processing (as sketched in Program 22.4) can begin. The only preparation is that of Program 22.3.

```
(define (process-document! doc)
  (file-write (transform-document doc) "res.lsp"))

(define (transform-document doc)
  (let ((top-level-forms (document-forms doc)))
   (map
     (lambda (subform)
       (subform-keyword subform))
     top-level-forms)))


(define document-forms cdr)
(define subform-keyword car)
```

Program 22.4 *A simple demo processing of the document. We just extract some information about the document. No attempt is made here to implement the language, nor to process the document in any realistic way.*

In Program 22.5 we see a sample dialog and execution of the abstractions in Program 22.3 and Program 22.4.

```
1> (let* ((port (open-input-file "new-document.lsp"))
          (new-document (read port))
         )

   ; new-document is a reference to the list structure
   ; representation of the new document.

   (process-document new-document)

   (close-input-port port))

name
number-of-lectures
lecture-names
current-lecture
links
do-process
```

Program 22.5 *Execution dialogue. We show what happens when the document is read and processed in the simple manner programmed above.*

22.7. References

[course-plan-examples] Example of the LAML course home page system
http://www.cs.auc.dk/~normark/scheme/examples/course-plan-xml-in-laml/index.html

23. Language embedding

In Chapter 22 we discussed how to establish languages, especially in Lisp.

In this chapter we will discuss how to combine two (or more) existing languages. More specifically, we will look at ways to embed one language into another.

23.1. Embedded languages

Lecture 6 - slide 9

We start with a definition of language embedding.

> A new language N is an *embedded language* in an existing language E if an expression in N can be used as a subexpression of a construct in E.

As a possible practical organization of the embedding of a new language into another language, the interpreter of the new language N is made available as a function in the existing language E.

In the web domain there are many examples of language embeddings. Below we mention some of them.

- There are many examples of embedding web languages and programming languages
  - Embedding of CSS in HTML, SVG and other XML languages
  - Embedding of Javascript in HTML for client dynamics
  - Embedding of a programming language in HTML at the server-side
    - ASP: HTML embedding of Visual Basic fragments
    - JSP: HTML embedding of Java fragments
    - PHP: HTML embedding of C-like fragments
    - BRL: HTML embedding of Scheme fragments

23.2. Examples of language embedding in HTML

Lecture 6 - slide 10

In this section we will illustrate some examples of language embedding from the web domain. More specifically, we will see how fragments in various programming languages can be embedded into HTML. Such language embedding is widely used at the server side of the World Wide Web.

Concrete illustrations of JSP, ASP, and BRL documents

The ASP and JSP examples are available via the slide and the annotated slide view of this material. (The examples are too large to warrant inclusion at this location of the material). Please take a look at the web material for the details.

The BRL [Lewis00] example is included here, because it is relatively small. In some sense it also covers the essence of the two others. We see Scheme fragments (emphasized with red color) within a conventional HTML document. When the web document is delivered by the server, the program fragments are executed. The functional results of the program execution become part of the web document. In a nutshell, this is a very common way to deal with dynamic web contents.

```
<html>
 <head>
 [
  (inputs word) ; HTML input.  Will be null if no such input.
  (define newword
    (if (null? word)
        "something"
        word))
 ]
  <title>Backwards</title>
 </head>

 <body>
  <form>
  Type a word: <input name="word">
  <input type="Submit">
  </form>

  <p>[newword] spelled backwards is
     [(list->string (reverse (string->list newword)))]
  </p>

  <p>This message brought to you by [(cgi SERVER_NAME)] as a public
  service.</p>

 </body>
</html>
```

Program 23.1 *An example of a BRL document. BRL - Beautiful Report Language - is a Scheme based web server framework which allows the web programmer to embed Scheme fragments into HTML*

23.3. Course home page embedding in Scheme

Lecture 6 - slide 11

We will illustrate embedded languages with an embedded list-based language in Scheme. This is done as a direct continuation of the course home page example from Section 22.4.

The simple course home page language is an embedded *list language* in Scheme

In Program 23.2 we see the course home page expression, emphasized with red color. This is a list structure, formed in its own language: A simple course home page language. The language is list-based, and the non-constant parst of the language are brought in via quasiquotation (also known as backquoting). Thus, the course home page subdocument makes use of variables and expressions from the surrounding Scheme program. Notice, however, that a special interpreter is needed to process the backquoted `course-home-page` expression.

```
(let ((ttl "Programming Paradigms")
      (max 5)
      (current 3)
     )

 `(course-home-page

   (name ,ttl)

   (number-of-lectures ,max)

   ,(cons
     'lecture-names
     (map downcase-string
       (list "intr" "scheme" "HIGHER-ORDER-FN"
       "eval-order" "lisp-languages")))

   (current-lecture ,current)

   (links
     "schemers.org" "http://www.schemers.org/"
     "LAML" "http://www.cs.auc.dk/~normark/laml/"
     "Haskell" "http://haskell.org/"
   )
  )
)
```

Program 23.2 *A sample embedding of a course home document in a Scheme program. We use a quasiquotation to provide for a representation of the course home page as a list structure in the Scheme context.*

23.4. References

[-] BRL
http://brl.sourceforge.net/

[lewis00] Bruce R. Lewis, "BRL---A database-oriented language to embed in HTML and other markup", October 2000.

24. Language Mirroring

In this chapter we will discuss language mirroring, in part as a contrast to language embedding from Chapter 23.

24.1. Mirrored Languages

Lecture 6 - slide 13

Let us start with a definition of a mirrored language.

> A new language N is a *mirrored language* in an existing language E if an expression in N in a systematic way can be represented as an expression in E.

The mirror of N in E does not call for a new interpreter. A new interpreter as need for an embedded language i E. A mirror expression N-expr is written in E, and it can be evaluated by the processor (interpreter) of E.

- LAML provides mirrors of a number of XML languages in Scheme:
  - HTML 4.01 and XHTML1.0
  - SVG
  - A number educational languages, such as LENO and the Course Home Page language (Course Plan)

24.2. Course home page mirroring in Scheme (1)

Lecture 6 - slide 14

Let us now illustrate how to mirror the simple course home page language in Scheme. The mirror which we deal with is a mirror of an XML language in LAML. Recall that we programmed the course home page document with simple functional abstractions in Section 22.4 and that we embedded the course home page language in Scheme in Section 23.3. Thus, the treatment below is actually our third attempt to accommodate the simple course home page abstractions in Scheme.

> The simple course home page is mirrored as an XML language in Scheme and LAML

We will start by giving an overview of the practical process that leads to the creation of mirror of some XML language in Scheme and LAML.

- Steps involved in the mirroring process:
  - Write an XML DTD of the language
  - Parse the XML DTD to a Scheme data structure
  - Synthesize the mirror of the language by the XML-in-LAML mirror generation tool
  - When using the course home page language, load the mirror as a Scheme library

It is fairly straightforward to write an XML DTD for a new 'little language', although the SGML inherited language may seem a little strange at first sight. Take a look at Program 24.1.

```
<!ENTITY % Number "CDATA">
    <!-- one or more digits -->

<!ENTITY % URI "CDATA">
    <!-- a Uniform Resource Identifier, see [RFC2396] -->

<!ELEMENT course-home-page
  (lecture-names, links)
>

<!ATTLIST course-home-page
  name                  CDATA          "#REQUIRED"
  number-of-lectures    %Number;        "#REQUIRED"
  current-lecture       %Number;        "#IMPLIED"
>

<!ELEMENT lecture-names
  (lecture-name+)
>

<!ELEMENT lecture-name
  (#PCDATA)
>

<!ELEMENT links
  (link*)
>

<!ELEMENT link
  (#PCDATA)
>

<!ATTLIST link
  href              %URI;             "#REQUIRED"
>
```

Program 24.1 *The course home page DTD. The DTD is essentially a context free grammar of the new XML language. XML DTDs are a heritages from SGML (The Standard Generalized Markup Language).*

The XML DTD can be parsed with the LAML DTD parser. We usually make a simple LAML script for such purposes, as shown in Program 24.2.

```
(load (string-append laml-dir "laml.scm"))
(load (string-append laml-dir "tools/dtd-parser/dtd-parser-4.scm"))

(parse-dtd "course-home-page")
```

Program 24.2 *The script that parses the DTD.*

The DTD parser creates a Lisp list structure representation of the DTD. This list structure is passed as input to the LAML mirror generator. The LAML script in Program 24.3 shows how the mirror generator is activated.

```
(load (string-append laml-dir "laml.scm"))
(laml-tool-load "xml-in-laml/xml-in-laml.scm")

; ------------------------------------------------------------------------
; Tool parameters

; The name of the language for which we create a mirror
(define mirror-name "course-homepage")

; The full path to the parsed DTD:
(define parsed-dtd-path
  (in-startup-directory "course-home-page.lsp"))

; The full path of the mirror target directory
(define mirror-target-dir (string-append (startup-directory) "../mirror/"))

(define action-elements '(course-home-page))

(define default-xml-represent-white-space "#f")

(define auto-lib-loading "#t")


; End tool parameters
; ------------------------------------------------------------------------

(let ((mirror-destination-file
        (string-append mirror-target-dir mirror-name "-mirror" ".scm")))
  (generate-mirror parsed-dtd-path mirror-destination-file mirror-name))
```

Program 24.3 *The script that generates the mirror.*

The output of the mirror generator is a Scheme source file, which represents the mirror of the course home page language from Program 24.1. As most other automatically generated source files, the mirror library of the demonstrational course home language is not easy to read. We have therefore not included it in this version of the material. You can access it from the web version via the slide view.

24.3. Course home page mirroring in Scheme (2)

Lecture 6 - slide 15

In this section we will se how to use the mirror of the course home page language, which we created in Section 24.2.

A sample course home page document that uses the XML-in-LAML course home page mirror functions

```
(load (string-append laml-dir "laml.scm"))
(define (course-home-page! doc) 'nothing)
(load "../mirror/course-homepage-mirror.scm")

(let ((ttl "Programming Paradigms")
      (max 5)
      (current 3))

 (course-home-page  'name ttl 'number-of-lectures "5"
                    'current-lecture "3"
   (lecture-names
     (map
       (compose lecture-name  downcase-string)
       (list "intr" "scheme" "HIGHER-ORDER-FN"
             "eval-order" "lisp-languages")))
   (links
     (link  "schemers.org" 'href "http://www.schemers.org/")
     (link  "LAML" 'href "http://www.cs.auc.dk/~normark/laml/")
     (link  "Haskell" 'href "http://haskell.org/")
   )))
```

Program 24.4 *A sample course home page that uses the course home page mirror functions.*

The first three lines in Program 24.4 loads the laml library and the mirror library. Before loading the mirror library we need to define an *action procedure* of the top-level element, course-home-page. Notice that this element was announced as an action element in Program 24.3. As an action element, the action procedure takes over the rest of the transformation process, typically to HTML. In this demo setup, the action procedure is empty.

The mirror function applications in the course-home-page expression are all emphasized in red. Notice the smooth integration of the course home page mirror functions and other Scheme functions. You should in particular compare the way mapping is done with the similar mapping in Program 23.2.

Further processing and transformation is done by the *action procedure* `course-home-page!`

24.4. Course home pages ala Course Plan

Lecture 6 - slide 16

The course home pages of the Programming Paradigms is made by the Course Plan system. The principles used in the Course Plan system are the same as illustrated about for the toy course home pages.

A real life course home page mirror in Scheme - The Course Plan system

In the web version of this material there is a program that shows a real course home page from LAML. This is called a course plan. The example is too long for the paper version.

24.5. Embedding versus mirroring

Lecture 6 - slide 17

In this section we compare language embedding ala the example from Section 23.3 with language mirroring as discussed in this chapter.

How does a list-embedding of new language in Scheme compare to a mirroring of the language Scheme?

Embedding in Scheme	**Mirroring in Scheme**
New language fragments are represented as lists	New language fragments are represented as Scheme expressions
Many different interpretations can be provided for	The most typical transformation is 'built in', as obtained by evaluation of the Scheme expression
Processing requires a specialized interpreter	The (first level of) processing is done by the standard Scheme interpreter
Relatively awkward to combine with use of higher-order functions	Mixes well with higher-order functions

24.6. References

[course-plan-examples] The generated Course Plan page (web only)
http://www.cs.auc.dk/~normark/scheme/examples/course-plan-xml-in-laml/html/example.html

[transf] LAML transformation functions
http://www.cs.auc.dk/~normark/scheme/lib/xml-in-laml/man/xml-in-laml.html#SECTION18

25. Lisp in Lisp

Let us now jump to a topic, which is quite different from language embedding and language mirroring as discussed in Chapter 23 and Chapter 24. Recall, however, that the topic of the current lecture is linguistic abstraction, which is about establishing new languages in an existing language.

We will now see how to establish Lisp in Lisp. Thus, we will study a situation where the new language and the existing language are (almost) identical. This calls for a more detailed explanation and rationale, which is given in Section 25.1.

25.1. Why 'Lisp in Lisp'

Lecture 6 - slide 19

In this section we will look at a principled implementation of Lisp in Lisp. In concrete terms we will study a partial Scheme implementation in Scheme itself.

Why do we study an implementation of Scheme in Scheme?

- Motivations:
  - To illustrate the idea of linguistic abstraction in Lisp
    - Lisp is both the *implementation language* and the *new language*
  - To understand the overall principles of interpreters
  - To illustrate the use of important Lisp implementation concepts, such as environments
  - To provide a playground that provides for easy experimentation with the semantics of Scheme

We will refer to a concrete Scheme implementation from the book 'Structure and Interpretation of Computer Programs' (SICP).

We have earlier referred to the book 'Structure and Interpretation of Computer Programs' [Abelson96]. The part of the book which is relevant for linguistic abstraction and Scheme interpreters is chapter 4.

25.2. An overview of Scheme constructs

Lecture 6 - slide 20

When we are interested in implementing Scheme in Scheme it is important to have a good classification of constructs in Scheme.

As a basic distinction, some forms are denoted as *syntax*, and others as *procedures*. (In this particular context, 'procedures' also covers 'functions'). As another distinction, some abstractions are *fundamental* - they form the core language; Others are *library* abstractions in the sense that they can be implemented by use of the fundamental abstractions. You can consult section 1.3 of the Scheme report [Abelson98] to learn more about these distinctions.

What is the basic classification of constructs in Scheme?

- **Syntax**
  - Fundamental syntactical constructs such as `lambda` , `define` , and `if`
- **Primitive functions and procedures**
  - Fundamental functions and procedures, which cannot in a reasonable way be implemented in the language
- **Library Syntax**
  - Syntactical extensions which can be implemented by macros
- **Library functions and procedures**
  - Functions and procedures which can be implemented on the ground of more primitive features

Parenthesized prefix notation is used as a common notation for all kinds of constructs

This provides for an *uniform notation* across the different kinds of constructs in the language

25.3. Scheme in Scheme

Lecture 6 - slide 21

It is interesting and instructive to understand the most general processing primitive in a Scheme system, namely `eval`. Together with `apply` - which calls primitive functions, library functions, and your own functions - it is shown in Program 25.1.

It is possible to write a relatively full, but brief meta circular Scheme interpreter in Scheme

```
(define (eval exp env)
  (cond ((self-evaluating? exp) exp)
        ((quoted? exp) (text-of-quotation exp))
        ((variable? exp) (lookup-variable-value exp env))
        ((definition? exp) (eval-definition exp env))
        ((assignment? exp) (eval-assignment exp env))
        ((lambda? exp) (make-procedure exp env))
        ((conditional? exp) (eval-cond (clauses exp) env))
        ((application? exp)  (apply (eval (operator exp) env)
                                    (list-of-values (operands exp) env)))
        (else (error "Unknown expression type -- EVAL" exp))))


(define (apply procedure arguments)
  (cond ((primitive-procedure? procedure
           (apply-primitive-procedure procedure arguments)))
        ((compound-procedure? procedure)
           (eval-sequence (procedure-body procedure)
                          (extend-environment
                             (parameters procedure)
                             arguments
                             (procedure-environment procedure))))
        (else  (error "Unknown procedure type -- APPLY" procedure))))
```

Program 25.1 *The `eval` and `apply` functions (procedures) of the Scheme interpreters. The full interpreter needs a lot of relatively small helping functions (procedures) that we do not show here.*

The two central functions of the language implementation - `eval` and `apply` - are made available in the language itself

You are encouraged to read a much more comprehensive story about the Scheme in Scheme interpreter in chapter 4 of [Abelson96].

Below we will dwell a little on `eval` and `apply`, in the form they are available in Scheme.

25.4. The `eval` and `apply` primitives

Lecture 6 - slide 22

The `eval` procedure makes the Scheme interpreter directly available as a primitive in the language.

The `apply` procedure is handy when we call a function on a 'first class parameter list'; That is, in situations where the parameters are available in a list.

The implementation primitive `eval` of a Lisp systems is typically made available in the language, hereby providing access to evaluation of syntactical expressions (lists) in a given environment

The `apply` primitive is also available as a convenient mechanism for application of a function, in cases where all the parameters are available in a list

Examples of both are given in Table 25.1 below.

Expression	Value

`(let* ((ttl "My Document")`
`(bdy (list 'p "A paragraph"))`
`(doc`
`(list 'html`
`(list 'head`
`(list 'title ttl))`
`(list 'body bdy)))`
`)`
`(render (eval doc)))`
	`<html>`
	`<head>`
	`<title>My Document</title>`
	`</head>`
	`<body>`
	`<p>A paragraph</p>`
	`</body>`
	`</html>`

`(let* ((ttl "My Document")`
`(bdy (list 'p "A paragraph"))`
`(doc`
`` `(html ``
`(head (title ,ttl))`
`(body ,bdy))))`
`(render (eval doc)))`
	`<html>`
	`<head>`
	`<title>My Document</title>`
	`</head>`
	`<body>`
	`<p>A paragraph</p>`
	`</body>`
	`</html>`

`(+ 1 2 3 4)`	`10`
`(+ (list 1 2 3 4))`	`Error: + expects argument of type number;` `given (1 2 3 4)`
`(apply + (list 1 2 3 4))`	`10`

Table 25.1 *An illustration of `eval` and `apply`. In the first two rows we construct a list structure of the usual `html`, `head`, `title`, and `body` HTML mirror functions. In the first row, the `list` structure is made by the list function. In the second row, we use the convenient backquote (semiquote) facility. In both cases we get the same result. The last three rows illustrate the use of `apply`. `apply` is handy in the cases where the parameters of a function is already organized in a list. What it interesting in our current context, however, is that `apply` is really an implementation primitive of Scheme, which is made available in the language itself.*

With this we are done with Linguistic abstraction, and as such with the main lectures of this material.

The remaining chapters represent side tracks, in which we cover additional details about LAML, object-oriented programming in Scheme, and the imperative aspects of Scheme.

25.5. References

[-] R5RS: Apply
http://www.cs.auc.dk/~normark/prog3-03/external-material/r5rs/r5rs-html/r5rs_63.html

[-] R5RS: Eval
http://www.cs.auc.dk/~normark/prog3-03/external-material/r5rs/r5rs-html/r5rs_64.html#SEC66

[abelson98] Richard Kelsey, William Clinger and Jonathan Rees, "Revised^5 Report on the Algorithmic Language Scheme", *Higher-Order and Symbolic Computation*, Vol. 11, No. 1, August 1998, pp. 7--105.

[abelson96] Abelson, H., Sussman G.J. and Sussman J., *Structure and Interpretation of Computer Programs, second edition*. The MIT Press, 1996.

26. An introduction to LAML

LAML means 'Lisp Abstracted Markup Language'. LAML is a software package written by the author of this material. The purpose of LAML is to support web development in Scheme - both of static materials and of server-side applications.

In the first and the main part of this material we have used LAML whenever possible, to illustrate functional programming in Scheme with examples from the web domain. We are now ready for a more solid introduction and description of LAML.

26.1. HTML documents in LAML

Lecture 7 - slide 2

We start gently by taking a look at a very simple HTML document - see Program 26.1. The document in Program 26.1 can be accessed via [initdoc].

> We introduce LAML by studying the LAML counterpart of a very simple HTML document

```
<html>
   <head>
     <title>This is the document title</title>
   </head>
   <body>
      <h1>Document title</h1>
      <p>Here is the first paragraph of the document</p>
      <p>The second paragraph has an <em>emphasized item</em>
         and a <b>bold face item</b>.
      </p>
   </body>
</html>
```

Program 26.1 *A very simple HTML document that illustrates the overall HTML document structure. It may, however, be tedious to write these tags for each and every small HTML page you have to produce.*

The LAML counterpart of Program 26.1 is shown in Program 26.2. Each HTML element instance used in Program 26.1 is available in Scheme as a function. The set of such functions are called a *mirror of HTML in Scheme*. In Chapter 27 we will study the HTML mirror functions in details.

```
(html
 (head
  (title "This is the document title")
 )

 (body
  (h1 "Document title")

  (p "Here is the first paragraph of the document")

  (p "The second paragraph has an" (em "emphasized item")
     "and a" (b "bold face item") _ ".")
  )
)
```

Program 26.2 *The same document as an LAML expression. We see that the the shift from HTML to LAML is a matter of a few changes, some of which can be claimed to be of lexcical nature, and some of which are of concrete syntactical of nature. Notice the use of the underscore character, which suppresses the insertion of white space. This document cannot be processed immediately. However, the next version that we show can be processed.*

The final document on this page, Program 26.3, shows a complete LAML document. The red parts of the program make up the differences between Program 26.2 and Program 26.3. The first two lines of the red parts load the necessary software. The `write-html` clause renders the HTML document to text, and it writes the HTML document to a file in the file system.

```
(load (string-append laml-dir "laml.scm"))
(style "simple-html4.01-transitional-validating")

(write-html '(raw prolog epilog)
 (html
  (head
   (title "This is the document title")
   )

  (body
   (h1 "Document title")

   (p "Here is the first paragraph of the document")

   (p "The second paragraph has an" (em "emphasized item")
      "and a" (b "bold face item")_".")
   )
  )
)
```

Program 26.3 *The document from above embedded in the LAML framework. Besides initial loading, we see the imperative writing of the functionally generated document to a specific target file. If the source file is `doc1.laml`, the target file will be `doc1.html`.*

The document generated by Program 26.3 can be accessed via [lamldoc].

26.2. Authoring of web materials

Lecture 7 - slide 4

We leave the HTML mirror functions temporarily. We come back to them in Chapter 27. Here we will give an overview of ways to produce web materials. Our main purpose with this section is to put our approach - programmatic authoring - in perspective and in relation to more well-known and more commonly used techniques.

Several different approaches to web authoring can be identified

- **Writing the document in a markup language**
  - HTML - low level and non-extensible
  - XML - requires subsequent transformations or specification of the document rendering
- **Using a visual tool - a structure editor on top of the markup language**
  - Good for low skill users
  - Difficult to manage large and complex materials
- **Transforming the document from another format**
  - Will often result in a web edition of a paper document
  - Difficult to make effective use of the WWW's hypertext potentials
- **Writing the document in a programming language**
  - Potentially good for users with programming skills
  - To be explored in this lecture

In the next section we define and describe the last approach - programmatic authoring - in more details.

26.3. Programmatic authoring

Lecture 7 - slide 5

Programmatic authoring is the idea behind the use of LAML for creation of static web pages.

Using *programmatic authoring* the document source is a program written in a programming language. By executing the program, the document source program is transformed to another format, typically HTML.

The generation of HTML upon execution of a programmatically authored document can be questioned. It can be argued that several possible kinds of processings and transformation can be imagined. Why bind the program execution to a single kind of processing?

In LAML we have more recently prepared for an intermediate document representation, which is produced when a document is processed. This is part of the XML-in-LAML approach, which we touch on in Section 28.4. This document representation is akin to abstract syntax trees. Thus, when

evaluating an expression such as the one in Program 26.2 in a context where all the mirror functions and defined, an abstract syntax is returned. This intermediate representation can be rendered as textual HTML, or it can be processed in another direction.

- Expected advantages of programmatic authoring
  - We can use all the *'programming tricks'* in the web authoring area
    - Authoring of complex materials parallels creation of non-trivial programs
  - Programmatic authoring is probably not feasible in many main stream languages
    - Java, C++, C, Pascal, Perl, ...

Using programmatic authoring the power of a programming language is available *everywhere* in the document, and at *any time* in the authoring process

26.4. LAML: Lisp Abstracted Markup Language

Lecture 7 - slide 7

We now introduce the LAML system, which basically is Scheme with access to libraries which mirror HTML. In addition, we support a number of functions which in our experience are very helpful during a typical web authoring process. Some of these functions are organized in document styles, others in tools of various kinds.

LAML means Lisp Abstracted Markup Language

LAML provides abstractions, in terms of functions, for HTML. Beyond these it is possible to create arbitrary abstractions, along the line of XML.

- LAML fundamentals:
  - The Scheme programming language itself
  - Mirrors of HTML and XML languages in Scheme
  - A number of useful libraries
  - A number of LAML document styles
  - A number of LAML tools
  - An Emacs Environment for practical use of LAML

Briefly stated, but also somewhat simplified, we can summarize LAML as follows.

LAML = Scheme + The HTML and XML mirrors

26.5. References

[lamldoc] The LAML generated document as rendered in a browser
http://www.cs.auc.dk/~normark/prog3-03/html/notes/external-html/doc1.html

[initdoc] The initial HTML document as rendered in a browser
http://www.cs.auc.dk/~normark/prog3-03/html/notes/external-html/initial-document.html

27. HTML mirror functions in LAML

In this chapter we will elaborate on the HTML mirror functions, which we have encountered numerous times during the initial parts of this material.

The mirror functions are generated automatically from a document type definition of HTML.

The apparatus for generation of HTML mirrors is also available for XML languages. This is XML-in-LAML. We have used XML-in-LAML for several non-trivial tasks. The source of this material is authored in an XML-in-LAML language called LENO.

27.1. The HTML mirrors in LAML

Lecture 7 - slide 9

We will here describe the HTML4.01 transitional and the XHTML mirrors in LAML. All the essential observations also hold for XML-in-LAML languages. The XHTML mirrors are made via the XML-in-LAML framework.

A *HTML mirror in Scheme* is a set of functions that in an accurate way makes the HTML elements of a particular HTML version available in Scheme

The key properties of a HTML mirror in Scheme - as provided by LAML - are the following.

- A one-to-one mapping of HTML elements to named functions in Scheme
- Generates **well-formed** and **valid** HTML documents
- Prevents accidental emission of '<', '>', and '&' as part of the textual contents
- The mirror functions return abstract syntax trees which can be rendered as 'HTML text'
- Supports optional pretty printing of the resulting HTML code

The generation of valid HTML in reality boils down to check of the context free composition of the document on the fly, while it is generated by the mirror functions.

As of the fall of 2003, the generation of the validation procedures is fully automatic. XML elements with so-called *element contents* [xml10] (as opposed to the elements with *mixed content*, i.e. elements with alternatives and PCDATA) are validated by deterministic finite state automata.

The point below is rather naive, but nevertheless important.

Not too many functions - not too few. You cannot by accident use a non-standard HTML element

Using raw XML you can by accident use a non-existing tag or element. When used through LAML you will discover this as soon as you attempt to execute the LAML document - simply because the counterpart of the non-existing tag is a non-defined Scheme function.

In Section 27.2 and the subsequent sections we will describe and discuss the detailed rules of the mirror functions. The rules will be summarized in Section 27.6.

27.2. Mirroring of HTML (1)

Lecture 7 - slide 10

We describe the mirror functions in this and the following sections.

We here focus on the basic elements and their composition. As a contrast to Program 26.2 from the previous chapter, we also include the attributes in the forms below

> The mirror function `f` distinguishes between *attribute names*, *attribute values*, *explicit white space*,*character references*,and *content strings* via the runtime types of the parameters together with their mutual position in the parameter list.

```
(f 'a1 "v1" 'a2 "v2" "Some text." "More text")
```

Program 27.1 *A HTML mirror expression with attribute names (symbols), attribute values (strings following symbols) and content strings.*

The LAML Scheme form in Program 27.1 corresponds to the HTML form in Program 27.2.

```
<f a1="v1" a2="v2"> Some text. More text</f>
```

Program 27.2 *The rendering of the value of the HTML mirror expression.*

Attribute names are represented as symbols. The string that follows a symbol is the attribute value. All other strings are 'white space concatenated' to the element instance contents. The details of the white space handling is described in Section 27.3.

As a consequence, and as a generalization in relation to HTML and XML, the attributes and the contents may be mixed arbitrarily. As an example, all the attributes may come after the content strings.

The HTML mirror functions traverse the actual parameters and 'sorts' them into attributes and contents contributions. From a programming language point of view, the attributes are handled as keyword parameters - although simulated with use of symbols as explained above.

27.3. Mirroring of HTML (2)

Lecture 7 - slide 11

The handling of white spaces is a minor detail. Nevertheless, it is a detail which is important to 'get right'.

The basic idea behind the handling of white spaces in LAML is formulated in the following point.

> Instead of specifying where to add white spaces we tell where to suppress it

The underscore symbols (shown with red emphasis) in Program 27.3 suppress white space. Underlying, the underscore symbol is bound to a boolean false value. This is rather arbitrary. What matters is really that a distinguished and unique run-time value is used for the purpose. Symbols and strings are ruled out, because they are used already for content and attributes. A boolean value is fine.

```
(p "Use" (kbd "HTML") _ "," (kbd "XHTML") _ ","
        (kbd "XML")_ "," "or" (kbd "LAML") _ ".")
```

Program 27.3 *An HTML mirror expression which suppresses white space in front of punctuation characters.*

The Scheme fragment in Program 27.3 returns an internal structure, which can be rendered as shown in Program 27.4.

```
<p>
  Use <kbd>HTML</kbd>, <kbd>XHTML</kbd>, <kbd>XML</kbd>, or <kbd>LAML</kbd>.
</p>
```

Program 27.4 *The rendering of the value of the HTML mirror expression.*

27.4. Mirroring of HTML (3)

Lecture 7 - slide 12

Until now we have met symbols, strings and boolean values as parameters in the HTML mirror functions. We have not yet used lists. We will do that now.

When a list is encountered among the parameters of a HTML mirror function, the elements of the list are spliced into the surrounding parameter lists. The implementation of the splicing is recursive, such that lists of any depth are flattened and spliced in the contextual list of HTML mirror function parameters.

It turns out that this handling of this is extremely flexible and important in a language, where the primary structuring of data is done with lists.

List of contents and lists of attributes are processed recursively and spliced together with their context

In Program 27.5 we see an `ul` unordered list, in which the contents is passed as a list. The list is formed by mapping the `li` mirror function on `(list( "one" "two" "three")`.

```
(body

  (ul
    (map li (list "one" "two" "three"))
  )


  (let ((attributes
          (list 'start "3" 'css:list-style-type "lower-roman"))
        (contents (map li (list "one" "two" "three"))))
    (ol 'id "demo" contents attributes)))
```

Program 27.5 *An HTML mirror expression in which lists are passed as parameters to the HTML mirror functions.*

With the convention of splicing lists into its surround, the example would be more clumsy too, because we then have to apply `ul` on a certain list, using `apply`, cf. Section 25.4.

In the `let` construct of Program 27.5 we bind the blue names `attributes` and `contents` to two lists. Both lists are spliced into the list with the `id` symbol and the "demo" string. In the `ol` form we just refer to these lists via their names. Notice that the `id` attribute of `ol` is passed 'the normal way'.

```
<body>
    <ul><li>one</li> <li>two</li> <li>three</li></ul>

    <ol style="list-style-type: lower-roman;" id="demo" start="3">
        <li>one</li>
        <li>two</li>
        <li>three</li>
    </ol>
</body>
```

Program 27.6 *The rendering of the value of the HTML mirror expression.*

27.5. Mirroring of HTML (4)

Lecture 7 - slide 13

In this section we show how CSS attributes - Cascading Style Sheet attributes - are handled in the HTML mirror functions.

CSS attributes and HTML attributes are uniformly specified

CSS attributes are prefixed with `css:`

```
(em 'css:background-color "yellow" "Functional Programming in Scheme")
```

Program 27.7 *An HTML mirror form in which we highlight a CSS attribute.*

The main point to notice is the uniform notation used for both HTML attributes and CSS attributes. The notation is somehow in conflict with the notation used for name spaces in XML.

The rendering of the expression in Program 27.7 is shown in Program 27.8.

```
<em style="background-color: yellow;">Functional Programming in Scheme</em>
```

Program 27.8 *The rendering of the value of the HTML mirror expression.*

HTML attributes are validated in LAML, but CSS attributes are not.

27.6. Summary of Mirror Rules

Lecture 7 - slide 14

The parameter passing rules of the HTML mirror functions are summarized below. Notice that the exact same rules apply for XML mirror functions in the so-called XML-in-LAML framework.

The HTML mirror conventions of LAML can be summarized in six rules

- **Rule 1**
  An attribute name is a symbol in Scheme, which must be followed by an expression of type string, which plays the role as the attribute's value.
- **Rule 2**
  Parameters which do not follow a symbol are content elements (strings or instances of elements).
- **Rule 3**
  All content elements are implicitly separated by white space.
- **Rule 4**
  A distinguished data object (the boolean value false) which we conveniently bind to a variable named _ suppresses white space at the location where the value appears.
- **Rule 5**
  Every place an attribute or a content element is accepted we also accept a list, the elements of which are processed recursively and unfolded into the result.
- **Rule 6**
  An attribute with the name **css:** ***a*** refers to the ***a*** attribute in CSS.

In the next chapter we will discuss a number of practical aspects of the LAML system.

27.7. References

[xml10] World Wide Web Consortium, "Extensible Markup Language (XML) 1.0", February 1998.

28. Additional LAML topics

In this chapter we describe some more practical aspects of using LAML. We also touch on the XML-in-LAML approach.

28.1. LAML document processing

Lecture 7 - slide 16

As one of the underlying ideas of LAML, we organize the programmatic source of a web document as a Scheme program. When the Scheme program is executed, the underlying HTML files are generated.

It is of practical importance to make it easy and flexible to execute the Scheme program. Below we summarize the possibilities supported in the LAML system.

The LAML system supports a number of different ways to process a document

- **From the command prompt (shell)**
  - Good for some Unix users
  - Good for tool composition - piping
- **From a Scheme prompt (LAML prompt)**
  - Makes it possible to interact with LAML at a more fine grained level
- **From Emacs**
  - Direct support of LAML from an advanced text editor
  - Synchronous and asynchronous processing
  - Keyboard shortcuts, templates, and menu support
  - Support of embedding of a substring in a Lisp form - and the inverse unembedding

Let us be more concrete. Scheme source files with LAML documents are supposed to have the file extension 'laml'. We will assume that we have written source document in a file called 'd.laml'.

From a command prompt or a Unix shell, you can execute it and hereby generate the underlying HTML file by the command `laml d`.

From a Scheme interpreter (a Scheme prompt) in which the basic LAML software has been loaded, you can type `(laml "d")` followed by "enter". This is a call of the `laml` procedure on "`d`".

From within Emacs (in which the LAML mode has been loaded) you can process a the document directly, by issuing the command **M-x laml-process-current-buffer**. (It is assumed, however, that the buffer is associated with a file at some location in the file system). The emacs command `laml-process-current-buffer` is bound to some convenient shortcut, `C-o`, which makes it very easy to do the processing.

In Emacs, it is possible to start an interactive Scheme/LAML session. This is done with **M-x run-laml-interactively**. Using this possibility, the Scheme system is started, and selected LAML libraries are loaded. The Emacs Lisp variables `interactive-laml-mirror-library` and `interactive-laml-load-convenience-library` control which HTML mirror library and which additional support libraries to load in the interactive LAML session.

> The Emacs support of LAML can be seen as powerful *environment* for programmatic authoring

As a practical remark, we will strongly recommend that LAML is used via Emacs. Authors of LAML-based materials should make use the features of the Emacs laml mode. Similarly, users who interact with the Scheme system can make good use of the additional support in Emacs, such as the `run-laml-interactively` command mentioned above, and the variables that control the behavior of this command.

28.2. LAML document styles and tools

Lecture 7 - slide 17

The purpose of this section is to give a short overview of some of the major document styles and tools in LAML. A document style is a LAML based language. All such new languages are derived from an XML document type definition.

The overview below is by no means comprehensive. You are referred to the LAML Software home page [normark02e] for details.

> A number of document styles and tools have been built on top of the basic LAML libraries and the mirrors

- Document styles
  - *Domain specific web languages*
  - LENO in which this material is written
  - Course plan in which the course home page is written
  - The Manual document style in which the LAML software is documented
  - ...
- Tools
  - The Scheme Elucidator used to explain programs in this material
  - Web Calendar used on the course home page
  - XML parser and XML mirror generation tool
  - ...

28.3. LAML server-side programming - CGI

Lecture 7 - slide 18

LAML can be used for authoring of static web material using a Scheme mirror of an XML language. LAML has also been used for server side programming using the Common Gateway Interface - CGI.

In this material we will not cover CGI programming in Scheme, but we will refer to an elucidative tutorial on the topic.

In the web version of this material there is an 'Elucidate' button which brings you to a Scheme elucidator that explains how to make a LAML CGI program, cf.[laml-cgi-tutorial].

A number of non-trivial server side web systems have been made in LAML via use of CGI

- LAML Web System
  - IDAFUS - a distance education manager used for CS open university activities during three years
  - The calendar manager
  - The exercise manager
  - ...

IDAFUS - ***I**nsitut for **DA**talogis **F**jern**U**ndervisnings **S**ystem* - is a major example of a LAML server application used for non-trivial purposes during a number of years in open university educations. The exercise manager is a minor system for synchronous management of exercise sessions. It requires user name and password to try out these systems.

The LAML calendar manager is widely used at Aalborg University for course and semester calendar management. The author's (simple) calendar can be accessed as an example [kn-calendar]. From a given calendar you can create your own, if you want.

28.4. XML in LAML

Lecture 7 - slide 20

XML-in-LAML is the LAML framework for mirroring of XML languages in Scheme.

In LAML there is a DTD parser, which produces a Lisp representation of a DTD. Based on this representation it is possible to synthesize an exact Scheme mirror of the XML language. The properties of this mirror is described below.

XML-in-LAML is a mirroring technique that makes XML languages available in Scheme

- **XML-in-LAML**
  - Mirror derivation from Document Type Definitions (DTDs)
  - Automatic derivation of validation predicates
  - Two or more XML languages can co-exist
- **Existing XML-in-LAML languages**
  - XHTML (strict, transitional, frameset) with full validation
  - SVG - Scalable Vector Graphics - with partial validation
  - LENO - the XML language used as the source of this material - with full validation
  - Course Plan - the course home page system.
  - Program Dissection - A simple program explanation facility
  - Photo Show - A web photo presentation system

Currently we convert several existing LAML document styles to the XML-in-LAML framework. As an important language for this material - LENO has already been transformed.

28.5. More information

Lecture 7 - slide 22

There are a number of sources to more information about LAML

- **Academic**
  - The papers available from the LAML Home Page
- **Tutorial**
  - The elucidative LAML Tutorial
- **LAML user/programmer information**
  - The LAML software home page

The LAML home page [laml-home] contains all the papers that have been written above LAML and LENO.

The aim of the LAML tutorial [laml-tutorial] is to introduce various aspects of LAML in a gentle way.

The LAML software home page [laml-software] contains an overview of all the software, and it gives access to a download page where the latest distribution is made available as free software.

LAML can be downloaded as free software from the LAML home page

The exercise below is seen as a typical, practically oriented example where LAML can be used for management of everyday information.

Exercise 7.1. *Bookmark administration*

Exercise motivation: It is not practical to bind web bookmarks to a single machine nor to a single browser. Therefore we will maintain a list of bookmark entries in a file, and generate web bookmark pages from this description.

We will assume that we maintain a *list* of bookmark entries, each of the form

(`bookmark` *URL title category comment*)

The first constituent is a symbol (a tag that distinguishes bookmark entries from other structured data) and the other fields are text strings.

You can find an example of a bookmark list here. See the link in the web version.

The exercise is to complete a frame-based bookmark browser, given a list of bookmarks. You can find an example of a frame-based bookmark browser here. See again the link in the web version. To make it realistic for you to solve this exercises, you are only asked to make the left frame: the category overview frame. The frameset page and the right frame page are already programmed.

You are, however, supposed to understand the pre-programmed frame of the exercise. So start to read through the existing pieces of the program.

You can find the pre-programmed part of the bookmark browser in the zip file bookmarks-zip.zip. Please consult the web version of the material to get access to it. Unzip it, and LAML process the file `bookmark.laml`. Bring up the file `bookmark-frameset.html` in a browser to get started.

You are welcome to extend the solution, for example with a good use of the comment field from the bookmark entry.

28.6. References

[laml-software] The LAML software home page - development version
http://www.cs.auc.dk/~normark/scheme/index.html

[laml-tutorial] The LAML tutorial
http://www.cs.auc.dk/~normark/scheme/tutorial/index.html

[laml-home] The LAML home page
http://www.cs.auc.dk/~normark/laml/

[laml-cgi-tutorial] Guess a Number - A simple CGI program in Scheme with LAML
http://www.cs.auc.dk/~normark/scheme/tutorial/cgi-programming/cgi-programming.html

[kn-calendar] The authors personal LAML calendar
http://www.cs.auc.dk/~normark/cgi-bin/calendar/data/normark.html

[normark02e] Kurt Nørmark, "The LAML Software Home Page", 2003.

29. Classes and objects in Scheme

In Section 8.5 we have described function-objects, which are returned when we evaluate lambda expressions. Function objects are also called *closures*, because a function object captures and 'closes around' the free name, on which the function object depends (due to the use of static binding of free names).

In this section we will see how to combine function objects to represent objects, in the sense of the object-oriented programming paradigm.

29.1. Functions with private context

Lecture 8 - slide 4

We start gently with some observations about private context around functions. It turns out later that this is one of the key ideas when we want to represent objects by means of functions. The insight in this section shows how to arrange some private, encapsulated state around a function. This is relevant because the idea of *encapsulation* is central in the object-oriented paradigm.

It is possible to define a private context around a function

The function below is defined in a private context. The `let` construct sets up a number of names (not shown directly in the program), which can be accessed from the lambda expression. Moreover, no other places than the lambda expression can access these names. Therefore the context is private to the shown lambda expression.

```
(define function-with-private-context
 (let (CONTEXT)
   (lambda (PARAMETERS)
     BODY)))
```

Program 29.1 *A template of a function with private context. The lambda expression appears in the context of the `let` name binding construct. When the definition is evaluated a name binding context is established around the lambda expression. The lambda expression is the only place in the program which have a possibility to reach the name bindings. Therefore the name bindings are local to the lambda expression. The CONTEXT is a list of name bindings, as such name bindings appear in a `let` construct.*

Before we proceed with a deeper understanding and exploration of private context around functions, we will give a concrete example in Table 29.1 below. We take one of our favorite examples, namely the use of the HTML mirror functions `html`, `head`, `body`, etc, to define a document function. The document function has a private context, in which we redefine `html` and `body` . The redefinition binds a couple of relevant attributes, by use of the function `modify-element`, which we introduced in Section 18.5.

Expression	Value
(define document (let ((html (xml-modify-element html 'xmlns "http://www.w3.org/1999/xhtml")) (body (xml-modify-element body 'bgcolor (rgb-color-encoding 255 0 0)))) (lambda (ttl bdy) (html (head (title ttl)) (body bdy)))))	
(document "A title" "A body")	<html xmlns = "http://www.w3.org/1999/xhtml"> <head> <title>A title</title> </head> <body bgcolor = "#ff0000"> A body </body> </html>

Table 29.1 *An example in which we abstract a `(html (head (title..)) (body ...))` expression in a lambda expression, of which we define a private context. The context redefines the `html` and `body` functions to 'specialized' versions, with certain attributes. Notice the importance of the 'simultaneous name bindings' of `let` in the example (as explained in an earlier lecture). Also notice that we have discussed the `modify-element` higher-order function before.*

Now we know how to deal with private context around a function. In the next section we will use this knowledge to approach the definition of classes, as known from the object-oriented programming paradigm.

29.2. Classes and objects

Lecture 8 - slide 5

We will now demonstrate that a function definition can be interpreted as a class, and that a function call can play the role of an object. In other words, certain lambda expressions will be regarded as classes, and certain closures will be seen as objects.

Due to (1) the **first class** status of functions, and due to (2) the use of **static binding of free names**, it is possible to interpret a closure as an *object*

With this interpretation, it is possible to regard certain function definitions as *classes*

In Program 29.2 we see the definition of `Point`, which we want to play the role of a class. The purple lambda expression, the value of which is returned from `Point`, is the *object handle*, which represents the object. Notice that this object handle really is a dispatcher, which returns one of the red methods given a message parameter as input. Thus, the object handle manages *method lookup* in the object. The `letrec` construct organizes the methods, which all are defined inside the scope of the green instance variables, which are just the 'construction parameters' of the point.

```
(define (point x y)
  (letrec ((getx     (lambda () x))
           (gety     (lambda () y))
           (add      (lambda (p)
                       (point
                        (+ x (send 'getx p))
                        (+ y (send 'gety p)))))
           (type-of (lambda () 'point))
          )
    (lambda (message)
      (cond ((eq? message 'getx) getx)
            ((eq? message 'gety) gety)
            ((eq? message 'add)  add)
            ((eq? message 'type-of) type-of)
            (else (error "Message not understood"))))))
```

Program 29.2 *The definition of a 'class `Point`' with methods `getx`, `gety`, `add`, and `type-of`. On this page we have also defined the syntactical convenience function `send` that sends a message to an object. In MzScheme, be sure that you define `send` before `Point` (such that `send` in the `add` method refers to our `send`, and not an already existing and unrelated definition of the name `send`).*

In the `add` method we use the `send` function. The `send` function sends a message to an object. We show the definition of the `send` function in Program 29.3. The `send` function just looks up the method, after which it calls the method by means of `apply`.

```
(define (send message obj . par)
  (let ((method (obj message)))
    (apply method par)))
```

Program 29.3 *The `send` method which is used in the `Point` class. The function `apply` calls a function on a list of parameters. This should be seen in contrast to a normal call, in which the individual parameters are passed.*

The `send` function is of course of interest also outside the classes. Whenever we need to communicate with an object, we do it by use of the `send` function. We will encounter `send` and other similar functions in the following sections.

On the practical side, it might be a good idea to rename `send`, for instance to `send-message`. It is likely that `send` already is the name of a function in your Scheme system. In DrScheme and MzScheme this is indeed the case. So in order to avoid problems with this, you can consider a systematic renaming when you are playing with the `Point` class in Exercise 8.1.

Exercise 8.1. *Points and Rectangle*

The purpose of this exercise is to strengthen your understanding of functions used as classes in Scheme.

First, play with the existing `Point` class defined on this page available from the on-line version of this material.

As an example, construct two points and add them together. Also, construct two lists of each four points, and add them together pair by pair.

Define a new method in the `Point` class called `(move dx dy)`, which displaces a point with `dx` units in the x direction and `dy` units in the y direction. We encourage you to make a functional solution in which `move` creates a new displaced point. After that you can make an imperative solution in which the state of the receiving point is changed.

Finally, define a new class, `Rectangle`, which aggregates two points to a representation of a rectangle. Define `move` and `area` methods in the new class.

As a practical remark to the 'class `Point`' and the `send` primitive, be sure to define `send` before you define `Point`. (This is done to redefine an existing `send` procedure in MzScheme).

29.3. A general pattern of classes

Lecture 8 - slide 6

We will now generalize the ideas exemplified in the `Point` class above. With this we will discuss a general pattern for simulation of classes and objects in Scheme.

If you want additional information about the simulation of object-oriented concepts in Scheme you can consult [Normark90a].

The following shows a template of a function that serves as a class

In the program below pattern of a class is shown. As explained in Program 29.4 there are construction parameters, instance variables, methods, and the `self` function. The methods and the `self` function are defined through explicit `define` forms. This is just syntactic sugar, instead of using `letrec` as in `Point` - see Program 29.2.

```
(define (class-name construction-parameters)
 (let ((instance-var init-value)
       ...)

   (define (method parameter-list)
     method-body)

   ...

   (define (self message)
     (cond ((eqv? message selector) method)
           ...

           (else (error "Undefined message" message))))

   self))
```

Program 29.4 *A general template of a simulated class. `construction-parameters` are typically transferred to the `let` construct, which we want to play the role as instance variables. Next comes explicitly defined methods, and last is the object handle called `self`. Notice that the value returned by the class is the value of `self` - the object handle.*

Below we show the `send` function, which we also saw in Program 29.3. The version in Program 29.5 includes a little error handling in addition to method lookup and method activation. The program also contains a syntactic sugaring function called `new-instance` , which just 'calls the class' with the purpose of class instantiation.

```
(define (new-instance class . parameters)
  (apply class parameters))

(define (send message object . args)
  (let ((method (object message)))
    (cond ((procedure? method) (apply method args))
          (else (error "Error in method lookup " method)))))
```

Program 29.5 *Accompanying functions for instantiation and message passing.*

29.4. Example of the general class pattern

Lecture 8 - slide 7

Let us now take a look at the `Point` class, programmed with the class pattern introduced in Section 29.3 . We carefully introduce instance variables `x` and `y`, even though they are not strictly needed. The construction parameters of `Point` - also called `x` and `y` - are sufficient to represent the object state.

The Point class redefined to comply with the general class pattern

```
(define (point x y)
 (let ((x x)
       (y y)
      )

   (define (getx) x)

   (define (gety) y)

   (define (add p)
    (point
     (+ x (send 'getx p))
     (+ y (send 'gety p))))

   (define (type-of) 'point)

   (define (self message)
     (cond ((eqv? message 'getx) getx)
           ((eqv? message 'gety) gety)
           ((eqv? message 'add)  add)
           ((eqv? message 'type-of) type-of)
           (else (error "Undefined message" message))))

   self))
```

Program 29.6 *The class `Point` implemented with use of the general class template. The `Point` class corresponds to the `Point` class defined on an earlier page. Notice that the bindings of `x` and `y` in the `let` construct is superfluous in the example. But due to the simultaneous name binding used in `let` constructs, they make sense. Admittedly, however, the `let` construct looks a little strange.*

Below, in Program 29.7 we show a scenario in which we create two points, and bind them to the variables `p` and `q`. The sum of `p` and `q` is bound to the variable named `p+q` . We also send `getx` and `gety` messages to the points in order to assure, that they are located as expected.

```
1> (define p (new-instance point 2 3))

2> (send 'getx p)
2

3> (define q (new-instance point 4 5))

4> (define p+q (send 'add p q))

5> (send 'getx p+q)
6

6> (send 'gety p+q)
8
```

Program 29.7 *A sample construction and dialogue with point.*

29.5. A general pattern of classes with inheritance

Lecture 8 - slide 8

Now that we have seen how the simple aspects of object-oriented programming can be simulated in Scheme, we will move on to a slightly more advanced aspect, namely inheritance. We will see that it is possible to organize object parts in such a way that they can be considered as an object of a class, which inherits from another class.

In Program 29.8 the binding of `super` to new a 'super part' (the red program fragment) is the place to look first. At this location we instantiate the super part of the current object. In the dispatcher - earlier called `self` - we carry out method lookup in `super` , in case we do not find the method in the current object part. This is in reality the operational manifestation of inheritance (from subclass part to super part - and not as usually conceived, the other way around).

You should already now take a look at the left (green) part of Figure 29.1 to understand the super chain of the object. The `self` variable in the figure holds the value of `dispatch`. Thus, `self` refers to the object handle.

The following shows a template of a function that serves as a subclass of another class

```
(define (class-name parameters)
 (let ((super (new-part super-class-name some-parameters))
       (self 'nil))
   (let ((instance-variable init-value)
         ...)

     (define (method parameter-list)
       method-body)
     ...

     (define (dispatch message)
       (cond ((eqv? message 'selector) method)
             ...
             (else (method-lookup super message))))

     (set! self dispatch))

   self))
```

Program 29.8 *A general template of a simulated class with inheritance.*

When object parts refer to each other in a chain of `super` variables, the top object part is deemed to be special. As usual, the most general class is called `Object`. It is shown in Program 29.9. The `super` reference is 'empty' (the empty list), and the `dispatch` function ends the method lookup at this point.

```
(define (object)
  (let ((super '())
        (self 'nil))

    (define (dispatch message)
        '())

    (set! self dispatch)
    self))
```

Program 29.9 *A simulation of the root of a class hierarchy.*

Below, in Program 29.10, the syntactic sugaring functions `new-instance` , `new-part` , `send` , and `method-lookup` are shown. The function `new-part` is used to make a new *part object*, whereas `new-instance` is used to make a *whole object*. Currently they are identical, but later we will make a small difference in between them. In reality `method-lookup` and `send` have survived from Section 29.3. Only some additional error handling has been added.

```
(define (new-instance class . parameters)
  (apply class parameters))

(define (new-part class . parameters)
  (apply class parameters))

(define (method-lookup object selector)
 (cond ((procedure? object) (object selector))
       (else
         (error "Inappropriate object in method-lookup: "
                object))))

(define (send message object . args)
 (let ((method (method-lookup object message)))
  (cond ((procedure? method) (apply method args))
        ((null? method)
         (error "Message not understood: " message))
        (else
         (error "Inappropriate result of method lookup: "
                method)))))
```

Program 29.10 *Accompanying functions for instantiation, message passing, and method lookup.*

29.6. An example of classes with inheritance

Lecture 8 - slide 9

We will of course take a look at a concrete example with class inheritance. Below we define a heir of `Point` . For the sake of the example, we could also have extended `Point` with a third dimension, but we choose the somewhat foolish specialization called `ColorPoint` .

We sketch one of the favorite toy specializations of `Point` - `ColorPoint`

```
(define (color-point x y color)
 (let ((super (new-part point x y))
       (self 'nil))
   (let ((color color))

     (define (get-color)
       color)

     (define (type-of) 'color-point)

     (define (dispatch message)
       (cond ((eqv? message 'get-color) get-color)
             ((eqv? message 'type-of) type-of)
             (else (method-lookup super message))))

     (set! self dispatch))

   self))
```

Program 29.11 *A specialization of* `Point` *which is called* `ColorPoint`.

We also include a sample dialogue with color points, see Program 29.12. Of course, it is best for you to play with the classes yourself. Adding two color points together creates a point, not a color point. In Exercise 8.2 we explore this problem.

```
1> (define cp (new-instance color-point 5 6 'red))

2> (send 'get-color cp)
red

3> (send 'getx cp)
5

4> (send 'gety cp)
6

5> (define cp-1 (send 'add cp (new-instance color-point 1 2 'green)))

6> (send 'getx cp-1)
6

7> (send 'gety cp-1)
8

8> (send 'get-color cp-1)
Undefined message get-color

9> (send 'type-of cp-1)
point
```

Program 29.12 *A sample construction and sample dialogue with* `ColorPoint`.

Exercise 8.2. *Color Point Extension*

On this page we have introduced the class `ColorPoint`, which inherits from `Color` .

In the sample dialogue with a couple of color points we have identified the problem that the sum of two color points is not a color point. Why is it so?

You are now supposed to make a few changes in the classes `Point` and `ColorPoint`. In order to make it realistic for you to play with the classes, your starting point is supposed to be a pre-existing file, with all the useful stuff (available from the on-line version of this material).

When you experiment with `points` and `color-points`, use `M-x run-laml-interactively` from Emacs.

1. First take a look at the existing stuff, and make sure you understand it. Be aware that both of the classes `Point` and `ColorPoint`use virtual methods, as explained below.
2. Add a method `class-of` to both `Point` and `ColorPoint` that returns the class of an instance. Underlying, the method `class-of` is supposed to return a function.
3. Now repair the method `add` in `Point`, such that it always instantiate a class corresponding to the class of the receiver. In other words, if the receiver of `add` is a `Point`, instantiate a `Point`. If the receiver of `add` is a `ColorPoint`, instantiate a `ColorPoint`. You can probably use the method `class-of` from above. (If you run into a problem of a missing parameter in the instantiation of the 'sum point' - you are invited to take a look at my solution).

29.7. The interpretation of `self`

Lecture 8 - slide 10

The simulation of inheritance involves an aggregation of object parts to a holistic object. In order to tie the whole object together, `self` (the object handle) of all parts must point to the most specialized object part.

In Figure 29.1 we show what we want to achieve. The green hierarchy to the left shows the situation until now, where `self` at each level points to the current object part. The yellow hierarchy to the right shows the situation we want to establish.

In order to obtain *virtual methods* of classes we need another interpretation of `self`

The interpretation of `self` can be related to virtual methods in the object-oriented paradigm. A method is virtual if the type of the receiving object, rather the type of the qualifying class, determines the method binding in a method lookup process.

Thus, from an arbitrary object part, we want to be able to access the most specialized interpretation of a method. In order to provide for this, `self` must give access to the most specialized object part. Without this, there is no way at all to access the 'top object part' from a 'non-top object part'.

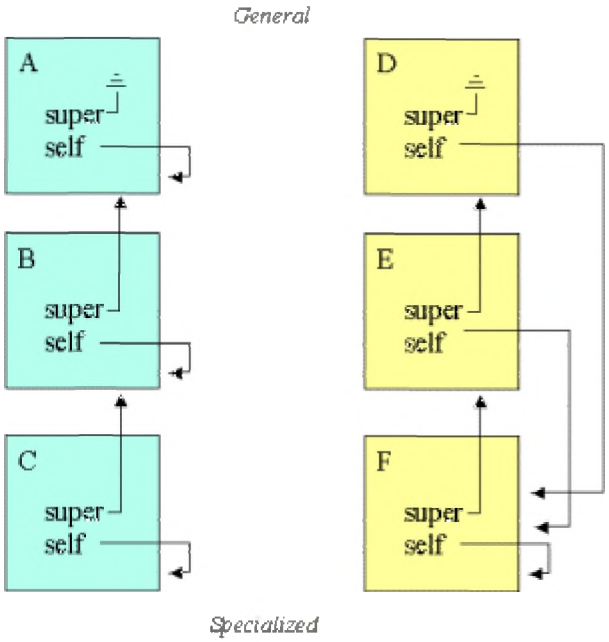

Figure 29.1 *Two different interpretations of `self`. We see two linear class hierarchies. The class C inherits from B, which in turn inherits from A. And similarly, F inherits from E, which inherits from D. The one to the left - in the green class hierarchy - is the naive one we have obtained in our templates and examples until now. The one to the right - in the yellow class hierarchy, shows another interpretation of `self`. Following this interpretation, `self` at all levels refer to the most specialized object part.*

29.8. A demo of virtual methods

Lecture 8 - slide 11

It is now time to show the effect of *virtual methods*. In Program 29.13 we define a class `x` which is the superclass of a class `y` in Program 29.14. Thus, `y` inherits from `x` . In both of the classes we see an extra bookkeeping method `set-self!` , which is responsible for mutating `self` to the proper object part. Notice that `set-self!` is not of interest to the programmers, who use the `x` and `y` classes. The `set-self!` methods are internal affairs of the classes.

On this page we will make two artificial classes with the purpose of demonstrating virtual methods

```
(define (x)
 (let ((super (new-part object))
       (self 'nil))

   (let ((x-state 1)
         )

     (define (get-state) x-state)

     (define (res)
       (send 'get-state self))

     (define (set-self! object-part)
         (set! self object-part)
         (send 'set-self! super object-part))

     (define (self message)
         (cond ((eqv? message 'get-state) get-state)
               ((eqv? message 'res) res)
               ((eqv? message 'set-self!) set-self!)
               (else (method-lookup super message))))

      self)))  ; end x
```

Program 29.13 *A base class* `x`. *The method* `res` *sends the message* `get-state` *to itself. If an* `x` *object receives the message* `res`, *it will return the number 1.*

```
(define (y)
 (let ((super (new-part x))
       (self 'nil))

   (let ((y-state 2)
         )

     (define (get-state) y-state)

     (define (set-self! object-part)
         (set! self object-part)
         (send 'set-self! super object-part))

     (define (self message)
         (cond ((eqv? message 'get-state) get-state)
               ((eqv? message 'set-self!) set-self!)
               (else (method-lookup super message))))

      self)) ; end y
```

Program 29.14 *A class* `y` *that inherits from* `x`. *The* `y` *class redefines the method* `get-state`. *The* `y` *class inherits the method* `res`. *If a* `y` *method receives the message* `res`, *it will be propagate to the* `x` *part of the object. Due to the use of virtual methods,* `self` *in the* `x` *part refers to the* `y` *part. Therefore the* `get-state` *message returns the* `y`*-state of the* `y` *part, namely 2.*

The dialogue in Program 29.15 is a minimal example that illustrates the effect of the new interpretation of `self`. Sending the `res` message to the `y`-object `b` gives the value 2, which shows that the `get-state` of `y` (not the `get-state` of `x`) is called by the `res` method. Notice that the `res` method is inherited from `x` to `y`.

```
1> (define a (new-instance x))

2> (define b (new-instance y))

3> (send 'res a)
1

4> (send 'res b)
2
```

Program 29.15 *A dialogue using class* `x` *and class* `y`*. Instances of the classes* `x` *and* `y` *are created and* `res` *messages are send to both of them.*

The program below, Program 29.16, shows the `new-instance` function, which - in a declarative fashion - asks for virtual operations. The function `virtual-operations` , sends the `set-self!` method to the object, which in turn will activate the `set-self!` methods at all levels in the object. This causes the adjustment of `self` , as illustrated in the right part of Figure 29.1.

```
(define (new-instance class . parameters)
 (let ((instance (apply class parameters)))
   (virtual-operations instance)
   instance))

; Arrange for virtual operations in object
(define (virtual-operations object)
  (send 'set-self! object object))
```

Program 29.16 *The functions* `new-instance` *and* `virtual-operations`.

Exercise 8.3. *Representing HTML with objects in Scheme*

This is an open exercise - maybe the start of a minor project.

In the original mirror of HTML in Scheme, the HTML mirror functions, return strings. In the current version, the mirror functions return an internal syntax tree representation of the web documents. With this, it is realistic to validate a document against a grammar while it is constructed. In this exercise we will experiment with an object representation of a web document. We will use the class and object representation which we have introduced in this lecture.

Construct a general class `html-element` which implement the general properties of a HTML element object. These include:

1. A method that generates a rendering of the element
2. A method that returns the list of constituents
3. An abstract method that performs context free validation of the element

In addition, construct one or more examples of specific subclasses of `html-element` , such as `html` , `head` , or `body`. These subclasses should have methods to access particular, required constituents of an element instance, such as the head and the body of a HTML element, and title of a head element. Also, the concrete validation predicate must be redefined for each specific element.

29.9. References

[normark90a] Kurt Nørmark, "Simulation of Object-oriented Concepts and Mechanisms in Scheme", No. R 90-01, Department of Mathematics and Computer Science, Institute of Electronic Systems, Aalborg University, January 1990, .

[oop-sim] Simulation of object-oriented mechanisms in Scheme - A technical report
http://www.cs.auc.dk/~normark/oo-scheme.html

30. Imperative Scheme and LAML Constructs

This material is about functional programming in Scheme. So why do we include this section about imperative aspects of Scheme and LAML?

The reason is that Scheme in reality is a multi-paradigm programming language - although with strong historical roots in the functional paradigm. As a related observation, in Chapter 29 we saw that the distance between concepts in Scheme and the concepts in object-oriented programming is little too.

When Scheme is used for real world applications - like in the web domain - it is not possible to avoid use of imperative features. Just take, as an example, the handling of files on the harddisk. Some people may be able to regard file handling - deletion and file writing for instance - in a functional way. I prefer to think of these aspects as belonging to the well-known imperative paradigm.

In this brief side track chapter we will review some of the more important imperative aspects of Scheme.

30.1. Imperative Scheme Constructs

Lecture 9 - slide 2

We start by enumerating the main commands - and groups of commands - of the Scheme language. Be sure to notice the assignment syntactical form `set!`, which is the most important of them all.

The most fundamental imperative Scheme construct is the assignment `set!`

- Other imperative constructs:
  - `(begin` e_1 `...` e_n`)`
  - The iterative `do` control structure
  - The input output procedures
  - The list, string and vector mutators

As a notational convention, most imperative abstractions in Scheme ends with "`!`"

As noticed above it is hard to avoid using imperative constructs when writing real-world Scheme programs in the web domain. But we should be careful. We do not want to mix traditional use of 'the imperative programming style' with functional programming. Thus, we should clearly avoid using `set!` side by side with all the functional means, which we have described in this material.

As a good rule of thumb, the imperative constructs in our programs should be kept at a minimum, and - most important - they should be used at a few places - typically at top level - such that the inner parts of the program are purely functional.

30.2. List mutators

Lecture 9 - slide 3

In Chapter 6 we have studied the list concepts in Scheme (proper and improper lists) and we have seen the functional primitives for list construction and list selection - `cons`, `car`, `cdr` and the functions on top of these.

In this section we will mention and list the mutating functions, which are also available in Scheme.

Besides the list constructor `cons` and the list selectors `car` and `cdr` there are also list mutators called `set-car!` and `set-cdr!`

- The list mutator procedures:
  - The command `(set-car! x y)` changes the value of the car position of the cons cell referred by `x`. The car position is assigned to `y`
  - The command `(set-cdr! x y)` changes the value of the cdr position of the cons cell referred by `x`. The cdr position is assigned to `y`
  - Using the list mutators it is possible to make circular structures

Without use of the list mutators, and with use of structural equivalence predicates, it is not possible to tell the difference between a list structure and a copy of the list structure

If we make use of list mutators we can tell the difference between a list structure, LS, and a copy of LS. The way to do it is to mutate LS and observe that the copy is not changed. Using only the functional subset of Scheme (and disregarding the very discriminating equality predicates such as `eq?` and `eqv?`) a copy of LS serves the same purposes as LS itself.

30.3. String mutators

Lecture 9 - slide 4

Strings can be mutated in Scheme. Below we mention the main procedure for this, `string-set!`, and we also look at a similar procedure, `string-fill!`.

A string can be mutated by the `string-set!` and the `string-fill!` procedures

- The string mutator procedures:
  - `(string-set! str k chr)` changes character number `k` in `str` to `chr`
  - `(string-fill! str chr)` changes every character in `str` to `chr`

There are similar functions that mutate the elements in vectors

30.4. Imperative features in LAML

Lecture 9 - slide 6

As mentioned in the introduction to this chapter, we need imperative features to deal with the real-world needs in both static and more dynamic web authoring and programming.

In this section we enumerate some of the more important imperative aspects in the LAML software package.

LAML needs imperative features for file IO, handling of LAML context information, and high level commands

- Overview of imperative features in LAML:
  - The procedures `write-text-file` and `read-text-file`
    - At a higher level procedures such as `write-html` which surrounds many HTML mirror documents in Scheme
  - Many high level procedures which represent tools or commands
    - Such as `xml-parse`, `xml-pp`, and `schemedoc`
    - File handling in general
  - Definition of LAML context information
    - `fake-startup-parameters`, `laml-cd`, ...

In some situations we have internally in LAML used *imperative patching* of functional programs, because *functional patching* has been too difficult and too time consuming

Unfortunately, we have not systematically used the 'exclamation mark' naming convention of LAML procedures.

30.5. References

[-] LAML tools and commands
http://www.cs.auc.dk/~normark/scheme/distribution/laml/man/laml.html#SECTION7

[-] The Text Collection and Skipping Library
http://www.cs.auc.dk/~normark/scheme/distribution/laml/lib/man/collect-skip.html

[-] Reading of writing of text files
http://www.cs.auc.dk/~normark/scheme/distribution/laml/lib/man/file-read.html

[-] R5RS: Vectors in Scheme
http://www.cs.auc.dk/~normark/prog3-03/external-material/r5rs/r5rs-html/r5rs_62.html#SEC64

[-] R5RS: Strings Scheme
http://www.cs.auc.dk/~normark/prog3-03/external-material/r5rs/r5rs-html/r5rs_61.html#SEC63

[-] List and pairs in Scheme

http://www.cs.auc.dk/~normark/prog3-03/external-material/r5rs/r5rs-html/r5rs_58.html#SEC60

[-] Input output procedures
http://www.cs.auc.dk/~normark/prog3-03/external-material/r5rs/r5rs-html/r5rs_65.html#SEC67

[-] do
http://www.cs.auc.dk/~normark/prog3-03/external-material/r5rs/r5rs-html/r5rs_35.html#SEC37

[-] begin
http://www.cs.auc.dk/~normark/prog3-03/external-material/r5rs/r5rs-html/r5rs_34.html

[-] Scheme assignment
http://www.cs.auc.dk/~normark/prog3-03/external-material/r5rs/r5rs-html/r5rs_30.html