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.