KRF

Knowledge Representation Framework Project

Department of Computer and Information Science, Linköping University,
and Unit for Scientific Information and Learning, KTH, Stockholm

Erik Sandewall

List Processing in the Knowledge Representation Framework

This series contains technical reports and tutorial texts from the project on
the Knowledge Representation Framework (KRF).

The present report, PM-krf-011, can persistently be accessed as follows:

Project Memo URL:	http://www.ida.liu.se/ext/caisor/pm-archive/krf/011/
AIP (Article Index Page):	http://aip.name/se/Sandewall.Erik.-/2010/008/
Date of manuscript:	2011-01-02

Related information can also be obtained through the following www sites:

KRFwebsite:	http://www.ida.liu.se/ext/krf/
AIP naming scheme:	http://aip.name/info/
The author:	http://www.ida.liu.se/∼erisa/

Introduction

This is the first one in a sequence of lecture notes that are intended for use in a course on Artificial Intelligence and its software techniques. The present one is dedicated to the traditional issue of *list processing,* including the use of lists (recursively nested sequences), symbols and their attributes. The main part of the text uses a computer-oriented variant of the traditional notation of discrete mathematics and formal logic, as defined in the Knowledge Representation Framework. The text end ends with an account of the Lisp programming language, which is similar in spirit but uses somewhat different notational conventions.

The next lecture note in the series proceeds with an account methods for the preservation and management of knowledgebase content. The emphasis is on knowledgebases of moderate size and with a rich structure, rather than massively large ones with a simpler and uniform structure.

The third lecture note addresses software techniques for intelligent autonomous agents, including software architectures for such agents (BDI, HTN, etc) and languages such as KQML, FIPA-ACL, KIF, PDDL.

There is a preceding text, "Knowledge Representation Framework: Overview of Languages and Mechanisms," that should be read as an introduction and before beginning to read the present text. It will be referred to in the sequel as the *KRF Overview.*

Chapter 1

Introduction

1.1 History of List Processing

Knowledge-based autonomous systems are software systems that maintain and use a collection of information about their environment, and often also about themselves, and that use this information for taking action in that environment. The software technology for such systems is based on two principles: First, techniques for the computation of *recursive functions of symbolic expressions,* which is usually called *list processing* and was pioneered by the Lisp programming language. [1] "list processing." Second, the use of *formal logic,* and in particular first-order predicate logic as a representation language.

The history of these software techniques is almost as long as software technology itself. The Lisp language was defined in 1959, and the systematic use of logic for these purposes began with the introduction of the `Prolog` language ten years later. Developments since then include both developments within the respective languages, and the transfer of techniques to other kinds of programming languages. Lisp has developed with respect to programming environment (the Interlisp system), standardization (CommonLisp), and the inclusion of object-oriented constructs (CLOS). It has also inspired languages that are strongly similar to itself (Scheme) or moderately similar (Haskell, Python), as well as languages that have imported specific facilities (Simula and Java for garbage collection, Java for serialization of data structures, Ruby for closures). Lisp is a basis for the family of *functional programming languages* which include e.g. ML and Microsoft's Sharp-F language, besides Haskell. Prolog has developed in similar ways and given rise to, or influenced Constraint Logic Programming (CLP) and the language Oz.

Although there are many cases where an application or an experimental system has been written directly in Lisp, this language plays an equally important role as an *advanced implementation language* for other, high-level or application-specific languages. Well-known practical examples include its use as the internal language in the Emacs text editor and in the AutoCAD software for computer-aided design. "Understanding Lisp" shall therefore not only be understood as "understanding how to write programs in Lisp,"

[1] The name Lisp was originally an acronym for

is must also be understood as "understanding how languages with new concepts can be implemented in Lisp."

In an interesting and relatively recent development, specialists in several fields outside of computer science have begun using the Scheme language for modelling phenomena and process in their respective disciplines. Scheme has been obtained from Lisp by "cleaning up" the design and making it more systematic and elegant. This use of Scheme is based on the textbook *Structure and Interpretation of Computer Programs* by Sussman and Abelson.

1.2 Goal and Approach in this Compendium

The purpose of the present compendium (= lecture notes) is to present the above-mentioned cluster of programming techniques in a concise and coherent fashion. The purpose is not merely to introduce a particular programming *language,* but to introduce a language together with a number of general *design principles* for programming languages and systems in this family of languages, as well as *software techniques* that are important in the contexts where these languages are widely used. The *Knowledge Representation Framework* is used throughout as a well-defined notational and conceptual' basis.

The present, first part of the compendium is language-oriented. It begins with a basic notation called CEL (Common Expression Language), and then proceeds to introduce the Lisp language as a notational variant of CEL. At the end of Part I the reader should be able to write simple programs in Lisp as well as function definitions in CEL; he or she should also have understood the related system design principles.

The Leonardo software system is an implementation of the Knowledge Representation Framework, and it is used as the software tool for the present text.

The CEL notation is closely related to the notation of *predicate logic,* and indeed one major reason for first introducing CEL and then describing Lisp as a variant of it is to show the connection between Logic and `Lisp.` Expressions in predicate logic are called *propositions,* and propositions are used extensively in the present text, but a full-fledged introduction to logic has to be made separately. The reader who is not already familiar with predicate logic is advised to study the accompanying compendium, "Introduction to CEL and Logic" concurrently with the present text.

1.3 Prerequisites and Exercises

The *KRF Overview* should be read before the present text and as a preparation for it, as was mentioned in the Introduction section.

The reader should also be familiar with the Boolean operators *and, or,* and *not* , the usual truth-tables for defining their evaluation, and the standard equivalence rules whereby an expression using these operators can be rewritten to other, equivalent expressions.

No other prerequisites should be needed.

The working of exercises and lab assignments is strongly recommended as a part of the study of the present text.

Chapter 2

The Common Expression Language on Sequences

The syntax for *Knowledge Representation Expressions* and for the *Common Expression Language* were defined in the *KRF Overview* report. The present report (Part I) will address a subset of these languages where sequences are the only kind of composite expressions. This is the topic of *list processing.*

2.1 Expressions on Sequences

The KRE syntax allows three kinds of elementary expressions: *symbols, strings,* and *numbers,* and a variety of composite expressions that can be constructed from these elements. In this section we consider that part of KRE that only allows one kind of composite expressions, namely *sequences.* Sequences are enclosed by angle brackets in the notation, as in

```
<symbol-1 "This is a string" 3.1416 symbol-2>
```

Symbols are a kind of *entities.* Full KRE also allows *composite entities* but these are not considered in the present chapter, so for now symbols and entities are the same thing.

Expressions in CEL are formed using functions and other operators that have KRE expressions as their arguments and values. As a special case, each KRE expression is itself a CEL expression. Each correctly formed CEL expression has a *value;* KRE expressions have themselves as values.

Consider now CEL expressions that also contain functions, as in the following example

```
(e3  <a b c d e>)
```

The function `e3` obtains (i.e., has as value) the third element of the sequence given as its argument. The value of the CEL expression above is therefore the entity `c` .

In our further examples we shall write the CEL expression and its value on the same line, for compactness, and separated by `=>` as follows:

```
(e3 <a b c d e>)   =>   c
```

This is similar to the notation in the Leonardo system dialog, except that there you type the expression on one line and then obtain the `=>` arrow and the value on the next line.

The functions `e1, e2, e3` and `e4` are defined; for later elements one uses the more general function `en` that takes the desired position as its first argument, as in

```
(en 3 <a b c d e>)   =>   c
```

The function `concat` concatenats two or more sequences, as in

```
(concat <a b> <c d e> <g h>)   =>   <a b c d e g h>
```

The functions `t0, t1, t2, t3, t4` are analogous to the `ei` functions, but they have the effect of obtaining the *rest of* a sequence when a few elements at the beginning are omitted. For example,

```
(t3 <a b c d e>)   =>   <d e>
(t1 <a b c d e>)   =>   <b c d e>
(t0 <a b c d e>)   =>   <a b c d e>
```

The function `tn` of two arguments is analogous to `en`. The rest of a list that is obtained in this way is often called a *tail* which is the reason for using the letter `t` in these function names. The function `tl` is defined both with a digit `1` and a letter `l` for convenience.

2.2 Evaluation and Session Variables in a Leonardo Session

The examples in the previous section showed *expressions* that have a *value* which is obtained as the result of *evaluating* the expression. This is the main topic of the present report, but for presentational convenience we shall make a brief digression into how expressions can be evaluated in an implementation such as a Leonardo system.

As soon as a Leonardo session has started it is ready to evaluate CEL expressions. In this case the user input shall consist of a point character (.) followed by the expression in question on a single line, and the return 'character'. Formally, the point is considered as a command that says "evaluate my argument and show the result." The value of the expression is obtained on the next line, for example as follows

```
502) .(t2 <a b c d e>)
  => <c d e>
```

In each session it is possible to introduce *session variables* and assign values to them. These values can then be used in further input expressions. The following command

```
ssv .alpha 5
```

introduces `.alpha` as a session variable, if it is not already, and assigns the value 5 to it, so `ssv` behaves like assignment in most programming languages. Thereafter during the same session, one can use it as a component of expressions, as in the following examples.

```
<3 4 .alpha 6 7>   =>   <3 4 5 6 7>
.alpha   =>   5
```

Also, of course, after having entered the command

```
ssv .beta <3 .alpha 7>
```

one will have for example

```
(concat (t1 .beta) <9 11>)   =>   <5 7 9 11>
```

Session variables are in fact of relatively little use in actual systems since there are other, and more structured ways of storing and accessing information. They are however useful during program development sessions, for example for keeping test data, and they are also convenient for the examples in the present compendium.

The general rules are as follows. Symbols beginning with a point character are considered as variables and not as entity symbols. If a sequence expression contains one or more variables, then the current values of those variables are used when the expression is evaluated. The `ssv` command-verb takes two arguments where the first one must be given as a variable; the second argument is evaluated and then its value is assigned to the variable that is the first argument.

2.3 Predicates and Propositional Connectives

The following are some simple examples using the predicate `equal`

```
[equal <a b> <a b>]   =>   [true]
[equal 4  <c d>]      =>   [false]
[equal 4  <4>]        =>   [false]
```

In general, a *literal* is formed using a *predicate* followed by its arguments, and surrounded by square brackets. Literals are CEL expressions whose value can be `[true]` or `[false]` . Certain other values may also occur, for example `[unknown]` , but they belong to the more advanced topics. These *truth-values* are in fact also literals, so `[true]` and `[false]` are predicates without arguments that have their own expressions as values.

Each predicate has a *negation* which is written by prefixing the predicate symbol with the dash sign. The value of the negation is the opposite of the value of the original predicate with the same arguments, for example:

```
[-equal <a b> <a c>]   =>   [true]
```

An expression formed using a negated predicate and its arguments is called a *negative literal.*

The equal sign `=` can sometimes be used instead of symbol `equal` but in this particular case the negation is written as `/=` rather than `-=`

Literals can be combined using the operators `and, or,` and `not,` always enclosed by ordinary, round parentheses. These *propositional connectives* evaluate according to the standard truth-tables, as long as their arguments are nothing else than `[true]` or `[false]` . The operators `and` and `or` can take any number of arguments; `not` can only take one argument. One example should be sufficient:

```
(and [equal 4 4] (not [equal 4 5])]   =>   [true]
```

Notice that if `-pred` is the negation of a predicate `pred` of for example two arguments, then `[-pred .x .y]` has the same value as `(not [pred .x .y] )` for any values of the arguments.

2.4 Function Definitions

The element functions `e1` and onwards, the tail functions `t0` and onwards, and the `concat` function are the basic ones for operating on sequences, and they may be used directly for defining the effects of action-verbs. A later chapter will describe how to do this. However, it is very often required to define additional functions in terms of the given ones. This is done using the action-verb `def` like in the following example:

```
def [equal (myfun .a .b) (concat .b <"-"> .a)]
```

After this definition has been received in the session, the function `myfun` can be used like in the following example:

```
(myfun <"alpha" "beta"> <"phi" "psi">)   =>
                    <"phi" "psi" "-" "alpha" "beta">
```

In general the operator `def` takes a single argument which must be given as an expression headed by the predicate `equal.` The first argument of `equal` must be given as an expression consisting of the function to be evaluated, with arguments that are given as variables; the second argument of `equal` must be given as a CEL expression which may use those variables.

Note: When we write that an argument must be "given as" a particular structure, we mean that it must be written directly in that form. For example, in the case of the `def` operator, it would *not* be correct to write

```
def [equal .c (concat .b <"-"> .a)]
```

referring to a variable `.c` whose *value is* the expression `(myfun .a .b).` In the opposite case, where it is accepted to write the argument using functions, variables, etc which are evaluated before the operation in question is performed, we write merely that the argument *must be* this or that, when restrictions on it have to be made.

In practice one will usually write function definitions in files that can be loaded into a session, instead of having to type them into the session directly. The methods for doing this will be described after we have first introduced logical primitives and their use.

Most function definitions contain conditional expressions, for example:

```
def [equal (altfun .a .b) (if [equal .a 0] .b .a)]
```

The operator `if` shall have two or three arguments, where the first argument must be given as a proposition (whose value can only be `[true]` or `[false]` in fact) and the other argument(s) are given as arbitrary CEL expressions. An expression headed by `if` is evaluated in the fairly obvious way: the first argument is evaluated, and then the second or the third argument is evaluated and the value is obtained, depending on the value of the first argument. If the third argument is omitted then the value `nil` is intended.

The following is the definition of a function that obtains the last element of a non-empty sequence, as an example.

```
def [equal (last .a) (if [equal (t1 .a) <>] (e1 .a)
                         (last (t1 .a)) )]
```

This definition uses the function `last` *recursively.* The use of recursive function definitions is characteristic of Lisp as well as all other languages that operate on symbolic expressions, and getting used to writing recursive function definitions is an important aspect of understanding such languages.

2.5 From CEL to Lisp

The simple CEL language that we have introduced so far is similar to the central part of the Lisp language. There are a number of differences with respect to the naming of functions and predicates, but these are of course entirely trivial. Most of the function names in the core of Lisp were chosen in the early days of the language, and some of them are a bit weird, such as `car, cadr, caddr` for `e1, e2, e3.` We choose to replace such names in the present compendium.

There is however one major difference, which concerns the representation of variables. CEL is similar to most programming languages, and to Logic, in that it makes a distinction between variables and atomic data items. Lisp does not make that distinction, and uses an operation called quoting instead.

This peculiarity in Lisp is due to another basic choice, namely, that all data and all evaluable expressions are written using one single kind of brackets, namely the round parentheses. Thus a KRE sequence such as

```
<a b c d>
```

will be written as follows in Lisp

```
(a b c d)
```

and the following CEL expression

```
(concat <a b> <c d>)
```

will be written as follows in elementary Lisp

```
(concat (quote (a b)) (quote (c d)))
```

The `quote` is not a representation of the angle bracket; it is needed in order to identify the arguments as data. If one writes the following expression

```
(concat (a b) (c d))
```

in Lisp then it assumes that `a` is a function of one argument, and that its argument `b` is a variable whose value is to be looked up before the function `a` is applied to it. The same applies for `(c d)` of course.

The function `concat` is actually called `append` in Lisp but this is a technical detail. Expressions formed recursively using round parentheses in Lisp are called *S-expressions* and each parenthesized level in such an expression is called a *list.* Functions on lists in Lisp correspond fairly directly to functions on sequences in CEL, therefore.

One consequence of this arrangement is that Lisp also needs a list-forming function. The following expression in CEL

```
<alpha beta .x delta>
```

would be written as follows in Lisp

```
(list (quote alpha) (quote beta) x (quote delta))
```

Here, the `quote` expressions mark that their arguments shall *not* be treated as variables, so the system shall not try to look up their values, but `x` is not quoted, so it is a variable in this position, and its value is looked up. The function `list` forms a list consisting of the values of its successive arguments.

Repeated use of the `quote` operator tends to be clumsy and it can therefore be abbreviated using the single-quote character. The last example above can therefore equivalently be written as

```
(list 'alpha 'beta x 'delta)
```

but the internal representation in the executing system uses the unabbreviated representation.

The approach of Lisp with respect to variables and quoting may take some time for getting used to, but it has considerable advantages as well as some long-term disadvantages. The major advantage is formal simplicity: it leads to an absolutely minimal language which is used for both 'data' and 'programs'. This makes it very easy to write programs that operate on other programs, for example, which is important when Lisp is used as an advanced implementation language.

One disadvantage with the Lisp approach that does not go away with practice is that it increases the superficial difference between Lisp and the conventional notation of Logic. CEL is much closer to logic notation. This explains why we have chosen to use CEL for the introductory parts of this compendium, postponing the more extensive treatment of Lisp to a later chapter.

2.6 From CEL to Logic

Whereas the step from CEL to Lisp is a change of representation and requires a transformation on the expressions concerned, the step from CEL to the notation of Logic is mostly a change of *usage* of these expressions. We have described CEL as a language for expressions that can be evaluated in a computational context. Predicate logic uses expressions which have exactly the form of CEL propositions, since they are constructed using the propositional connectives (`and, or,` etc.) operating on literals consisting of predicates and their arguments. However, the primary use of propositions in Logic is as *assertions,* in the sense that the user states a number of propositions as being known facts. In effect they constitute the knowledgebase. The primary operation on the knowledgebase is to *draw conclusions,* that is, to obtain other propositions by combining known information in well-defined ways. This is a different kind of operation compared with the evaluation of propositions and other CEL expressions which occurs in the computational setting.

This said, one must however also be aware that even in logic there is a notion of evaluation of propositions, but it is used as a formal tool for proving the correctness of particular methods of drawing conclusions.

Propositions are used in several ways, therefore: as an argument of operators such as the `if` of CEL, and as assertions in the context of logic. There are in fact additional uses as well, namely, in order to specify the preconditions of actions, and to characterize their intended or actual effects. The use of the notation of logic will therefore be a recurrent theme in the present compendium, and especially in Part II.

2.7 Infix Variants of CEL

We have defined all these functions and predicates with prefix notation, so that the operator in question precedes all its arguments. This is a principled and systematic notation. However, for many of the two-argument operators it is natural often to write them on infix form, so that the operator appears between its two arguments. Specific declarations in the Leonardo system at hand indicate in which cases this is admitted. The following is an example of using this possibility for the predicate `equal` in the definition of the function `last`

```
def [(last .a) equal (if [(t1 .a) equal <>] (e1 .a)
                         (last (t1 .a)) )]
```

Lisp does not have this possibility, except in the Interlisp variant which is not in much use nowadays.

However, in all cases it is necessary to retain the parentheses and brackets; several operators can not be mixed on the same parenthesization level.

Chapter 3

Operations on Scalars and Sequences

By 'scalars' in CEL we mean integers, so-called real numbers, and strings. Like in other computer languages, when we say 'real number' we really mean a rational number expressed using an integer part and decimals. The following is a basic set of functions and predicates on sequences and scalars which shall be used for the present compendium and the associated course. We provide brief definitions of each function, but omit technical details as well as information about what value or other response is obtained if the function is invoked with incorrect arguments.

3.1 Type and Coercion Operations

In some cases it is important to know the type of a particular data object which has been obtained as a value from a function. The following function is used.

`(type-of .x)`

The value is an entity that characterizes the type of the argument. It can be one of the entities `symbol, string, integer, real` or `sequence.` Additional alternatives will be introduced in later chapters.

Coercion functions are functions that convert from one datatype to another. They have a prefix `coe.` indicating that they are this kind of function.

`(coe.string .x)`

Converts a number or a symbol `.x` to a corresponding string.

`(coe.number .s)`

Converts a string `.s` to a corresponding number: if there exists a number `.n` such that the value of `(coe.string .n)` equals `.s` then this number is obtained.

`(coe.entity .s)`

Converts a string `.s` to a corresponding entity.

`(coe.tag .s)`

Converts a string or entity `.s` to a corresponding tag, as in

```
(coe.tag "name")   =>   :name
(coe.tag name)   =>   :name
```

Tags are used in the formation of records. (The syntax for records was defined in the *KRF Overview.*) There are also functions for converting from tags to entities, and to and from variable symbols. These are defined in Chapter 6. Functions for converting between integers and reals are defined in Section 3.2.

3.2 Numerical Functions

`(+ .x .y ... .z)`

Addition of numbers. Each number can be integer or real. The value is a real number if at least one of the arguments is so, otherwise integer.

`(* .x .y ... .z)`

Multiplication of numbers. Integers vs reals are handled like for the `+` function.

`(- .x .y)`

Subtraction of numbers. Integers vs reals are handled like for the `+` function. This function can be used with one or two arguments, but not with more than two. An expression `(- .x)` is interpreted as `(- 0 .x)` which is natural.

`(/ .x .y)`

Division of numbers, where `.y` must be different from zero. The value is an integer if both arguments are integers and the first argument is an even multiple of the second argument, otherwise a real.

`(/up .x .y)`

Similar to the `/` function, but the value is an integer which is obtained by rounding the quotient upwards, in the sense of away from zero, if necessary.

`(/down .x .y)`

Similar to the `/` function, but the value is an integer which is obtained by rounding the quotient downwards, in the sense of towards zero, if necessary. For example,

```
(/down 13 -3)   =>   -4
```

`(coe.up .r)`

The argument is a real number or an integer; the value is then always an integer. A real argument obtains the next 'higher' integer in the sense of moving away from zero, increasing for positive arguments and decreasing for negative arguments. For integers this is the identity function.

`(coe.down .r)`

Similar to `coe.up` but obtains the nearest integer in the direction towards zero.

These functions should not be confounded with the string functions `str.upcase` and `str.downcase` which are defined in Section 3.4.

`(random .n)`

The argument shall be a positive integer. The value is a positive integer that is less than or equal to `.n` .

The underlying Lisp system contains additional functions for computation with ratios (i.e., exact quotients between integers), double and multiple precision 'real' numbers, and very large integers. Applications requiring these facilities should be programmed on the Lisp level, therefore.

3.3 Numerical Predicates

The predicate `equal` has already been introduced and applies to numbers as well as to other objects. In the particular case where the arguments are numbers it can also be written as the symbol `=` , for example `[= .a 4]` By way of exception and for mnemonic reasons, the negation of `=` is written as `/=` and not as `-=`. We also use the following predicates for inequalities between numbers.

`[ls .a .b]`

The first argument is strictly less than the second argument.

Other inequality predicates are `gt` for 'strictly greater than,' `ls=` for 'less than or equal to' and `gt=` for 'greater than or equal to.' The predicate `gt=` is the same as `-ls` and `ls=` is the same as `-gt,` but we use them all in order to benefit from their mnemonic character.

These predicates can only be used with exactly two arguments.

3.4 String Functions

The following functions are defined for strings.

`(str.concat .s1 .s2 ... .sn)`

This function concatenates the strings given as arguments, for example:

```
(str.concat "abc" "---" "def")   =>   "abc---def"
```

`(str.length .s)`

Obtains the length of the string given as argument, expressed as a non-negative integer. The length of the empty string is zero.

`(substring .s .m .n)`

Obtains a substring of `.s` by removing all characters after the first `.n` characters, and also removing the first `.m` characters. For example,

```
(substring "abcdefgh" 2 6)   =>   "cdef"
```

The third argument may be omitted and the value will then extend to the end of the given string.

`(str.upcase .s), (str.downcase .s)`

These functions convert the string given as argument to uppercase characters and to lowercase characters, respectively. Non-letter characters remain unchanged.

Extensive manipulation of strings requires additional functions, including functions that can operate efficiently on the level of individual characters, and on arrays. Such functions are not defined in CEL, and if they are needed for an application of Leonardo then that part of the programming should be done on the Lisp level. They are not of direct interest for the topic of Knowledge-Based Autonomous Systems.

3.5 String Predicates

Besides the `equal` predicate we use the following ones for strings.

`[str.begins .p .s]`

This predicate obtains `[true]` if the first argument is an initial substring of, or equal to the second argument, and `[false]` otherwise.

`[str.ends .p .s]`

Similar to `str.begins` but `.p` is a final substring of, or equal to `.s`

`[str.prec .p .s]`

The first argument precedes strictly the second argument alphabetically.

`[str.prec= .p .s]`

Similar to `str.prec` but for 'precedes or is equal to.'

Notice that the corresponding "string succeeds" predicates can be written as `-str.prec=` and `-str.prec` respectively. The alphabetical ordering is defined in such a way that upper-case and lower-case letters are considered as equivalent. Letters outside the 26-letter alphabet from A to Z are considered to have the order that they have in the Latin-1 character set.

Chapter 4

Other List Processing Operations

4.1 Operations on Sequences

The function `concat` and the functions in the `en` and `tn` families have already been mentioned. The predicate `equal` applies of course to sequences as well as to other types of objects. In addition we have the following operations.

`(length .s)`

Obtains the length of a sequence as a non-negative integer.

`(cons .e .s)`

The first argument can be an arbitrary KRE-expression, the second argument must be a sequence. The value is obtained as the extended sequence where `.e` has been prepended at the beginning of `.s` for example:

```
(cons red <green blue white>)   =>   <red green blue white>
```

`(subseq .s .m .n)`

Obtains a subsequence of the given sequence, in the same way as the function `substring`. All three arguments are required. (If it is intended to extend the subsequence to the end of the given sequence, then the function `tn` may be used).

`(reverse .s)`

Obtains a sequence having the elements of the given sequence, but in reverse order. (Cautionary note for the experienced programmer: This operation destroys the datastructure given as argument.)

`(sort .s .p)`

Obtains a sequence having the elements of the sequence `.s` but sorted according to the predicate `.p`, for example

```
(sort <4 3 7 5> ls)   =>  <3 4 5 7>
```

The predicate `str.prec` is also suited to be used as the second argument of this function.

4.2 Control Operators

The operator `if` has already been introduced, and in principle one can write any program merely using this operation combined with recursion. However, the following operators are also convenient in actual programming.

```
[let :var1 expr1 :var2 expr2 ... :vark exprk in expr]
```

The value of this expression is obtained by first evaluating the expressions `expr1` to `exprk` and binding the variables `.var1` to `.vark` to the respective values obtained. After this, the final expression `expr` is evaluated in the context of these bindings. The symbol `in` is only used as a separator. For example, in an environment where `.v2` has the value 5,

```
[let :v1 (+ .v2 1) in <.v1 .v2>]   =>   <6 5>
```

The `let` operation is useful when it helps to avoid repeating the same large subexpression several times in a surrounding expression, which can help both readability and computational efficiency. Sometimes the value is also affected, as is seen by comparing the following two expressions.

```
<(random 100)(random 100)>
[let :v1 (random 100) in <.v1 .v1>]
```

The second expression will always obtain a sequence of two equal numbers, whereas this is not the case for the first expression.

```
(seq.map .v .x .e)
```

The first argument of this operator must be given as a variable, the second argument must be a sequence, the third argument is given as a CEL-expression. The entire expression is evaluated by first evaluating `x` , and then producing a sequence with the same length as that list, and where each element has been obtained by evaluating `.e` with the variable `.v` bound to the corresponding element in the value of `x`. For example,

```
(seq.map .a <1 2 3 4 5> (* .a .a))   =>   <1 4 9 16 25>
```

```
(apply .f .l)
```

The first argument must be a function, the second argument must be a list. The elements of that list are given as arguments to the function, for example:

```
(apply + <3 4 5>)   =>   12
```