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)`
	`30414093201713378043612608166064768`
	`844377641568960512000000000000`

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.

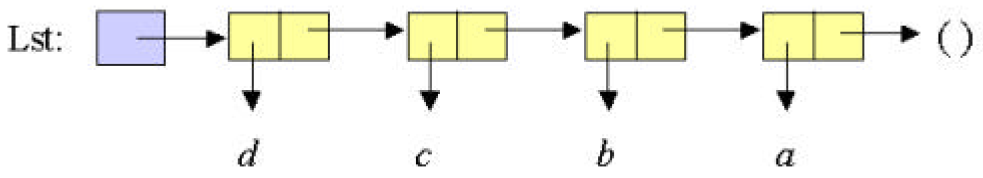

Figure 6.1 *A list (d c b a) constructed by evaluating the expression* `(cons 'd (cons 'c (cons 'b (cons 'a '()))))`.

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.

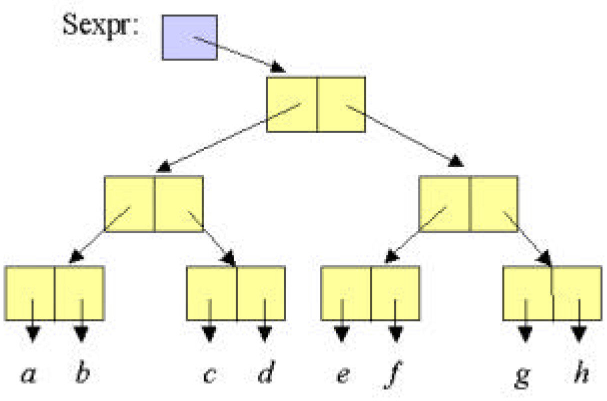

Figure 6.2 *A symbolic expression which illustrates the most general form it can take - a binary tree*

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.

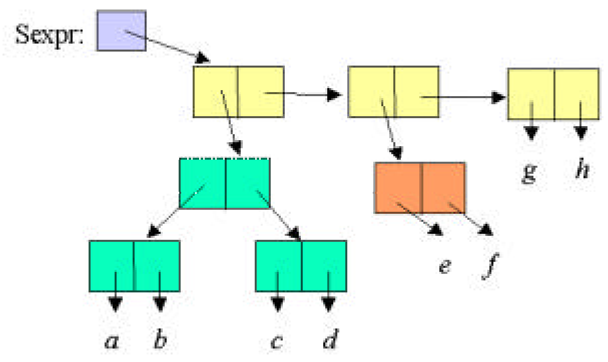

Figure 6.3 *The same symbolic expression laid out as a list. The expressions is a proper list if and only if h is the empty list. If h is not the empty list, the symbolic expression is an improper list.*

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 extensiona lly 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.

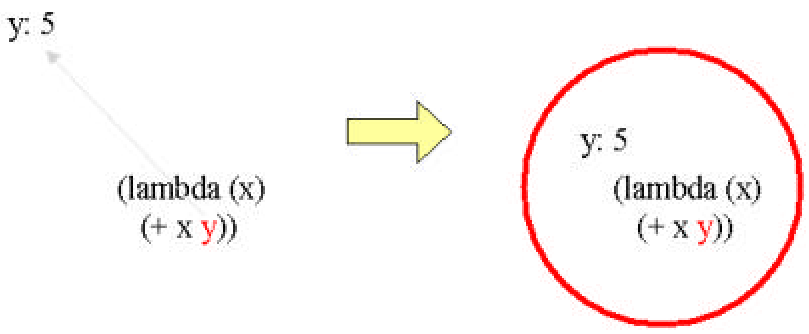

Figure 8.1 *A lambda expression with a free name y. The name y is bound outside the lambda expression. A closure is formed by associating the lambda expressions (the syntactic form) with the binding of the free names.*

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.