aLisp User’s Guide

Tore Risch
Stream Analyze Sweden AB
Sweden

Version 2.0
2022-01-25
sa_Lisp_2.0.pdf

aLisp is an interpreter for a subset of CommonLisp built on top of the storage manager of the sa.engine system. The storage manager in scalable and extensible, which allows data structures to grow very large gracefully and dynamically without performance degradation. Its garbage collector is incremental and based on reference counting techniques. This means that the system never needs to stop for storage reorganization and makes the behaviour of aLisp very predictable. aLisp is written in C99 and is tightly connected with C. Thus Lisp data structures can be shared and exchanged with C programs without data copying and there are primitives to call C functions from aLisp and vice versa. The storage manager is extensible so that new data structures shared between C and Lisp can be introduced to the system. aLisp runs under Windows, Unix, and OSX an can easily be ported to any architecture having a C99 compiler. It is delivered as either an executable or a C library (DLL, shared library), which makes it easy to embed aLisp in other systems. This report documents how to use the aLisp system.

Table of contents

1. Introduction

aLisp is a small but scalable Lisp interpreter that has been developed with the aim of being embedded in other systems. It is tightly interfaced with C and can share data structures and code with C to avoid unnecessary copying. aLisp supports a subset of CommonLisp and conforms to the CommonLisp standard [1][2] when possible. However, it is not a full CommonLisp implementation, but rather such CommonLisp constructs are not implemented that are felt not being critical and difficult to implement efficiently. These restrictions make aLisp small and light-weight, which is important when embedding it in other systems.

aLisp is designed to be embedded in other systems, in particular the sa.engine data analytics system [4]. However, aLisp is a general system and can be used as a stand-alone Lisp engine as well. This document describes the stand-alone aLisp system and its differences to CommonLisp. aLisp includes a main-memory storage manager used for processing data in real-time. Thus all data structures are dynamic and can grow without performance degradation. The data structures grow gracefully so that there are never any significant delays for data reorganization, garbage collection, or data copying. (Except that the OS might sometimes do this, outside the control of aLisp). There are no limitations on how large the data area can grow, except OS address space limitations. The performance is of course dependent on the size of the available main memory and thrashing may occur when the amount of memory used by aLisp is larger than the main memory.

A critical component in a Lisp system is its real-time garbage collector. Lisp programs often generate large amounts of temporary data areas that need to be reclaimed by the garbage collector. Furthermore, aLisp is designed to be used in a real-time main-memory data storage system and therefore it is essential that the garbage collection is predictable, i.e. it is not acceptable if the system would stop for garbage collection now and then. The garbage collector must therefore be *incremental* and continuously reclaim freed storage. Another requirement for aLisp is that it can *share data structures* with C, in order to be tightly embedded in other systems without causing delays by copying data between subsystems. Therefore, unlike most other implementations of Lisp, both C and Lisp data structures are allocated in the same memory area and there is no need for expensive copying of large data areas between C and Lisp. This is essential for a predictable interface between C and Lisp, in particular if it is used for managing large database objects.

Section 2 explains how to get started to run the aLisp interpreter. Section 3 describes the system functions in aLisp. The differences w.r.t. CommonLisp [1][2] are documented. Section 4 describes functions to handle time while Section 5 describes the I/O system and Section 6 describes the error handling mechanisms.

Section 7 gives an overview of the debugging facilities. aLisp includes a code documentation and search system that documents Lisp functions and allows for searching for Lisp code having certain properties (Sec. 8). The code search system is connected to Emacs editor managing Lisp source code. A code rule driven validation system (Sec. 8.3) searches through Lisp code to find possible errors such as unbound variables, undefined functions, or other questionable Lisp code.

2. Starting aLisp

aLisp is a subsystem of sa.engine [4]. When you download and install sa.engine you also get an executable `sa.core` which includes a stand-alone aLisp engine. Start aLisp from the console with:

```
sa.core
```

When started the system first reads a *database image* file, `sa.core.dmp`, which must be in the same folder as where the `sa.core` executable is located. When started the system runs the *aLisp REPL*(Read Eval Print Loop [3]), where it accepts and evaluates Lisp expressions from the console. In the aLisp REPL the system reads S-expressions, evaluates them, and prints the results from the evaluation.

The macro `setf` assigns one or several variables. It is a CommonLisp generalization of the classical Lisp assignment function `setq`. For example:

```
> (setf a '(a b c))
WARNING! Setting undeclared global variable: A
(A B C)
> (reverse a)
(C B A)
```

As the example shows, aLisp warns the user when an undeclared global variable is set. To avoid this error message all global variables should be declared before being used using `defvar, defparameter, defconstant` or `defglobal` (Sec. 3.2.2). For example, to avoid the warning above one can first declare the variable to be global using `defglobal`:

```
> (defglobal _a_) ← declare _a_ as global
_A_
> (setf _a_ '(a b c))    ← no warning
 (A B C)
```

Here a coding convention is used that global variables always have `_` (undeline) as first character. Let's define a function:

```
> (defun foo (x)(cons 0 a))
Undeclared free variable A in FOO
FOO
>
```

As the example shows, aLisp warns the user when it encounters forms in function definitions that may contain questionable code (Sec. 8.3). In this case we forgot that we renamed variable `a` to globally declared `_a_`. Let's correct the error:

```
> (defun foo (x)(cons 0 _a_))
FOO
(FOO REDEFINED)
WARNING! Redefined function: FOO
>
```

This time the system just warned that `foo` was redefined.

All Lisp code and data is stored inside the *database image* which is a dynamic main memory area. The image can be saved on disk in a file named "myfile" by evaluating:

```
(rollout "myfile.dmp")
```

To later connect sa.engine to a previously saved image, issue the OS command:

```
sa.core myfile.dmp
```

The aLisp system includes a command line debugger which is enabled by default when you enter the Lisp REPL. If debugging is enabled and you make an error the system will enter a *break loop* where the error can be investigated. The simplest thing to do when in the break loop is to enter `:r` to reset the error. For example:

```
> (defun fie (x)(+ x b))
Undeclared free variable B in FIE
FIE
> (fie 1)
```

```
ERROR! Unbound variable: B
When evaluating: B
(FAULTEVAL BROKEN)
In *BOTTOM* brk>
```

In the break loop the system prints a summary of available debug commands when evaluating:

```
:help
```

Try it! The aLisp debugging facilities are documented in Sec. 7.

The documentation of a Lisp function can be printed by calling the function `doc`, for example:

```
(doc 'doc)
(doc 'calling)
```

You can search for Lisp functions whose names include a given substring by calling `apropos`, for example:

```
(apropos 'doc)
(apropos 'calling)
```

The documentation system is documented in Sec. 8.2.

aLisp is a subsystem of sa.engine so the aLisp REPL can be entered from the OSQL REPL of sa.engine by issuing the OSQL command:

```
lisp;
```

To get back to the OSQL REPL evaluate:

```
:osql
```

Quit aLisp by evaluating:

```
(quit)
```

3. Basic Primitives

This section describes the basic aLisp data types and the functions operating over them.

The CommonLisp standard functions are defined in [1][2]. Significant differences between an aLisp function and the corresponding CommonLisp function are described in this document.

3.1. Data types

Every object in aLisp belongs to exactly one data type. There is a *type tag* stored with each object that identifies its data type. Each data type has an associated *type name* as a symbol. The symbolic name of the data type of an aLisp object `O` can be accessed with the Lisp function:

```
(type-of x)
```

For example:

```
(type-of 123) => INTEGER
```

aLisp provides a set of built-in basic data types. Furthermore, through its C-interface aLisp can be extended with new datatypes implemented in C. The system tightly interoperates with C so that i) data structures can be shared between C and aLisp (Sec. 8.3), ii) the aLisp garbage collector is available in C, iii) aLisp can call functions implemented in C, iv) aLisp functions can be called from C, and v) C can utilize aLisp's error management.

3.2. Symbols

A *symbol* (data type name SYMBOL) is a *unique* text string with which various system data can be associated. Symbols are used for representing *functions*, *macros*, *variables*, and *property lists*. Functions and macros represent executable aLisp code, variables bind symbols to values, and property lists associate data values with properties of symbols. Symbols are unique and the system maintains a hash table of all symbols. Symbols are **not** garbage collected and their locations in the database image never change. It is therefore **not** advisable to make programs that generate unlimited amounts of symbols. Symbols are mainly used for storing system data (such as programs) while other data structures, e.g. hash tables, arrays, lists, etc. are used for storing user data. Symbols are always internally represented in upper case and symbols entered in lower case are always internally capitalized by the system.

The system symbol `nil` represents both the empty list and the truth value false. All other values are regarded as having the truth value true.

Each symbol has the following associated data:

1. The *print name* is a string representing the symbol. The print name of a symbol *s* can be accessed by the function (`mkstring` *s*). For example:

   ```
   > (mkstring nil)
   "NIL"
   ```

2. Symbols represent variables and bind them to values. The *global value* of a symbol (Sec. 3.2.1) binds it to a global value. Most values are *local* and bound on a variable binding stack maintained by the system when functions are called or code blocks entered.

3. Each symbol *nm* has an associated *function cell* where an aLisp function definition named *nm* is stored. The function cell of a Lisp symbol *nm* is retrieved with the CommonLisp function `(symbol-function` *nm*`)`. It returns the function definition of *nm* if there is one; otherwise it returns `nil`. A function definition can be one of the following:

   i) A *lambda function* which is a function defined in Lisp (Sec. 3.2.1). A lambda function definition is represented as a list, `(lambda` *args* `.` *body*`)`. It is defined by the special form `defun`, e.g.:

   ```
   > (defun rev (x)(cons (cdr x)(car x)))
   REV
   > (rev '(1 2))
   ((2) . 1)
   ```

   ii) A *macro* (Sec.3.14) is defined by the special form `defmacro`. A macro is a Lisp function that takes as its argument a form and produces a new equivalent form. Many system functions are defined as macros.

They are Lisp's *rewrite rules*.

iii) An *external Lisp function* is implemented in C]. A external Lisp function is represented by the data EXTFN printed as `#[EXTFN`*`n fn`*`]`, where *`n`* is the arity of the function and *`fn`* is its name. The EXTFN data structure internally contains a pointer to the C definition. For example:

```
> (symbol-function 'car)
#[EXTFN1 CAR]
```

iv) A *variadic external Lisp function* can take a variable number of actual arguments. It definition is printed as `#[EXTFN-1 `*`fn`*`]`. For example:

```
> (symbol-function 'list)
#[EXTFN-1 LIST]
```

v) A *special form* is an external Lisp functions with varying number of arguments and where the arguments are not evaluated the standard way. Special forms are printed as `#[EXTFN-2 `*`fn`*`]`. For example:

```
> (symbol-function 'quote)
#[EXTFN-2 QUOTE]
```

4. The *property list* (Sec 3.2.3) associates property values with the symbol and other symbols called *property indicators*.

In function descriptions of this document `X`... indicates that the expression `X` can be repeated, while `[X]` indicates that `X` is optional. The *Type* of a function can be EXTFN (defined in C), SPECIAL (special form), LAMBDA (defined in Lisp), or MACRO (Lisp macro) (Sec. 3.14). The type of functions that are similar or equivalent to standard CommonLisp functions in http://www.cs.cmu.edu/Groups/AI/html/cltl/cltl2.html are prefixed with a '*'. A system variable can be either SPECIAL (dynamically scoped) or GLOBAL (Sec. 3.2.2).

3.2.1. Defining functions

The special forms `defun` creates new list functions. In debug mode the system will automatically analyze the new function definitions to detect eventual fishiness. A fundamental capability of CommonLisp (and aLisp) is its very powerful macro definitions overviewed in Sec. 3.14. Macros are defined using `defmacro`.

Function	Type	Description
`(defc `*`fn def`*`)`	EXTFN	Associate the function definition *`def`* with the symbol *`fn`*. Same as `symbol-setfunction`.
`(defun `*`fn args form`*`...)`	*SPECIAL	Define a new Lisp function.
`(defmacro `*`fn args form`*`...)`	*SPECIAL	Define a new Lisp macro (Sec. 3.14).
`(extfnp `*`x`*`)`	LAMBDA	Return T if *x* is a function defined in C.
`(flet ((`*`fn def`*`) ...) `*`form`*`...)`	*MACRO	Bind local function definitions and evaluate the forms *`form`*...
`(fboundp `*`fn`*`)`	*EXTFN	True if *`fn`* is a function, macro or special form.
`(fmakeundef `*`fn`*`)`	*LAMBDA	Remove function, macro or special form definition of *`fn`*.
`(getd `*`fn`*`)`	EXTFN	Get function definition of symbol *`fn`*. Same as `symbol-function.`
`(lambdap `*`x`*`)`	LAMBDA	Return T if *x* is a lambda expression.

`(movd fn1 fn2)`	EXTFN	Make *fn2* have the same function or macro definition as *fn1*.
`(symbol-function s)`	*EXTFN	Get the function definition associated with the symbol *s*. Same as `getd`.
`(symbol-setfunction s d)`	EXTFN	Set the function definition of symbol *s* to *d*. Same as `defc`.

3.2.2. Binding variables

Symbols hold variable bindings. Variables bindings can be either global or bound locally inside a Lisp function or a code block. Local variables are bound when defined as formal parameters of functions or when locally introduced when a new code block is defined using `let` or other variable binding expressions. Both local and global variables can be (re)assigned using the macro `setf`.

In aLisp global variables should be declared before they are used, using the macro `defglobal`. aLisp gives warnings when setting undeclared global variables or using them in functions. There are a number of built-in global variables that store various system information and system objects.

For example:

```
> (setf x 1)
WARNING! Setting undeclared global variable: X   ← because X is undeclared
1
lisp 1> (defglobal _g_)        ← declare _G_ as global
NIL
lisp 1> (setf _g_ 1)           ← assign number 1 to _G_
1
lisp 1> _g_                    ← evaluate _G_
1
lisp 1> (let ((_g_ 3))         ← New block where local variable _G_ initalized
to 3
           _g_)                ← Value of local variable _G_ returned from
block
3
lisp 1> _g_
1                              ← Global value did not change
```

`let` defines a new code block with new variables. For example:

```
> (let ((x 1)                  ← Local X initialized to 1
        Y                      ← Local Y initialized to nil
        (z 2))                 ← Local Z initialized to 2
    (list x y z))              ← Return list of the local variables
(1 NIL 2)
```

Unlike most other programming languages, global Lisp variables can also be *dynamically scoped* so that they are rebound when a code block is entered and restored back to their old values when the code block is exited. In CommonLisp dynamically scoped variables are declared using the special form `defvar` or `defparameter`. Dynamically scoped variables provide a controlled way to handle global variables as they are restored as local variables are when a code block is exited. As a convention, dynamically scoped variables have '*' as first character. For example, assume we have a package to do trigonometric computations using radians, degrees, or new degrees:

```
> (defvar *angle-unit* 1)        ← Units in radians to measure angles
```

```
*ANGLE-UNIT*
> (defun mysin(x)(sin (* x *angle-unit*)))
MYSIN
> (defun hl (ang x)(/ x (mysin ang)))
HL
> (hl 0.785398 10)        ← Compute length of hypotenuse using radians
14.1421
> (setf *angle-unit* (/ 3.1415926 180))  ← Let's use degrees instead
0.0174533
> (hl 45 10)
14.1421
> (setf *angle-unit* (/ 3.1415926 200))   ← Let's use new degrees instead
0.015708
> (hl 50 10)
14.1421
```

Now suppose we want to have a special version of HL that computes the hypotenuse only for regular degrees:

```
> (defun hyplen (ang x)
    (let ((*angle-unit* (/ 3.1415926 180))) ← Rebind *angle-unit* inside HL
       (hl ang x)))
HYPLEN
> (hyplen 45 10)              ← Using degrees inside HYPLEN
14.1421
> (hl 50 10)
14.1421                       ← Restored back to new degrees outside HYPLEN
```

The following system functions handle variable bindings.

Function	Type	Description
`(boundp` *`var`*`)`	*EXTFN	Return T if the variable *`var`* is bound. Unlike CommonLisp, `boundp` works not only for special and global variables but also for local variables, allowing to test whether a variable is locally bound.
`(constantp` *`x`*`)`	*SUBR	Returns T if *`x`* evaluates to itself. Same as `kwoted`.
`(defconstant` *`var val`* `[`*`doc`*`])`	*SPECIAL	Declare symbol *`var`* to denote a constant value assigned to *`val`* that cannot be changed, optionally having a documentation string *`doc`*.
`(defglobal` *`var`* `[`*`val`*`][`*`doc`*`])`	MACRO	Declare *`var`* to be a *global variable* with optional initial value *`val`* and optional documentation string *`doc`*. Unlike dynamically scoped variables global variables are **not** temporarily reset locally with `let` or `let*`. They are much faster to look up than dynamically scoped variables. See also `defvar`.
`(defparameter` *`var val`* `[`*`doc`*`])`	*SPECIAL	Declare *`var`* to be a *special variable* set to *`val`* with optional documentation string *`doc`*.

`(defvar var [val][doc])`	*SPECIAL	Declare `var` to be a *special variable* optionally initialized to `val` set unless `var` is previously assigned, and an optional documentation string `doc`. Special variables are dynamically scoped. See also `defglobal`.
`(global-variable-p var)`	LAMBDA	Return true if `var` is declared to be a global variable.
`(let ((var init)...) form...)`	*MACRO	Bind local variables `var` to initial values `init` in parallel and evaluate the forms `form....` Instead of a pair the binding can also be just a variable, binding the variable to `nil`.
`(let* ((var init)...) form...)`	*MACRO	As `let` but local variables are initialized in sequence.
`(prog-let ((var init...)...) form...)`	MACRO	As `let` but if `(return v)` is called in `form...` then `prog-let` will exit with the value `v`. Notice that the classical Lisp functions `prog` and `go` are NOT implemented in aLisp.
`(quote x)`	*SPECIAL	Return `x` unevaluated.
`(resetvar var val form...)`	MACRO	Temporarily reset *global value* of `var` to `val` while evaluating `form...` The value of the last evaluated form is returned. After the evaluation `var` is reset to its old global value. Usually *special variables* combined with `let/let*` are used instead.
`(set var val)`	*EXTFN	Bind the value of the value of `var` to `val`. This function is normally not used; the normal function to set variable values is `setq` that does not evaluate its first argument.
`(setq var val)`	*SPECIAL	Change the value of the variable `var` to `val`. The macro `setf` (Sec. 3.4) is a generalization of `setq` to allow updating of many different kinds of data structures rather than just setting variable values.
(psetq var_1 val_1 … var_k val_k)	*MACRO	Set in parallel variables var_1 to val_1,..., var_k to val_k using `setq`.
`(special-variable-p v)`	*EXTFN	Return T if the variable `v` is declared as special with `defvar`.
`(symbol-value s)`	*EXTFN	Get the global value of the symbol `s`. Returns the symbol NOBIND if no global value is assigned.

3.2.3. Symbol manipulation

The following system functions do other operations on symbols than handling variable bindings, e.g. managing property lists, testing different kinds of symbols, or converting them to other data types.

A *property list* is represented as a list with an even number of elements where every second element are *property indicators* and every succeeding element represents the corresponding *property value*. Property lists are used for associating system information with symbols and can also be used for storing user data. However, notice that, as atoms are not garbage collected, dynamic databases should *not* be represented with property lists.

Function	Type	Description

`(addprop s i v [flag])`	EXTFN	Add a new value `v` to the list stored for the property `i` of the symbol `s`. If the `flag` is omitted the new value is added to the end of the old value list, otherwise it is added to the beginning.

`(explode s)`	EXTFN	Unpack a symbol `s` into a list of single character symbols. Symbols are exploded into symbols and strings into strings. For example:
		`(EXPLODE 'ABC) => (A B C)`
		`(EXPLODE "abc") => ("a" "b" "c"))`

`(gensym)`	*LAMBDA	Generate new symbols named `G:1`, `G:2`, etc.
`(get s i)`	*EXTFN	Get the property value of symbol `s` having the property indicator `i`.
`(getprop s i)`	EXTFN	Same as `get`.
`(keywordp x)`	*EXTFN	Return T if `x` is a keyword (i.e. symbol starting with ':').
`(mksymbol x)`	EXTFN	Coerce a string `x` to a symbol. The characters of `x` will be in upper-cased.
`(pack x...)`	LAMBDA	Pack the arguments `x`... into a new symbol.
`(packlist l)`	LAMBDA	Pack the elements of the list `l` into a new symbol.
`(put s i v)`	*EXTFN	Set the value stored on the property list of the symbol `s` under the property indicator `i` to `v`.
`(putprop s i v)`	EXTFN	Same as `put`.
`(remprop s i)`	*EXTFN	Remove property value stored for the property indicator `i` in the property list of symbol `s`.
`(symbol-plist s)`	*EXTFN	Get the entire property list of the symbol `i`.
`(symbolp x)`	*EXTFN	Return true if `x` is a symbol.

3.3. *Lists*

Lists (data type name LIST) represent linked lists. There are many system functions for manipulating lists. Two classical Lisp functions are `car` to get the head of a list, and `cdr` to get the tail. For example:

```
> (setf xx '(a b c))
(A B C)
> (car xx)
A
> (cdr xx)
(B C)
>
```

Function	Type	Description
`(adjoin x l)`	*EXTFN	Similar to `(cons x l)` but does not add `x` if it is already member of `l` (tests with `equal`).
`(append l...)`	*MACRO	Make a copy of the concatenated lists `l`... With one argument, `(append l)` copies the top level elements of `l`.
`(assoc x ali)`	*EXTFN	Search association list `ali` for a pair `(x . y)`. Tests with `equal`.
`(assq x ali)`	EXTFN	As `assoc` but tests with `eq`.
`(atom x)`	*EXTFN	True if `x` is not a list or if it is `nil`.

`(butlast l)`	*EXTFN	A copy of list `l` minus its last element.
`(caaar x)`	*EXTFN	Same as `(car (car (car x)))`. Can be updated with `setf`.
`(caadr x)`	*EXTFN	Same as `(car (car (cdr x)))`. Can be updated with `setf`.
`(caar x)`	*EXTFN	Same as `(car (car x))`. Can be updated with `setf`.
`(cadar x)`	*EXTFN	Same as `(car (cdr (car x)))`. Can be updated with `setf`.
`(caddr x)`	*EXTFN	Same as `(car (cdr (cdr x))` or `(third x)`. Can be updated with `setf`.
`(cadr x)`	*EXTFN	Same as `(car (cdr x))` or `(second x)`. Can be updated with `setf`.
`(car x)`	*EXTFN	The head of the list `x`, same as `(first x)`. Can be updated with `setf`.
`(cdaar x)`	*EXTFN	Same as `(cdr (car (car x)))`. Can be updated with `setf`.
`(cdadr x)`	*EXTFN	Same as `(cdr (car (cdr x)))`. Can be updated with `setf`.
`(cdar x)`	*EXTFN	Same as `(cdr (car x))`. Can be updated with `setf`.
`(cddar x)`	*EXTFN	Same as `(cdr (cdr (car x)))`. Can be updated with `setf`.
`(cddddr x)`	*LAMBDA	Same as `(cdr (cdr (cdr (cdr x))))`. Can be updated with `setf`.
`(cdddr x)`	*EXTFN	Same as `(cdr (cdr (cdr x)))`. Can be updated with `setf`.
`(cddr x)`	*EXTFN	Same as `(cdr (cdr x))`. Can be updated with `setf`.
`(cdr x)`	*EXTFN	The tail of the list `x`, same as `(rest x)`. Can be updated with `setf`.
`(cons x y)`	*EXTFN	Construct new list cell.
`(consp x)`	*EXTFN	Test if `x` is a list cell.
`(copy-tree l)`	*EXTFN	Make a copy of all levels in list structure `l`. To copy the top level only, use `(append l)`.
`(eighth l)`	*LAMBDA	The 8:th element in list `l`. Can be updated with `setf`.
`(fifth l)`	*LAMBDA	The 5:th element in list `l`. Can be updated with `setf`.
`(first l)`	*EXTFN	The first element in list `l`, same as `(car l)`. Can be updated with `setf`.
`(firstn n l)`	LAMBDA	A new list consisting of the first `n` elements in list `l`.
`(fourth l)`	*LAMBDA	Get fourth element in list `l`. Can be updated with `setf`.
`(getf l i)`	*EXTFN	Get value stored under the property indicator `l` in the property list `l`. Can be updated with `setf`.
`(in x l)`	EXTFN	Returns true if there is some substructure in `l` that is `eq` to `x`.
`(intersection x y)`	*EXTFN	A list of the elements occurring in both the lists `x` and `y`. Tests with `equal`.
`(intersectionl l)`	LAMBDA	Make the intersection of the lists in list `l`. Tests with `equal`.
`(last l)`	*EXTFN	Return the last tail of the list `l`. E.g. `(last '(1 2 3)) => (3)`
`(ldiff l tl)`	*LAMBDA	Make copy of `l` up to, but not including, its tail `tl`.
`(length x)`	*EXTFN	Compute the number of elements in a list, the number of characters in a string, or the size of a vector.
`(list x...)`	*EXTFN	Make a list of the elements `x`...
`(list* x...)`	*EXTFN	Is similar to `list` except that the last argument is used as the end

		of the list.
		For example:
		`(list* 1 2 3) => (1 2 . 3)`
		`(list* 1 2 '(a)) => (1 2 A)`

`(listp x)`	*EXTFN	True if *x* is a list cell or `nil`.
`(member x l)`	*EXTFN	Tests with `equal`if element *x* is member of list *l*. Returns the tail of *l* where *x* is found first. For example: `(member 1.2 '(1 1.2 1.2 3)) => (1.2 1.2 3)`
`(memq x l)`	EXTFN	As `member` but tests with `eq` instead of `equal`.

`(merge lx ly fn)`	*LAMBDA	Merge the two lists *lx* and *ly* with *fn* as comparison function.
		For example:
		`(merge '(1 3) '(2 4) (function <))`
		`=> (1 2 3 4)`

`(mklist x)`	EXTFN	Returns *x* if it is `nil` or a list. Otherwise `(list x)` is returned.
`(natom x)`	*EXTFN	True if *x* is not an atom and not `nil`. Anything not being a list is an atom.
`(ninth l)`	*LAMBDA	The 9:th element in list *l*. Can be updated with `setf`.
`(not x)`	*EXTFN	True if *x* is `nil`, same as `null`.
`(nth n l)`	*EXTFN	The *n*th element in list *l* with enumeration starting at 0. Can be updated with `setf`.
`(nthcdr n l)`	*EXTFN	Get the *n*th tail of the list *l* with enumeration starting at 0.
`(null x)`	*EXTFN	True if *x* is `nil`, same as `not`.
`(pair x y)`	EXTFN	Same as `pairlis`.
`(pairlis x y)`	*EXTFN	Form an association list by pairing the elements of the lists *x* and *y*.
`(pop l)`	*SPECIAL	Remove front of list*l*, same as `(setf l (cdr l))`.
`(push x l)`	*MACRO	Add *x* to the front of list *l*, same as `(setf l (cons x l))`.
`(remove x l)`	*EXTFN	Remove elements `equal` to *x* from list *l*.
`(remove-duplicates l)`	*EXTFN	Remove all duplicate elements in the list *l*. Tests with `equal`.
`(rest l)`	*LAMBDA	Same as `cdr`. Can be updated with setf (Sec. 3.4).
`(reverse l)`	*EXTFN	A list of the elements of *l* in reverse order.
`(second l)`	*EXTFN	The 2:nd element in list *l*. Same as `cadr`. Can be updated with `setf`.
`(set-difference x y [equalflag])`		
	*EXTFN	A list of the elements in *x* that are not member of the list *y*. Tests with `eq` unless *equalflag* is true.
`(seventh l)`	*LAMBDA	The 7:th element of the list *l*. Can be updated with `setf`.
`(sixth l)`	*LAMBDA	The 6:th element of the list *l*. Can be updated with `setf`.
`(sort l fn)`	*LAMBDA	Sort the elements in the list *l* using *fn* as comparison function.
`(sublis ali l)`	*EXTFN	Substitute elements in the list structure *l* as specified by the association list *ali* that has the format `((from . to)...)`. For example: `(sublis '((a . 1)(b . 2)) '((a) b a)) => ((1) 2 1)`
`(subpair from to l)`	EXTFN	Substitute elements in the list *l* as specified by the two lists *from* and *to*. Each element in *from* is substituted with the corresponding element in *to*. For example:

		`(subpair '(a b) '(1 2) '((a) b a)) => ((1) 2 1)`
`(subsetp x y)`	*LAMBDA	True if every element in the list *x* also occurs in the list *y*.
`(subst to from l)`	*EXTFN	Substitute *from* with *to* in the list structure *l*. Tests with `equal`. For example: `(subst '1 'a '((a) b a)) => ((1) B 1)`
`(tenth l)`	*LAMBDA	The 10:th element in list *l*. Can be updated with `setf`.
`(third l)`	*EXTFN	The 3:rd element in list *l*, same as `caddr`. Can be updated with `setf`.
`(union x y)`	*EXTFN	A list of the elements occurring in both the lists *x* and *y*. Tests with `equal`.

3.4. Destructive functions

The list functions introduced so far are constructing new lists out of other objects. For example, `(append x y)` makes a new list by copying the list *x* and then concatenating the copied list with the list *y*. The old *x* is removed by the garbage collector if no longer needed. If *x* is long `append` will generate quite a lot of garbage. This is not very serious because aLisp has a very efficient real-time garbage collector that immediately discards no longer used objects. However, sometimes one needs to actually modify list structures by physically replacing pointers. One may wish to do so for efficiency reasons as, after all, the generation of garbage has its cost. Another reason is that some data structures are maintained as lists updated destructively. For this Lisp has some destructive list manipulating functions that replace pointers rather than copying list cells. Notice that such destructive functions may cause bugs that are difficult to find. Therefore destructive functions should be avoided if possible.

To make destructive operations more transparent, most destructive functions are in CommonLisp (and aLisp) replaced with the generic `setf` macro calls a *setter macro* associated with a *getter* that accesses data. The setter macro will transparently update the getter value destructively. Using `setf` the previous example is expressed as follows:

```
> (setf a '(1 2 3 4))
(1 2 3 4)
> (setf (third a) 8)              ← Replace (third a) with 8
8
> a                               ← The list bound to A has changed
(1 2 8 4)
```

The macro `(setf` p_1 v_1 p_2 v_2 `...)` updates the value of each getter p_i to become EQ to v_i. That is, after executing `setf` all p_i=v_i. A getter is an expression accessing data. It can be a *variable* in which case `setf` sets the variables p_i like multiple calls to `setq`, e.g.: `(setf x 1 y 2)`. It can also be a call to a *getter function* (accessor) that has a corresponding setter macro defined, e.g. `(setf (third a) 8)`. In this case there is a setter macro associated with the getter function `third`. There are setter macros defined for the getters `aref` (arrays), `getl` (property lists), `getprop` (symbol property lists), `gethash` (hash tables), `get-btree` (B-trees), and `car…cdddr`, `first…tenth`, `rest, nth` (lists). When structures are defined (Sec. 3.15) the system will generate getter functions and setter macros to access and update their fields.

The user can extend the update rules for `setf` by defining new setter macros. If a getter function call `(`*fn* `...)` is used for accessing an updatable location then the setter macro *sm* is associated with *fn* by calling `(setfmethod fn sm)`, see Sec. 3.16.

The following destructive system list functions are supported:

Function	Type	Description
`(attach x l)`	EXTFN	Similar to `(cons x l)` but *destructive*, i.e. the head of the list `l` is modified so that all pointers to `l` will point to the extended list after the attachment. Notice that this does *not* work if `l` is not a list, in which case `attach` is **not** destructive and creates a new list cell just like `cons`.
`(delete x l)`	*EXTFN	Remove *destructively* the elements in the list `l` that are `eq` to `x`. The result is the updated `l`. Notice that if `x` is the only remaining element in `l` the operation is **not** destructive and `nil` is returned.
`(dmerge x y fn)`	LAMBDA	Merge lists `x` and `y` *destructively* with `fn` as comparison function. For example: `(dmerge '(1 3 5) '(2 4 6) #'<) => (1 2 3 4 5 6)` The value is the merged list; the merged lists are destroyed. See also `merge`.
`(lconc hdr l)`	SUBR	Add elements in list`l` to list header `hdr`. The concatenated list in maintained in `(car hdr)` as for function `tconc` below.
`(nconc l...)`	*MACRO	*Destructive* concatenate the lists `l...` (i.e. destructive `append`) and return the concatenated list. This classical Lisp function is known to be error prone. It can also be slow when `l` is long. If possible, use `lconc` instead.
`(nconc1 l x)`	EXTFN	Add `x` to the end of `l` *destructively*, i.e. same as `(nconc l (list x))`. This function is known to be error prone. It can also be slow when `l` is long. If possible, use `tconc` instead.
`(nreverse l)`	*EXTFN	*Destructively* reverse the list `l`. The value is the reversed list. The original list `l` will be destroyed. Very fast reverse.
`(putf l i v)`	EXTFN	Set the value of the property indicator `i` in the property list `l` to `v`. If possible, use `(setf (getf l i) v)` instead.
`(rplaca l x)`	*EXTFN	*Destructively* replace the head of list `l` with `x`. If possible, use `(setf (car l) x)` instead.
`(rplacd l x)`	*EXTFN	*Destructively* replace the tail of list `l` with `x`. If possible, use `(setf (cdr l) x)` instead.
(setf p_1 v_1 p_2 v_2 ...)	*EXTFN	General macro for destructively updating variable bindings or getter data accesses, as explained above. Called *generalized variables* in CommonLisp terminology [1][2].
`(tconc hdr x)`	EXTFN	Efficient *tail concatenation* of elements at the end of a list. First a list header `hdr` is created by calling `(tconc)` without arguments. Then a successive new element `x` is concatenated to the tail of `hdr` each time `tconc` is called. The concatenated list is maintained in `(car hdr)`. See also `lconc`.

3.5. Strings

Strings (data type name STRING) represent text strings of arbitrary length. Trings are always immutable, i.e. they cannit be destructively modified. Notice that, unlike CommonLisp, C etc., there is no special data type for single characters, which are represented single character strings.

Strings containing the characters " or \ must precede these with the *escape character*, '\'. Examples of strings:

```
> (setf a "This is a string")
"This is a string"
> (setf b "String with string delimiter \" and the escape character \\")
"String with string delimiter \" and the escape character \\"
> (concat a b)
"This is a stringString with string delimiter \" and the escape character \\"
>
```

Function	Type	Description
`(char-int` *str*`)`	*EXTFN	The integer encoding the first character in string *str*.
`(concat` *str...*`)`	EXTFN	Coerce the arguments *str*... to strings and concatenate them.
`(explode` *str*`)`	EXTFN	A list of strings representing the characters in *str*.
`(int-char` *i*`)`	*EXTFN	The single character string encoded as integer *i*. `nil` is returned is there is no such character.
`(length` *str*`)`	*EXTFN	The number of characters in string *str*.
`(mkstring` *x*`)`	EXTFN	Coerce object *x* to a string.
`(string-capitalize str)`	*EXTFN	Capitalize string *str*.
`(string-downcase` *str*`)`	*EXTFN	Upper case string *str* .
`(string-upcase` *str*`)`	*EXTFN	Lower case string *str* .
`(string<` *s1 s2*`)`	*EXTFN	True if the string *s1* alphabetically precedes *s2*.
`(string=` *s1 s2*`)`	*EXTFN	True if the strings *s1* and *s2* are the same. The will also be `equal`.
`(string-left-trim` *ch str*`)`	*EXTFN	Remove the initial characters in *str* that also occur in *ch*.
`(string-like` *str pat*`)`	EXTFN	True if *pat* matches string *str*. The string *pat* is regular expression where: * matches any sequence of characters (zero or more) ? matches any character [SET] matches any character in the specified set, [!SET] or [^SET] matches any character not in the specified set.
`(string-like-i` *str pat*`)`	EXTFN	Case insensitive `string-like`.
`(string-pos` *str x*`)`	EXTFN	The character position of the first occurrence of the string *x* in *str*. The character positions are enumerated from 0 and up.
`(string-right-trim` *ch str*`)`	*EXTFN	Remove the trailing characters in *str* that also occur in *ch*.
`(string-trim` *ch str*`)`	*EXTFN	Remove those initial and trailing characters in *str* also occurring in *ch*.
`(stringp` *x*`)`	*EXTFN	True if *x* is a string.

`(substring p1 p2 str)`	EXTFN	The substring in *str* starting at character position *p1* and ending in *p2*. The character positions are enumerated from 0 and up.

3.6. Numbers

Numbers represent numeric values. Numeric values can either be long integers (data type name INTEGER) or double precision floating point numbers (data type name REAL). Integers are entered to the Lisp reader as an optional sign followed by a sequence of digits, e.g.

```
1234 -1234 +1234
```

Examples of legal floating point numbers:

```
1.1 1.0 1. -1. -2.1 +2.3 1.2E3 1.e4 -1.2e-20
```

The following system functions operate on numbers.

Function	Type	Description
`(+ x...)`	*EXTFN	Add the numbers *x*...
`(- x y)`	*EXTFN	Subtract *y* from *x*.
`(1+ x)`	*MACRO	Add one to number *x*.
`(1- x)`	*MACRO	Subtract one from number *x*.
`(* x...)`	*EXTFN	Multiply the numbers *x*...
`(/ x y)`	*EXTFN	Divide *x* with *y*.
`(acos x)`	*EXTFN	Arc cosine of *x*.
`(asin x)`	*EXTFN	Arc sine of *x*.
`(atan x)`	*EXTFN	Arc tangent of *x* .
`(ceiling x)`	*EXTFN	The smallest integer larger than or equal to *x*.
`(cos x)`	*EXTFN	Cosine of *x*.
`(decf x)`	*MACRO	Decrement the variable *x* with *delta*, default 1.
`(exp x)`	*EXTFN	Natural exponent e^x
`(expt x y)`	*EXTFN	Exponent x^y.
`(floor x)`	*EXTFN	The largest integer less than or equal to *x*.
`(frand low high)`	EXTFN	A floating point random number in interval `[low,high)`
`(incf x [delta])`	*MACRO	Increment the variable *x* with *delta*, default 1.
`(integerp x)`	*EXTFN	True if *x* is an integer.
`(log x)`	*EXTFN	The natural logarithm of *x*.
`(max x...)`	*EXTFN	Return the largest of the numbers *x*…
`(min x...)`	*EXTFN	Return the smallest of the numbers *x*…
`(minus x)`	EXTFN	Negate the number *x*. Same as `(- x)`.
`(minusp x)`	*LAMBDA	True if *x* is a number less than 0.
`(mod x y)`	*EXTFN	The remainder when dividing *x* with *y*. *x* and *y* can be both integers or floating point numbers.

`(numberp x)`	*EXTFN	True if *x* is a number.
`(plusp x)`	*LAMBDA	True if *x* is larger than 0.
`(random n)`	*EXTFN	A random integer in interval `[0,n)`.
`(randominit n)`	EXTFN	Generate a new seed for the random number generator.
`(round x)`	*EXTFN	Round *x* to closest integer.
`(sqrt x)`	*EXTFN	The square root of *x*.
`(sin x)`	*EXTFN	Sine of *x*.
`(tan x)`	*EXTFN	Tangent of *x*.
`(zerop x)`	*LAMBDA	True if *x* is equal to 0.

3.7. Logical Functions

In CommonLisp `nil` is regarded as false and any other value as true. The global variable T, bound to itself, is usually used for representing true. For example:

```
> (setf x t)           ← regarded as true
T
> (setf y nil)         ← regarded as false
NIL
> (setf z 1)           ← regarded as true
1
> (or x y z)
T                      ← X = T is the first true value
> (and x y z)
NIL                    ← Y is nil
> (or z x y)
1                      ← Z = 1 is the first true value
> (not y)
T                      ← Y is nil
> (not z)
NIL                    ← Z is 1 (i.e. true)
```

The following functions return or operate on logical values.

Function	Type	Description
`(< x y)`	*EXTFN	True if the number *x* is strictly less than *y*.
`(<= x y)`	*EXTFN	True if the number *x* is less than or equal to *y*.
`(= x y)`	*EXTFN	True if the numbers *x* and *y* are the equal.
`(> x y)`	*EXTFN	True if the number *x* is strictly greater than *y*.
`(>= x y)`	*EXTFN	True if the number *x* is greater than or equal to *y*.
`(and x...)`	*SPECIAL	Evaluate the forms *x*... and return `nil` when the first form evaluated to `nil` is encountered. If no form evaluates to `nil` the value of the last form is returned.
`(compare x y)`	EXTFN	Compare order of two objects. Return 0 if they are equal, -1 if less, and 1 if greater.
`(eq x y)`	*EXTFN	True if *x* and *y* are the same objects, i.e. having the same address in

		memory.
`(equal x y)`	*EXTFN	True if objects *x* and *y* are equivalent. **Notice** that, in difference to CommonLisp, arrays are equal if all their elements are equal, and equality can be defined for user defined data types as well.
`(evenp x)`	*LAMBDA	True if *x* is an even number.
`(neq x y)`	*EXTFN	Same as `(not (eq x y))`
`(oddp x)`	*LAMBDA	True if *x* is an odd number.
`(or x...)`	*SPECIAL	Evaluate the forms *x*... until some form does not evaluate to `nil`. Return the value of that form.
`(not x)`	*EXTFN	True if *x* is `nil`; same as `null`.

3.8. Arrays

Arrays (data type name ARRAY) in aLisp represent one-dimensional sequences. The elements of an array can be of any type. Arrays are printed using the notation #(e1 e2 …). For example:

```
> (setf a #(1 2 3))
#(1 2 3)
```

Arrays are allocated with the function `(make-array size)`. For example:

```
> (make-array 3)
#(NIL NIL NIL)
```

Notice that aLisp supports only one-dimensional arrays (vectors) while CommonLisp allows arrays of any dimensionality.

Adjustable arrays are arrays that can be dynamically increased in size. They are allocated with the function:

```
(make-array size :adjustable t)
```

Arrays can be enlarged with the function

```
(adjust-array array newsize)
```

Enlargement of adjustable arrays is incremental, and does not copy the original array. Non-adjustable arrays can be enlarged as well, but the enlarged array may or may not be a copy of the original one depending on its size. In other words, you have to rebind non-adjustable arrays after you enlarge them.

For example:

```
> (setf a (make-array 3))
#(NIL NIL NIL)
> (adjust-array a 6)
#(NIL NIL NIL NIL NIL NIL)
> a
#(NIL NIL NIL)
> (setf a (make-array 3 :adjustable t))
#(NIL NIL NIL)
> (adjust-array a 6)
#(NIL NIL NIL NIL NIL NIL)
> a
```

```
#(NIL NIL NIL NIL NIL NIL)
>
```

Function	Type	Description
`(adjust-array a newsize)`	*EXTFN	Increase the size of the array `a` to `newsize`. If the array is declared to be adjustable at allocation time it is adjusted in-place, otherwise an array copy may or may not be returned.
`(adjustable-array-p a)`	*EXTFN	True if `a` is an adjustable array.
`(aref a i)`	*MACRO	Access element `i` of the array `a`. Enumeration starts at 0. Unlike CommonLisp, only one dimensional arrays are supported. Can be updated with `setf`.
`(array-total-size a)`	*EXTFN	The number of elements in the one-dimensional array `a`, same as `(length a)`.
`(arrayp x)`	*EXTFN	True if `x` is an array (fixed or adjustable).
`(arraytolist a)`	EXTFN	Convert array `a` to a list.
`(concatvector x y)`	LAMBDA	Concatenate vectors `x` and `y`.
`(copy-array a)`	*EXTFN	Make a copy of array `a`.
`(elt a i)`	EXTFN	Same as `(aref a i)`. Can be updated with `setf`.
`(listtoarray l)`	EXTFN	Convert list `l` to a non-adjustable array.
`(length v)`	*EXTFN	The number of elements in vector `v`.
`(make-array size :initial-element v :adjustable aflag)`	*MACRO	Allocate a one-dimensional array of pointers with `size` elements. :INITIAL-ELEMENT specifies optional initial element values `v`. If :ADJUSTABLE is true an adjustable array is created; the default is a non-adjustable array.
`(push-vector a x)`	EXTFN	Add `x` to the end of array `a` by adjusting it.
`(seta a i v)`	EXTFN	Set element `i` in array `a` to `v`. Returns `v`. If possible, use `(setf (aref a i) v)` instead.
`(vector x...)`	*EXTFN	Make an array with elements `x`...

3.9. Memory areas

The datatype MEMORY)represents references to binary memory areas stored in main memory outside the database image. This is used for storing buffers and other binary data structures. The memory areas as *pinned* in memory meaning that their contents is not moved by the system. Memory area is not referenced from other objects will be automatically freed by the garbage collector. Memory area objects are saved as other Lisp object when ROLLOUT is called.

The following aLisp function handle memory areas:

`(malloc sz)`	EXTFN	Allocate a new memory area having `sz` bytes.
`(read-file-raw f)`	LAMBDA	Make a memory area object of the contents of file `f`.
`(realloc m sz)`	EXTFN	Increase the size of memory area object `m` to `sz`.
`(write-file f m)`	EXTFN	Write memory area `m` as the contents of file `f`.

3.10. Hash Tables

Hash tables (data type name HASHTAB) are unordered dynamic tables that associate values with aLisp objects as keys. Hash tables are allocated with:

```
(make-hash-table)
```

Notice that, unlike standard CommonLisp, no initial size is given when hash tables are allocated. Instead the system will automatically and incrementally grow (or shrink) hash tables as they evolve.

Elements of a hash table are accessed with:

```
(gethash key hashtab)
```

Elements of hash tables are updated with

```
(setf (gethash key hashtab) new-value)
```

Iteration over all elements in a hash table is made with:

```
(maphash (function (lambda (key val) ...)) hashtab)
```

Notice that comparisons of hash table keys in CommonLisp is by default using EQ and **not** EQUAL. Thus, e.g., two strings with the same contents do not match as hash table keys unless they are pointers to the same string. Normally EQ comparisons are useful only when the keys are symbols. To specify a hash table comparing keys with EQUAL (e.g. for numeric keys or strings) use

```
(make-hash-table :test (function equal))
```

Example:

```
> (setf ht1 (make-hash-table))
#[O HASHTAB 233568 10 2]
> (setf (gethash "hello" ht1) "world")
"world"
> (gethash "hello" ht1)
NIL
> (setf ht2 (make-hash-table :test (function equal)))
#[O HASHTAB 233600 10 2]
> (setf (gethash "hello" ht2) "world")
"world"
> (gethash "hello" ht2)
"world"
>
```

The following system functions operate on hash tables:

Function	Type	Description
`(clrhash ht)`	EXTFN	Clear all entries from hash-table `ht` and return the emptied table.
`(gethash k ht)`	*EXTFN	Get value with key `k` in hash table `ht`. Can be updated with `setf`.
`(hash-bucket-firstval ht)`	EXTFN	The value for the first key stored in hash table `ht`. What is the first key is undefined and depends on the internal hash function used.

`(hash-buckets ht)`	EXTFN	The number of hash buckets in hash table *ht*.
`(hash-table-count ht)`	*EXTFN	The number of elements stored in hash table *ht*.
`(make-hash-table :size s :test eqfn)`	*MACRO	Allocate a new hash table. The CommonLisp parameter :SIZE is ignored as the hash tables in aLisp are dynamic and scalable. The keyword parameter :TEST specifies the function to be used for testing equivalence of hash keys. :TEST can be `(function eq)` (default) or `(function equal)`.
`(maphash fn ht v)`	*EXTFN	Apply `(fn key value v)` on each pair of key and value in hash table *ht*.
`(puthash k ht v)`	EXTFN	Set the value stored in hash table *ht* under the key *k* to *v*. If possible, use `(setf (gethash k ht) v)` instead.
`(remhash k ht)`	EXTFN	Remove the value stored in hash table *ht* under the key *k*.
`(sxhash x)`	*EXTFN	Compute a hash code for object *x* as a non-negative integer. `(equal x y) => (= (sxhash x) (sxhash y))`.

3.11. Main memory B-trees

Main memory B-trees (datatype BTREE) are ordered dynamic tables that associate values with aLisp objects as keys. The interfaces to B-trees are very similar to those of hash tables. The main difference between B-trees and hash tables are that B-trees are ordered by the keys and that there are efficient tree search algorithms for finding all keys in a given interval. B-trees are slower than hash tables for equality searches.

B-trees are allocated with:

`(make-btree)`

Elements of a B-tree are accessed with:

`(get-btree key btree)`

`setf` is used for modifying accessed B-tree element.

For example:

```
> (setf bt(make-btree))
#[O BTREE 454360 33 2]
> (setf (get-btree "hello" bt) "world")
"world"
> (get-btree "hello" bt)
"world"
>
```

System functions operating on main memory B-trees:

Function	Type	Description
`(get-btree k bt)`	EXTFN	Get value associated with key *k* in B-tree *bt*. The comparison uses

		function `compare`. Can be updated with `setf`.
`(make-btree)`	EXTFN	Allocate a new B-tree.
`(map-btree` *`bt lower upper fn`*`)`		
	EXTFN	Apply Lisp function `(`*`fn key val`*`)` on each key-value pair in B-tree *`bt`* whose key is larger than or equal to *`lower`* and less than or equal to *`upper`*. If any of *`lower`* or *`upper`* is the symbol '*' it means that the interval is open in that end. If both *`lower`* and *`upper`* are '*' the entire B-tree is scanned.
`(put-btree` *`k bt v`*`)`	EXTFN	Set the value stored in the B-tree *`bt`* under the key *`k`* to *`v`*. If possible, use `(setf (get-btree` *`k bt`*`)` *`v`*`)` instead. If *`v`* is `nil` the element is deleted.

3.12. Functional arguments and dynamic forms

Variables bound to functions or even entire expression can be invoked or evaluated by the system. Functional arguments (higher order functions) provide a very powerful abstraction mechanism that can replace many control structures in conventional programming languages. The map functions in Sec. 3.12.5 are examples of elaborate use of functional arguments.

The simplest case for functional arguments is when a function is passed as arguments to some other function. For example, assume we want to make a max function, `(sum2` *`x y fn`*`)` that calls the functional *`fn`* with arguments *`x`* and *`y`* and then adds together the two results (i.e. sum2 = fn(x) + fn(y)):

```
> (defun sum2 (x y fn)
(+ (funcall fn x)(funcall fn y))) ← The system function FUNCALL calls FN
SUM2
> (sum2 1 2 (function sqrt))       ← sqrt(1) + sqrt(2)
2.41421
```

In CommonLisp, the system function `funcall` must be used to call a function bound to a functional argument. Also notice that the special form `function` should be used when passing a functional argument, to be explained next.

3.12.1. Closures

In the example the special form `function` is used when passing a functional argument. **Notice** that `quote` should **not** be used when passing functional arguments. The reason is that otherwise the system does not know that the argument is a function. This matters particularly if the functional argument is a lambda expression. Consider a function to compute $X^N + Y^N$ using `sum2`:

```
> (defun sumpow (x y pow)
        (sum2 x y (function               ← FUNCTION must be used here
                     (lambda (z)          ← lambda expression = anonymous
function
                       (expt z pow)))))   ← POW is free variable in lambda
SUMPOW
> (sumpow 1 2 2)
5
```

Free lambda expressions as this one are very useful when passing free variables like `pow` into a functional argument. Now, let's see what happens if `quote` was used instead of `function`:

```
> (defun sumpow (x y pow)
      (sum2 x y (quote                          ← This is wrong!
                 (lambda (z)
                  (expt z pow)))))
(SUMPOW REDEFINED)
Suspicious use of QUOTE rather than FUNCTION: (QUOTE (LAMBDA (Z) (EXPT Z
POW))) in SUMPOW
SUMPOW
lisp 1> (sumpow 1 2 2)
ERROR! Unbound variable: POW
When evaluating: POW
(FAULTEVAL BROKEN)
In SUM2 brk>:r
```

As you can see, the system warns that `quote` is used instead of `function`. Notice that the variable `pow` becomes unbound when `sumpow` is called. The reason is that `quote` returns its argument unchanged while `function` makes a *closure* of its argument if it is a lambda expression. A closure is a datatype that holds a function (lambda expression) together with the local variables bound where it is called. In our example, the local variable `pow` is bound when `sum2` is called inside `sumpow`.

WARNING: Closures in saved images are invalid and can crash the system. This means that you should not bind closures to global variables and then save the image. The bound closures will be invalid when the image is used.

3.12.2. Variadic function calls

The macro `funcall` does not work if we don't know before run time the number of arguments of the function to call. In particular `funcall` cannot be used if we want to call a variadic function like + (plus). What we need is a way to construct a dynamic argument list before we call a function. For this the system macro `apply` is used. For example, assume we define a function `(combinel` *`x y fn`*`)` that applies *`fn`* on the elements of lists *`x`* and *`y`* and combines the results also using *`fn`*:

```
> (defun combinel (x y fn)
    (funcall fn
      (apply fn x)
      (apply fn y)))
COMBINEL
> (combinel '(1 2 3) '(4 5 6) (function +))
21
```

In this case we have to construct the arguments as a list to the inner function applications, and therefore `apply` has to be used in the definition of `combinel`. The function could also have been written as:

```
> (defun combinel (x y fn)
    (apply fn
     (list
       (apply fn x)
       (apply fn y))))
(COMBINEL REDEFINED)
COMBINEL
```

```
> (combinel '(1 2 3) '(4 5 6) (function +))
21
```

3.12.3. Dynamic evaluation

The most general way to execute dynamic expressions in Lisp is to call the system function `eval`. It takes as argument any Lisp form (i.e. expression) and evaluates it. For example:

```
> (setf a 1)
1
> (eval '(list a))
(1)
> (eval (list a))      ← This fails because we are trying to evaluate the form
(1)
ERROR! Undefined function: 1
When evaluating: (1)
(FAULTEVAL BROKEN)
In *BOTTOM* brk>:r
    > (list a)           ← This gives the same result as the aLisp REPL
calls eval
(1)
```

The function `eval` is actually very seldomly used. It is useful when writing Lisp programming utilities, like e.g. the aLisp REPL itself or remote evaluation (Sec. 5.3.3). **Notice** that you should avoid using `eval` unless you really need it, as the code executed by `eval` is not known until run-time and this is unpredictable, unsafe and prohibits compilation and program analysis. If possible, use `funcall` or `apply` instead. In most other cases macros (Sec. 3.14) replace the need for `eval` while at the same time producing compilable and analysable programs.

3.12.4. System functions for run-time evaluation

Function	Type	Description
`(apply` *`fn arg1`*`)`	*MACRO	Apply the function *`fn`* on the list of arguments *`arg1`*. The macro `apply` is used to call a function where the argument list is dynamically constructed with varying arity, a *variadic function call*.
`(apply` *`fn`* $arg_1 \ldots arg_k$`)`	*MACRO	The macro `apply` can have more than two arguments $arg_1 \ldots arg_k$, k>=2. In this case the dynamic argument list is constructed by prepending arg_k with $arg_1 \ldots arg_{k-1}$, i.e. the call is the same as `(apply` *`fn`* `(list*` $arg_1 \ldots arg_k$`))`.
`(applyarray` *`fn a`*`)`	EXTFN	Apply the Lisp function *`fn`* on the arguments in the array *`a`*.
`(eval` *`form`*`)`	*EXTFN	Evaluate *`form`*. **Notice** that unlike CommonLisp, the form is evaluated in the lexical environment of the `eval` call.
`(f/l` *`fn args form`*`...)`	MACRO	`(f/l (x) form...) <=> (function(lambda(x) form...))`. This is equivalent to the CommonLisp read macro (also supported): `#'(lambda (x) form...)`.
`(funcall` *`fn arg1`*`...)`	*MACRO	Call function *`fn`* with arguments *`arg1`*... The macro `funcall` is used when the called function *`fn`* is not known until run-time (e.g. bound to a variable).

`(function fn)` *SPECIAL Make a closure of the function *fn*.

3.12.5. Map functions

Map functions are functions and macros taking other functions as arguments and applying them repeatedly on elements in lists and other data structures. Map functions provide a general and clean way to iterate over data structures in a functional programming style. They are often a good alternative to the more conventional iterative statements (Sec. 3.13.3). They are also usually a preferred alternative to recursive functions as they don't eat stack as recursive functions do.

The classical Lisp map function is `mapcar`. It applies a function on every element of a list and returns a new list formed by the values of the applications. For example:

```
> (mapcar (function 1+) '(1 2 3))
(2 3 4)
```

The function `mapc` is similar, but does not return any value. It is useful when the applied function has side effects. For example:

```
> (mapc (function print) '(1 2 3))
1
2
3
NIL                                    ← mapc always returns nil
```

In CommonLisp the basic map functions may take more than one argument to allow parallel iteration of several lists. For example:

```
> (mapcar (function +)
          '(1 2 3) '(10 20 30))
(11 22 33)
```

Lambda expressions are often useful when iterating using map functions. For example:

```
> (defun rev2 (a b)
         (let (ra rb)
           (mapc #'(lambda (x y)
                      (push x ra)
                      (push y rb))
                 a b)
           (list ra rb)))
REV2
> (rev2 '(1 2 3) '(a b c))
((3 2 1) (C B A))
```

The following system map functions are available in aLisp:

Function	Type	Description

`(member-if` *fn l*`)`	*EXTFN	The function *fn* is applied on each element in list *l*. If *fn* returns non-nil for some element in *l* then `member-if` will return the corresponding tail of *l*
`(mapc fn` *l*...`)`	*MACRO	Apply *fn* on each of the elements of the lists *l*... in parallel.
`(mapcan` *fn l*...`)`	*MACRO	Apply *fn* on each of the elements of the lists *l*... in parallel and `nconc` together the results.
`(mapcar` *fn l*...`)`	*MACRO	Apply *fn* on each of the elements of the lists *l*... in parallel and build a list of the results.
`(mapl` *fn l*...`)`	*MACRO	Apply *fn* on each tail of the lists *l*...
`(every` *fn l*...`)`	*MACRO	True if *fn* returns non-nil result when applied on every element in the lists *l*... in parallel.
`(notany` *fn l*...`)`	*MACRO	True if *fn* applied on each element in the lists *l*... in parallel never returns true.
`(reduce` *fn l*`)`	*LAMBDA	Aggregate the values in *l* by applying the binary function *fn* pairwise between the elements in *l*. If *l* has a single element it is returned without applying *fn*. *nil* is returned if *l* is *nil*.
`(remove-if-not` *fn l*`)`	*EXTFN	The subset of the list *l* for which the function *fn* returns true.
`(remove-if` *fn l*`)`	*EXTFN	The subset of the list *l* for which the function *fn* returns false.
`(some` *fn l*...`)`	*MACRO	True if *fn* applies on each element in the lists *l*... in parallel returns true for some application.

3.13. Control Structures

This subsection describes system functions, macros, and special forms used for implementing syntactic sugar and control structures in aLisp.

3.13.1. Compound expressions

The functions `progn`, `prog1`, and `prog2` are used for forming a single form out of several forms. This makes sense only if some of the forms have side effects. For example:

```
> (progn (print "A") "B")
"A"
"B"                          ← Value of progn is value of last argument
> (prog1 (print "A") "B")
"A"
"A"                          ← Value of prog1 is value of first argument
```

Compound expressions can also be formed by lambda and `let` expressions and other control structures described in the next section.

Function	Type	Description
`(prog1` *x*...`)`	*EXTFN	The value of the first form in *x*...
`(prog2` *x*...`)`	*LAMBDA	The value of the second form in *x*...
`(progn` *x*...`)`	*SPECIAL	The value of the last form in *x*...

3.13.2. Conditional expressions

Conditional expressions are special forms that evaluate expressions conditionally depending on the truth value of some form. The classical Lisp conditional expression is `cond`. For example:

```
> (setf x 1
        y 2
        z nil)
NIL
> (cond (x)
        (t y))
1
1> (cond (z (print "NO"))
         (y (print "YES") 5)
         (t (print "NO")))
"YES"
5
```

The following conditional statements are available in aLisp:

Function	Type	Description
`(case` *`test`* `(`*`when then`*...`)` ...`(otherwise` *`default`*...`)`		

	*MACRO	For example:
		`(case (+ 1 2)`
		`      (1 'HEY)`
		`      ((2 3) 'HALLO)`
		`      (otherwise 'DEFAULT))`
		`=> HALLO`
		The form *`test`* is evaluates and successively compared with each *`when`* expression until a match is found. Then the corresponding forms *`then`*... forms are evaluated, and the last one is returned as the value of `case`. Atomic *`when`* expressions match if they are EQ to the value, while lists match if the value of `test` is member of the list. If no *`when`* expression matches the forms *`default`*... are evaluated and returned as the value of `case`. If no `otherwise` clause is present the default result is `nil`.

`(cond (`*`test form`*...`)`...`)`	*SPECIAL	Classical Lisp conditional execution of forms.
`(if` *`pred then else`*`)`	*SPECIAL	Evaluate *`then`* if *`pred`* evaluates to true otherwise evaluate *`else`*.
`(selectq` *`test`* `(`*`when then`*...`)`... *`default`*`)`		

	SPECIAL	For example:
		`(selectq (+ 1 2)`
		`         (1 'hey)`
		`         ((2 3) 'hallo)`
		`          'default)`
		`  => hallo`
		Same as
		`(case` *`test`* `(`*`when then`*...`)`... `(otherwise` *`default`*`))`

`(unless` *`test form`*...`)`	*MACRO	Evaluate *`form`*... if *`test`* is false.
`(when` *`test form`*...`)`	*MACRO	Evaluate *`form`*... if *`test`* is true.

3.13.3. Iterative statements

As in other programming languages Lisp provides iterative control structures, normally as macros. However, in most cases map functions (Sec. 3.12.5) provide the same functionality in a cleaner and often more general way.

Function	Type	Description
`(do` *`inits endtest form`*`...)`	*MACRO	General CommonLisp iterative control structure [1][2]. Loop can be terminated with `(return` *`val`*`)` in addition to the *`endtest`*.
`(do*` *`inits endtest form`*`...)`	*MACRO	As `do` but the initializations *`inits`* are done in sequence rather than in parallel.
`(dolist (`*`x l`*`)` *`form`*`...)`	*MACRO	Evaluate *`form`*... for each element *`x`* in list *`l`*. Same as `(mapc #(lambda (`*`x`*`)` *`form`*`...)` *`l`*`)`
`(dotimes (`*`i n`* `[`*`res`*`])` *`form`*`...)`	*MACRO	Evaluate *`form`*... *`n`* times varying *`i`* 0 to *`n-1`*. The optional form *`res`* returns the result of the iteration. In *`res`* the variable *`i`* is bound to the number of iterations made.
`(loop` *`form`*`...)`	*MACRO	Evaluate *`form`*... repeatedly. The loop can be terminated with the result *`val`* returned by calling `(return` *`val`*`)`.
`(return [`*`val`*`])`	*LAMBDA	Return value *`val`* form the block in which `return` is called. A block can be a `prog-let, prog-let*, dolist, dotimes, do, do*, loop,` or `while` expressions.
`(rptq` *`n form`*`)`	SPECIAL	Evaluate *`form n`* times. Recommended for performance measurements in combination with the macro `time`.
`(while` *`test form`*`...)`	MACRO	Evaluate the forms *`form`*... while *`test`* is true or until `return` is called.

3.13.4. Non-local returns

Non-local returns allows to bypass the regular function application order. The classical Lisp forms for this are `catch` and `throw`. The special form `(catch` *`tag form`*`)` evaluates *`tag`* to a *catcher* which must be a symbol. Then *`form`* is evaluated and if thereby the function `(throw` *`tag value`*`)` is called somewhere with the same catcher then *`value`* is returned. If `throw` is not called the value of *`form`* is returned. For example:

```
> (defun foo (x)(catch 'foo-catch (fie (+ 1 x))))
FOO
> (defun fie (y)(cond ((= y 2)(throw 'foo-catch -1))
                      (t y)))
FIE
> (foo 1)
-1
> (foo 2)
3
```

A related subject is how to catch errors. In particular `unwind-protect` is the general mechanism to handle any kind of non-local return and error trapping. This is described in Sec. 6.1.

Function	Type	Description
`(catch tag form)`	*SPECIAL	Catch calls to `throw` inside *form* matching *tag*.
`(throw tag val)`	*EXTFN	Return *val* as the value of a previous call to `catch` with the same *catcher tag* having called `throw` directly or indirectly.

3.14. Macros

Lisp macros provide a way to extend Lisp with new control structures and syntactic sugar. Because programs are represented as data in Lisp it is particularly simple to make Lisp programs that transform other Lisp programs. Macros provide the hook to make such code transforming programs available as first class objects. A macro should be seen as a rewrite rule that takes a Lisp expression as argument and produces another equivalent Lisp expression as result. For example, assume we want to define a new control structure `for` to implement for loops, where e.g. `(for i 2 10 (print i))` prints the natural numbers from 2 to 10. The macro `for` can be defined as:

```
> (defmacro for (var from to do)
      (subpair '(_var _from _to _do)     ← _var, _from, to, and _do are
substituted
                (list var from to do)    ← with these actual values
             '(let ((_var _from))        ← This is the code skeleton
               (while (<= _var _to)
                      _do
                      (incf _var)))))
FOR
lisp 1> (for i 2 4 (print i))             ← Macros are expanded by interpreter
2
3
4
NIL                                       ← Value of for
```

When defining macros as in the example there is usually a code skeleton in which one substitutes elements with actual argument expression. In the example we use `subpair` to do the substitution. A more convenient CommonLisp facility to define code skeleton is to use *back quote* (‘`’), which is a variant of `quote` where subexpressions can be marked for evaluation. Using backquote `for` could have been written as:

```
> (defmacro for (var from to do)
          `(let ((,var ,from))
               (while (<= ,var ,to)
                      ,do
                     (incf ,var))))
FOR
> (for i 2 4 (print i))
2
3
4
NIL
```

The backquote character ‘`’ marks the succeeding form *x* to be back quoted. The form *x* is substituted with a new expression by recognizing the symbols ’,’ (comma) and ’,@’ (comma at sign) in *x* as special markers. A comma in *x* is replaced with the value of the evaluation of the form following the comma. The form following a comma- at-sign is evaluated and ’spliced’ into the list.

For example, after evaluating

```
(setf a '(1 2 3)
      b '(3 4 5))
```

then

```
'(a (b ,a ,@ b)) -> (a (b (1 2 3) 3 4 5))
```

Macros can be debugged like any other Lisp code (Sec. 7). In particular it might be interesting to find out how a macro transform a given call. For this the system functions `macroexpand` and `macroexpand-all` can be used, normally in combination with pretty-printing the macroexpanded code with `ppv` (Sec 5). For example:

```
> (macroexpand '(for i 2 4 (print i)))
((LAMBDA (I) (WHILE (<= I 4) (PRINT I) (INCF I))) 2)
> (ppv (macroexpand '(for i 2 4 (print i))))
((LAMBDA (I)                              ← ppv makes more readable printing of
code
  (WHILE
     (<= I 4)
     (PRINT I)
     (INCF I)))
 2)
NIL
   > (ppv (macroexpand-all '(for i 2 4 (print i))))
   ((LAMBDA (I)
     (CATCH 'PROG-RETURN
      (INT-WHILE
        (<= I 4)
        (PRINT I)
        (SETQ I
           (+ I 1)))))
 2)
   NIL
```

The function `macroexpand` expands the top level of the form while `macroexpand-all` expands all macros in the form.

Notice that macros *should not have side effects*! They should be side effect free Lisp code that transforms one piece of code to another equivalent piece of code.

Macros can be used for defining functions whose arguments are always quoted. One such function is `(pp` *fn*$_1$…*fn*$_k$`)` that pretty-prints function definitions, for example:

```
> (pp pp)
(DEFMACRO PP (&REST FNS)
  "Pretty prints function definitions"
  (LIST 'PPF
     (LIST 'QUOTE FNS)))
(PP)
> (macroexpand '(pp pp))                 ← Let's look at how (pp pp) is
rewritten
(PPF (QUOTE (PP)))                       ← The function ppf does the pretty-
printing
```

MACROs are very efficient in aLisp because the first time the interpreter encounters a macro call it will modify the code and replace the original form with the macro-expanded one (just-in-time expansion). Thus a macro is normally evaluated only once. The definition of a macro is a regular function definition with a flag set indicating that its definition is a macro.

The following functions are useful when defining macros:

Function	Type	Description
`(andify l)`	LAMBDA	Make an `and` form of the forms in *l*.
`(bquote x)`	MACRO	`bquote` implements CommonLisp's read macro ' (back-quote).
`(defmacro name args form...)`	*SPECIAL	Define a new MACRO.

`(kwote x)`	EXTFN	Make *x* a quoted form.
		For example,
		`(kwote t) => T`
		`(kwote 1) => 1`
		`(kwote 'a) => (QUOTE A)`
		`(kwote '(+ 1 2)) => (QUOTE (+ 1 2))`

`(kwoted x)`	EXTFN	True if *x* is a quoted form. For example:
		`(kwoted 1) => T`
		`(kwoted '(quote (1))) => T`
		`(kwoted '(1)) => nil`

`(macro-function fn)`	*EXTFN	The function definition of *fn* if it is a macro; otherwise return `nil`.
`(macroexpand form)`	*EXTFN	If *form* is a macro return what it rewrites *form* into; otherwise *form* is returned unchanged.
`(macroexpand-all form)`	LAMBDA	Macroexpand *form* and all subforms in it.
`(prognify forms)`	LAMBDA	Make a single form from a list of forms *forms*.

3.15. Defining structures

aLisp includes a subset of the structure definition package of CommonLisp. The structures are implemented in aLisp using fixed size arrays. You are recommended to use structures instead of lists when defining data structures because of their more efficient and compact representation.

A new structure *s* is defined with the macro (defstruct *s* $field_1$ $field_2$...), for example:

```
> (defstruct person name address)
PERSON
```

The macro `defstruct` defines a new structure *s* with fields specified by $field_1$ $field_2$... It generates a number of macros and functions to create and update instances of the new structure. New instances are created with the generated macro:

(make-s :$field_1$ $value_1$:$field_2$ value2 ...)

For example:

```
> (setf p (make-person :name "Tore" :address "Uppsala"))
```

```
#(PERSON "Tore" "Uppsala")
```

The fields of a structure are updated and accessed using *accessor functions* generated for each field:

```
(S-FIELDi S)
```

for example:

```
> (person-name p)
"Tore"
```

Fields are updated by using `setf` with accessor functions:

```
(setf (s-field s) val)
```

For example:

```
> (setf (person-name p) "Kalle")
"Kalle"
> (person-name p)
"Kalle"
```

An object *x* can be tested to be a structure of type *s* using the generated function:

(*s*-p *x*)

For example:

```
> (person-p p)
T
```

3.16. Miscellaneous functions

Function	Type	Description
`(advise-around` *`fn code`*`)`	LAMBDA	Wrap the body of function *`fn`* with form *`code`* where each * is substituted with the original body of *`fn`*. If *`fn`* is a LAMBDA function the formal parameters of *`fn`* are free in *`code`*. If *`fn`* is an EXTFN the variable `!args` is bound in *`code`* to a list of the actual arguments. The function `advise-around` allows existing code to be instrumented without changing it, so called aspect oriented programming. Several system components, e.g. `trace, break, profile-functions` are defined using `advise-around`.
`(checkequal` *`text`* `(`*`form value`*`)`...)	SPECIAL	Lisp regression testing facility. The *`text`* is first printed. Then each *`form`* is evaluated and its result compared with the value of the evaluation of the corresponding *`value`*. An error message is printed if some evaluation of some *`form`* is not EQUAL to the corresponding *`value`*. Furthermore a regression test banner is printed if some `checkequal` test has failed.
`(declare ...)`	*MACRO	Dummy defined in aLisp for compatibility with CommonLisp.
`(evalloop)`	EXTFN	Enter the aLisp REPL. Return to caller when function `(exit)` is called.

`(exit [`*`rc`*`])`	EXTFN	Return from the aLisp REPL to the program that called it. In a stand-alone system `exit` is equivalent to `quit`. When aLisp is called from some other system `exit` will pass the control to the calling system.
`(frameno)`	EXTFN	The frame number of the top frame of the stack.
`(identity `*`x`*`)`	*LAMBDA	The identity function.

`(memo-function `*`fn`*`)`	LAMBDA	Make Lisp function *`fn`* into a *memo function*. This means that the system caches the arguments of *`fn`* when it is called so that it does not execute the function body when it is called repeatedly, to speed up execution. `(clear-memo-function `*`fn`*`)` clears the cache. For example:
		`(memo-function`
		` (defun fibonacci (x)`
		`   (if (< x 2) 1 (+ (fibonacci (- x 1))`
		`                    (fibonacci (- x 2))))))`

`(imagesize `*`size`*`)`	EXTFN	Extend the system's database image size to *`size`*. If *`size`* = `nil` the current image size is returned. The image is normally extended automatically by the system when memory is exhausted. However, the automatic image expansion may cause a short halting of the system while the OS is mapping more virtual memory pages. By using `imagesize` these delays can be avoided.
`(quit [`*`rc`*`])`	*EXTFN	Quit aLisp with optional return code *`rc`*. If it is embedded in another system it will terminate as well.
`(rollout `*`file`*`)`	EXTFN	Save the aLisp memory area (database image) in *`file`*. It can be restored by specifying *`file`* on the command line the next time aLisp is started.
`(setfmethod `*`access-fn setf-macro`*`)`	LAMBDA	Define setter macro for a getter function. For example: `(setfmethod 'gethash` ` '(lambda (place val)` `     `(puthash ,(second place) ,(third place)` ` ,val)))`
`(stacktop `*`size slack`*`)`	EXTFN	Change or obtain the size of Lisp's variable binding stack. *`size`* is the total stack size in stack frames, while *`slack`* indicates the number of stack frames that has to remain when an error happens. The parameter *`slack`* allows the break loop to work even when stack overflow happens, as it provides some remaining stack space when an error happens. The slack should be at least 300 (initial setting) for the break loop to work. The current setting is obtained as a pair by calling `(stacktop)` without arguments. Notice that the *`size`* can never be increased beyond the initial setting assigned when the system is started up. The initial stack size can be set in C by assigning the global C variable `a_stacksize` before the system is initialized. `stacktop` allows setting a smaller stack size than the initial one to prevent the system from crashing because of C stack overflow, which may happen in, e.g., DLLs where the calling system may have allocated a to small C stack size.
`(type-of `*`x`*`)`	*EXTFN	The name of the datatype of object *`x`*.
`(unwrap-fn `*`fn`*`)`	LAMBDA	Restore the original code for advised function *`fn`*. See `advise-`

around.

3.17. Hooks

Hooks are lists of Lisp forms executed at particular states of the system. Currently there is an *initialization hook* evaluating forms just after the system has been initialized, and a *shutdown hook* evaluating forms when the system is terminated.

To register a form to be executed just after the database image has been read call:

```
(register-init-form form [where])
```

The Lisp expression `form` is inserted into a list of forms stored in the global variable `after-rollin-forms`. The forms are evaluated by the system just after a database image has been read from the disk. If `where` = `first` the form is added in front of the list; otherwise it is added to the end. For example:

```
> (register-init-form '(formatl t "Welcome!" t))
OK
```

To register a form to be evaluated when the system is exited call:

```
(register-shutdown-form form [where])
```

The Lisp expression `form` is evaluated just before the system is to be exited using `(quit)`. The shutdown hook will **not** be executed if `(exit)` is called. The global variable `shutdown-forms` contains a list of the shutdown hook forms. For example:

```
> (register-shutdown-form '(formatl t "Goodbye!" t))
OK
```

The hooks are saved in the database image. For example, given that we have registered to above two hooks we can do the following:

```
> (rollout "myimage.dmp")   ← Save the database image in a file
T
> (quit)
Goodbye!                    ← The shutdown hook is evaluated.

sa.engine myimage.dmp ← Start sa.engine with the saved image
Database image: myimage.dmp
Welcome!                    ← The initialization hook is evaluated.
Release 2, v11, 64 bits
[sa.engine] 1>
```

4. Time Functions

Time points are represented in aLisp by the datatype TIMEVAL. It represents UTC time points as number of microseconds since EPOC (midnight GMT 1970-01-01). A TIMEVAL object has two components, *sec* and *usec*, representing seconds since EPOC and micro seconds beyond the second, respectively. A TIMEVAL object is printed as #[T *sec usec*], e.g. #[T 1569255397 470000] . Time differences are usually represented as seconds represented as floating point numbers.

The following Lisp functions operate on time points:

Function	Type	Description
`(clock)`	EXTFN	Compute the number of wall clock seconds spent so far during the run.
`(daylight-savingp)`	EXTFN	True if daylight saving time is active.
`(local-time [`*`tval`*`])`	EXTFN	The location dependent local UTC time string of TIMEVAL object *`tval`*. Current local wall time if *`tval`* omitted.
`(mktimeval `*`sec usec`*`)`	EXTFN	Create a new TIMEVAL object.
`(parse-utc-time `*`str`*`)`	EXTFN	Convert a UTC time string *`str`* into a TIMEVAL object.
`(timevalp `*`tval`*`)`	EXTFN	True if *`tval`* is a TIMEVAL object.
`(timeval-sec `*`tval`*`)`	EXTFN	The number of seconds since EPOC for a TIMEVAL object *`tval`*.
`(timeval-usec `*`tval`*`)`	EXTFN	The number of microseconds after the `timeval-sec` part of a TIMEVAL object *`tval`*.
`(gettimeofday)`	EXTFN	The TIMEVAL object representing the present wall time.
`(set-timer `*`fn period`*`)`	EXTFN	The function `set-timer` starts a *timer function*, which is a Lisp function called regularly by the system kernel. *`period`* specifies the minimal interval in seconds between successive calls to the function *`fn`*. In practice it will not be called that often, depending on OS scheduling and other activities. The timer function is terminated if it causes an error signal (Sec. 6). The statistical profiler (Sec. 7.4.1) is based on a timer function.
`(sleep `*`sec`*`)`	EXTFN	The system function `sleep` makes the system sleep for *`sec`* seconds. It can be interrupted with CTRL-C. *`sec`* is specified as a real number.
`(timeval-shift `*`tval sec`*`)`	EXTFN	Construct a new TIMEVAL object by adding *`sec`* seconds to *`tval`*.
`(timeval-span `*`tv1 tv2`*`)`	EXTFN	The difference in seconds between TIMEVAL *`tv2`* and *`tv1`*.
`(utc-offset)`	EXTFN	The setting in the computer of the offset in seconds from UTC, taking into account both the time zone and eventual daylight saving time.
`(utc-time [`*`tval`*`])`	EXTFN	The location independent UTC time string of TIMEVAL object *`tval`*. Current UTC wall time if *`tval`* omitted.

5. Input and Output

The I/O system is based on various kinds of *byte streams*. A byte stream is a datatype with certain attributes allowing its instances to be supplied as argument to the basic Lisp I/O functions, such as `print` and `read`. Examples of byte streams are: i) *file streams* (type STREAM) for terminal/file I/O, ii) *text streams* (type TEXTSTREAM) for reading and writing into text buffers, and iii) *socket streams* (type SOCKET) for communicating with other sa.engine/aLisp systems. The storage manager allows the programmer to define new kinds of byte stream. A byte stream argument `nil` or T represents *standard input* or *standard output* (i.e. the console).

Byte streams normally have functions providing the following operations:

- Open a new byte stream, e.g. `(openstream file mode)` creates a new file stream.
- Print objects to byte stream *str*. For example, `(print form str)` prints a form to byte stream *str* open for output. All kinds of objects inside the form are marshalled as S-expressions.
- Read bytes from byte stream *str*. For example, `(read str)` reads one form from a byte stream open for input and will thereby read bytes from the stream buffer. Notice that `print` and `read` are compatible so that a printed form will be recreated by `read`.
- Send the contents of byte stream *str* to its destination by calling `(flush str)`. This will empty the buffer associated with most kinds of byte streams.
- Close byte stream *str* by calling `(closestream str)`.

The following functions work on any kind of byte streams, including standard input or output:

Function	Type	Description
`(closestream str)`	EXTFN	Close byte stream *str*.
`_deep-print_`	GLOBAL	Normally the contents of fixed size arrays and structures are printed by `print` etc. This allows I/O of such datatypes. However, when `_deep-print_` is set to `nil` the contents of arrays and structures are *not* printed. Good when debugging large or circular structures. Default value of `_deep-print_` is T.
`(dribble [file])`	*LAMBDA	Log both standard input and output to *file*. The logging stops and the file is closed by calling `(dribble)`. Both standard input and output is printed on the console. Notice that only the user interaction with the system is redirected; i.e. printing to standard output using the basic C I/O routines (e.g. *printf* in C) is not redirected by `dribble`. To redirect all standard output use the function `redirect-basic-stdout`.

`(formatl str form...)`	LAMBDA	This function is a simple replacement of some of the functionality of the function `format` in CommonLisp [1][2]. It prints the values of the forms *form*... into byte stream *str*. A marker T among *form*... indicates a line feed while the string `"~PP"` makes the next element pretty-printed. For example:
		`(formatl t "One: " 1 ", two: " 2 t)`
		prints the line:
		`One: 1, two: 2`

`(fresh-line [str])`	*EXTFN	Print a line feed into *str* unless the output position is just after a new line.
`(pp fn...)`	MACRO	Pretty-print the functions or variables *fn*... on standard output. **Notice** that arguments of `pp` are not quoted. For example: `(pp ppv ppf)`.
`(pps s [str])`	LAMBDA	Pretty-print expression *s* in optional stream *str* or standard output

		and return *s*.
`(ppv s)`	LAMBDA	Pretty-print s on standard output and return `nil`.
`(prin1` *s* `[`*str*`])`	*EXTFN	Print the object *s* into byte stream *str* with escape characters and string delimiters inserted.
`(princ` *s* `[`*str*`])`	*EXTFN	Print the object *s* into byte stream *str* without escape characters and string delimiters.
`(princ-charcode` *n* `[`*str*`])`	EXTFN	Prints ASCII character number *n* into byte stream *str*.
`(print` *s* `[`*str*`])`	*EXTFN	Prints the object *s* into byte stream *str* so that it can be later read with `(read` *str*`)` to produce an object EQUAL to *s*. Same `(prin1 s str)` followed by `(terpri str)`.
`(printl` *l*...`)`	LAMBDA	Print the objects *l*... as a list on standard output.
`(read [`*str*`])`	*EXTFN	Read expression from byte stream *str*. IF *str* is a string, the system reads an expression from the string. For example: `(read "(A B C)") => (A B C)` See also `with-textstream`.
`(read-bytes` *n* `[`*str*`])`	EXTFN	Read *n* bytes from byte stream *str* as a string.
`(read-charcode [`*str*`])`	EXTFN	Read one byte from byte stream *str* and return it as an ASCII integer.
`(read-line [`*str eolchar*`])`	*EXTFN	Read the characters up to just before the next end-of-line character as a string. If *eolchar* is specified it is used as terminating character instead of end-of-line.
`(read-token [`*str delims brks strnum nostrings*`])`	EXTFN	Read the next token from byte stream *str*. *delims* are character used as delimiters between tokens, default: blank, tab, newline, carriage return. *brks* are break character, i.e. they become their own tokens, default “()[]\”’;`,”.  If *strnum* is `nil` numbers are parsed into numbers, otherwise no special treatment of numeric characters.  If *nostrings* is `nil` the system will interpret strings enclosed with “ as in Lisp (C, Java, etc), otherwise no special treatment of “.
`(spaces` *n* `[`*str*`])`	LAMBDA	Print *n* spaces into byte stream *str*.
`(terpri [`*str*`])`	*EXTFN	Print a line feed into byte stream *str*.
`(textual-streamp` *str*`)`	EXTFN	Returns true if the byte stream is open in *textual mode*, e.g. “rb” or “wb”. When binary objects are printed on stream in textual mode the convert binary fields to some textual representation, usually hexadecimal strings.
`(type-reader` *tpe fn*`)`	EXTFN	Define the lisp function `(fn` *tpe args stream*`)` to be a *type reader* for objects printed as `#[`*tpe x*...`]`. The type reader is evaluated by the aLisp reader when the pattern is encountered in an input byte stream. The parameter *tpe* is the type tag, *args* is the list of argument of the read object (i.e. *x*...), and *str* is the byte stream read from.
`(unread-charcode` *c str*`)`	EXTFN	Put character *c* back into byte stream *str*.

(y-or-n-p *prompt*)	*LAMBDA	Prompt user for y/n or yes/no on standard input.

5.1. File I/O

File streams are used for print to and reading from files. Their type name is STREAM. Standard output and standard input are regarded as file streams represented as nil. A new file stream is opened with:

(openstream *filename mode*)

where *mode* can be "r" for reading, "w" for writing, or "a" for appending. Furthermore, the option "b" indicates that the byte stream accepts writing or reading of binary data. For example:

```
> (setf s (openstream "foo.txt" "w"))
#[STREAM 3396656]
> (print '(hello world 1) s)
(HELLO WORLD 1)
> (closestream s)
#[STREAM 3396656]
> (setf s (openstream "foo.txt" "r"))
#[STREAM 3396800]
> (read s)
(HELLO WORLD 1)
> (closestream s)
#[STREAM 3396800]
>
```

The following system functions and variables handles file I/O and file streams:

Function	Type	Description
(delete-file *nm*)	*EXTFN	Delete the file named *nm*. Returns T if successful.
(file-exists-p *nm*)	*EXTFN	True if file named *nm* exists.
(file-length *nm*)	*EXTFN	The number of bytes in the file named *nm*.
(load *nm*)	*EXTFN	Evaluate the forms in the file named *nm*.
(openstream *nm mode*)	EXTFN	Open a file stream against a file named *nm*. *mode* is the Unix *file mode* i.e. "r", "w", or "a". As errors can happen during the processing of a file causing it not to be closed properly, you are advised to use the macro with-open-file instead of openstream when possible.
(redirect-basic-stdout *nm*)	EXTFN	Redirect all standard output to file named *nm*. In case the system is run inside another system, e.g. inside a web server, standard output is often disabled and this function allows logging in a file instead. To run this function when the system is started, use the '-r file' option or make an aLisp image where the after-rollin-forms (Section 3.17) redirects standard output by calling redirect-basic-stdout. An alternative is the function dribble that prints the user interaction with the aLisp REPL to both a file and the standard input/output streams.

(with-open-file (*str nm* [:direction *d*]) *form*...)

	*MACRO	The file stream `str` is first opened for reading, writing, or appending of a file named *nm*, then the forms `form`… are evaluated, and finally the stream is always closed, even if exceptions are raised while evaluating `form`… The file is opened for reading if `d` is `:input` (default). If `d` is `:output` the file is opened for writing. If `d` is `:append` the file is opened for writing at the end of the file.

5.2. Text streams

Text streams (datatype TEXTSTREAM) allow the I/O functions to work against dynamically expanded buffers instead of files. This provides an efficient way to destructively manipulate large strings. Each text stream internally stores its data as a memory area object (Sec 3.9).

The following aLisp functions are available for manipulating text streams:

Function	Type	Description
`(maketextstream [sz binary])`	EXTFN	Create a new text stream with an optional initial buffer size `sz`. Text streams are by default textual, but this can be overridden by providing a non-nil value of `binary`.
`(maketextstream mem [binary])`	EXTFN	Create a new text stream having the provided memory area `mem` as buffer.
`(textstreambuffer tstr [trim])`	LAMBDA	Get the entire internal memory area buffer of text stream `tstr`. If the flag `trim` is non-nil the text stream buffer is trimmed up to the current read/print cursor position.
`(textstreampos tstr)`	EXTFN	Get the position of the read/print cursor in text stream `tstr`.
`(textstreampos tstr pos)`	EXTFN	Move the read/print cursor to position `pos` in text stream `tstr`.
`(textstreamstring tstr)`	EXTFN	Retrieve the text stream buffer of text stream `tstr` as a string. **Notice** that this function requires the text stream to be non-binary.
`(closestream tstr)`	EXTFN	Reset the cursor of text stream `tstr` to position 0, same as `(textstreampos tstr 0)`.

`(with-textstream tstr str form...)`	MACRO	Open a text stream `tstr` for the string `str`, evaluate the forms `form`... with `tstr` open, and then close `tstr`. The value is the evaluation of the last S-expression in `form`....
		For example:
		`(with-textstream s "(a)(b)" (read s)(read s))`
		` => (B)`

5.3. Sockets

aLisp servers can communicate via TCP sockets. Essentially socket streams are abstracted as conventional I/O streams where the usual aLisp I/O functions work. The aLisp functions `print` and `read` are used for sending forms between aLisp systems using sockets.

5.3.1. Point to point communication

With point-to-point communication two aLisp servers can communicate via sockets by establishing direct TCP/IP socket connections.

The first thing to do is to identify the TCP host on which an aLisp system is running by calling: `(gethostname)` or `(get-my-ip)`.

Server side:

The first step on the server (receiving) side is to open a socket listening for accepted incoming connections. Two calls are needed on the server side:

A new socket object must be created which is going to accept on some port registrations of new socket connections from clients. This is done with:

`(open-socket nil` *`portno`*`)`

For example:

```
> (open-socket nil 1235)
#[socket NIL 1235 1936]
```

`open-socket` returns a new socket object that will listen on TCP port *`portno`*. If *`portno`* is 0 it means that the OS assigns a free port for incoming messages. When the port number of socket S has been assigned can be obtained with the function :

`(socket-portno` *`s`*`)`

Then the server must then wait some client to request connections by calling `accept-socket`:

`(accept-socket` *`s`* `[`*`timeout`*`])`

The function `accept-socket` waits for an `open-socket` call to the server from some client to establish a new connection. If *`timeout`* is omitted the waiting is forever (it can be interrupted with CTRL-C), otherwise it specifies a time-out in seconds. If an incoming connection request is received, `accept-socket` returns a new socket stream to use for communication with the client that issued the `open-socket` request. `accept-socket` returns `nil` if no `open-socket` request was received within the time-out period.

Client side:

On the client side a call to

`(open-socket` *`hostname portno`*`)`

opens a socket stream to the server listening on port number *`portno`* on host *`hostname`*. The parameter *`hostname`* can be a logical host name or an IP address, but must not be `nil` (which would indicate server socket). The result of `open-socket` is a SOCKET object, which is a byte stream. Thus, once `open-socket` is called the regular Lisp I/O functions can be used for communication.

Notice that since socket stream are buffered data is not sent on a socket stream before calling the function:

```
(flush s)
```

To check whether there is something to read on a socket use:

```
(poll-socket s timeout)
```

The function `poll-socket` returns true if something has arrived on socket stream *`s`* within *`timeout`* seconds, and `nil` otherwise. Polling can be interrupted with CTRL-C.

When a client has finished using a socket *`s`* it can be closed and deallocated with:

```
(close-socket s)
```

The garbage collector automatically calls `close-socket` when a socket object is deallocated.

You can associate an object `val` as property `prop` of socket `s` by calling :

```
(socket-put s prop val)
```

The property `prop` of socket `s` is accessed by:

```
(socket-get s prop)
```

Notice that pending data in the socket stream may be lost when `close-socket` is called.

5.3.2. Socket migration

A TCP socket opened by a process on a machine (computer, VM, container, …) may be transferred (migrated) to another process on the same machine. Both processes must be running before migration, as the PID of the receiving process must be known by the sending process. Furthermore, a TCP socket between the sending and receiving process is required for the transfer.

In the following example, a TCP socket on the sending process is migrated to the receiving process. The PID of the receiving process is . The TCP connection between the sending process and the receiving process is called

To migrate a socket from a sending process to a receiving process (with PID), the sending process first prepares a socket for migration:

```
(setf mig (make-migration-socket s receiving-pid))
```

The socket then contains This adds an attribute to the property list of the socket. The value of that property contains OS dependent information necessary to migrate the socket from the sending process to the receiving process. (is the operating system (e.g.)). Next, the sending process packages the socket information and sends it to the receiving process over the TCP connection to the receiving process ().

```
(pf (export-socket-minimal mig) socket-to-receiver)
```

The receiving process receives this information over the TCP connection to the sending process (), and establishes a new socket, called :

```
(setf new-socket (import-socket-minimal (read socket-to-sender)))
```

Now the receiving process can use as if was opened by the receiving process. The sending process should not use its socket (or) any more after migration. Note that no buffer content of socket is transferred during socket migration. Any content in the read buffer of in the sending process will *not* be present in the read buffer of in the receiving process.

Socket migration is currently available on Windows, Mac OS X, and Linux. Note that Linux and Mac OS X utilize unix domain sockets for transferring socket information. (Unix can only use a domain socket for migration of TCP sockets between processes.) This unix domain socket is opened when calling . The life time of this domain socket is one minute. Thus, the receiving process must call within one minute after the sending process called . Furthermore, only one socket can be migrated to a receiving process at any given time.

5.3.3. Remote evaluation

There is a higher level *remote evaluation* mechanism in sa.engine where the system can be set up as a server evaluating incoming Lisp forms from other sa.engine peers. With remote evaluation Lisp forms are sent from one sa.engine peer to another for evaluation there after which the result is shipped back to the caller. The remote evaluation requires the receiving peer to listen for incoming forms to be evaluated.

Server side:

On the server side the following makes an sa.engine server (SAS) named `SRV` behave as a remote evaluation server, accepting incoming forms to evaluate remotely.

First let’s start an sa.engine *name server* named `SRV` in some shell. A name server is a SAS that keeps track of what sa.engine peers listen to what ports for remote evaluation and can also be used as a regular SAS. To start a name server named `SRV` (always upper case), execute the shell command:

```
sa.engine -n srv
```

When the name server `SRV` is up and running it starts listening on incoming remote evaluations by default on port number 35021. You can specify another listening port number *p* by suffixing the SAS name with :*p*, e.g. the following command makes name server `S` listen on port 1234:

```
sa.engine -n s:1234
```

Client side:

Start a stand-alone aLisp system in another shell with:

```
sa.core
```

Now we can open a socket *connection* `_c_` to the SAS named *s* from the client by calling the function (`open-socket-to` *s*), for example:

```
(defglobal _c_ nil)
(setf _c_ (open-socket-to 'srv))
```

To ship as S-expression *form* for evaluation from the client to on an sa.engine server listening on connection *c*, simply call:

```
(socket-eval form c)
```

The function `socket-eval` ships a Lisp form to the connected server for evaluation. For example, the following form prints the string “Hey” on the standard output of the name server `SRV` and ships back the result “Hey Joe”:

```
(socket-eval '(concat (print "Hey") " Joe") _c_)
```

Errors occurring on server are shipped back to client, for example try this:

```
(socket-eval '(/ 3 0) _c_)
```

The function `socket-eval` is synchronous and blocks until the result is received. For non-blocking messages use instead:

```
(socket-send form c)
```

The difference to `socket-eval` is that *form* is evaluated on the server on its own; the client does not wait for the result and is thus non-blocking. Errors are **not** sent back. The function `socket-send` is faster than `socket-eval` since it does not wait for the result to return. If you want to synchronize after many non-blocking messages sent with `socket-send`, end with a `socket-eval`. For example, the following form will return the number 10000:

```
(progn (socket-send '(defglobal _cnt_ 0) _c_)
       (dotimes (i 10000)(socket-send '(incf _cnt_) _c_))
       (socket-eval '_cnt_ _c_))
```

6. Error handling

When the system detects an error it will call the Lisp function:

```
(faulteval errno msg o form frame)
```

where

ERRNO	is an error number (-1 for not numbered errors)
MSG	is an error message
O	is the failing Lisp object
FORM	is the last Lisp form evaluated when the error was detected.
FRAME	is the variable stack frame where the error occurred.

The aLisp default behaviour of `faulteval` first prints the error message and then calls the function `(reset)` to signal an error to the system, an *error signal*. To *reset Lisp* means to jump to a pre-specified *reset point* of the system. By default this reset point is the top level read-eval-print loop. It can also be an unwind protection to be explained next.

6.1. Trapping exceptions

The special form `unwind-protect` is the basic system primitive for trapping all kinds of exceptions.

```
(unwind-protect form [cleanup])
```

The *form* is evaluated until it is terminated, whether naturally or by means of a regular exit or an error signal. The *cleanup* form is then evaluated before control is handed back to the system. Note that the *cleanup* form is **not** protected by that `unwind-protect` so errors produced during evaluation of *cleanup* will be thrown out from the `unwind-protect` call. In such cases errors in *cleanup* can be caught in some other `unwind-protect` above the present one.

The special form `unwind-protect` traps any local or non-local exit, including error signals and `throw` (Sec 3.13). For example, a call to `throw` may cause a catcher to be exited leaving a file open. This is clearly undesirable, so a mechanism is needed to close the file and do any other essential cleaning up on termination of a construct, no matter how or when the termination is caused, which is the purpose of `unwind-protect`.

All errors raised during the evaluation of a form can be caught by using the macro `(catch-error` *form repair*`)`. It evaluates *form* and, if successful, returns the result of the evaluation. In case an error exception happened while evaluating *form* an ***error condition*** is returned from `catch-error`. Such an error condition looks like:

```
(:errcond (errno "errmsg" x))
```

For example:

```
> (catch-error a)
(:ERRCOND (1 "Unbound variable" A))
```

The optional *cleanup* form is evaluated if an error occurred during the evaluation of *form*. In the *cleanup* form the variable `_error-condition_` holds the error condition.

The function `(error?` *ec*`)` tests if *ec* is an error condition. It can be used for testing if `catch-error` returned an error condition. For an error condition *ec*, the function `(errcond-arg` *ec*`)` returns the object causing the error, `(errcond-number` *ec*`)` returns the error number, and `(errcond-msg` *ec*`)` returns the error message.

6.2. Raising errors

The function `(error` *msg x*`)` prints error message *msg* and raises an error for *x*.

To cause an error exception without any error message call `(reset)`.

As any other error these functions will go through the regular error management mechanisms, so user errors can be caught with `unwind-protect` or `catch-error`.

6.3. User interrupts

After an interrupt is generated by CTRL-C the system calls the Lisp function

```
(catchinterrupt)
```

By default `catchinterrupt` resets aLisp. In debug mode (Sec. 7) a break loop is entered when CTRL-C is typed.

For disable (i.e. postpone) CTRL-C during evaluation of a form, use:

(douninterrupted *form*)

6.4. Error management functions

Below follows short descriptions of system functions and variables for error management.

Function	Type	Description
(catch-error *form* [*cleanup*])	MACRO	Trap and repair errors.
(catchdemon *loc val*)	LAMBDA	See `setdemon`.
(catchinterrupt)	LAMBDA	This system function is called whenever the user hits CTRL-C. Different actions will be taken depending on the state of the system.
(douniterrupted *form*)	MACRO	Delays interrupts happening during the evaluation of *form* until `douniterrupted` is exited.
(errcond-arg *ec*)	LAMBDA	Get the argument of error condition *ec*.
(errcond-msg *ec*)	LAMBDA	Get the error message of error condition *ec*.
(errcond-number *ec*)	LAMBDA	Get the error number of error condition *ec*.
(error *msg x*)	EXTFN	Print message *msg* followed by ': ' and object *x* and then generate an error.
(errormessage *no*)	EXTFN	The error message for error number *no*.
(errornumber *msg*)	EXTFN	The error number for error message *msg*. The number is -1 if the message is not stored in an internal table of error messages.
(error-at *msg x fn*)	LAMBDA	Raise error in the context where the function `fn` was called.
(error? *x*)	LAMBDA	True if *x* is an error condition.
(faulteval *errno errmsg x form env*)	LAMBDA	The function `faulteval` is called by the system whenever it detects an error. If the system runs in debug mode (Sec. 7) `faulteval` then enters a break loop (Sec. 7.1). If the system is not in debug mode the function prints the error message and then calls `(reset)`.
(reset)	EXTFN	Signals an unspecified exception. The control is returned to the latest reset point executing all cleanup forms on the way. The reset point is either the top level aLisp REPLor some error trap using, e.g., `catch-error` or `unwind-protect`.
(unwind-protect *form cleanup*)	*SPECIAL	The general system primitive to trap exceptions.

7. Lisp Debugging

This section documents the debugging and profiling facilities of aLisp.

To enable run time debugging of aLisp programs the system should be put in *debug mode*. This is automatically done when entering the aLisp REPL. To enable Lisp debugging also in the OSQL REPL call `(debugging t)`. To disable debugging in the aLisp REPL call `(debugging nil)`. In debug mode the system checks assertions at run time and analyses Lisp function definitions for semantic errors, and thus runs slightly slower. Also, in debug mode the system will enter a *break loop* when an error occurs instead of resetting Lisp, as described next.

The interactive break loop for debugging is difficult or even impossible if you are using the system in a batch environment or an environment where an interactive break loop cannot be entered (e.g. under PHP). For debugging in batch environments set the global variable `_batch_` to true: `(setf _batch_ t)`

When `_batch_` is set and the system is in debug mode errors are trapped and cause a backtrace to be printed printed after which the error is thrown *without* entering the break loop.

7.1. The break loop

The break loop is a Lisp READ-EVAL-PRINT loop where some special debugging commands are available. This happens either when i) the user has explicitly specified a break point for debugging specific *broken functions*, ii) explicit break points are introduced in the code by calling `help`, or ii) when an error happens in debug mode. For example:

```
> (defun foo (x) (fie x))
FOO
> (defun fie (y) x)
Undeclared free variable X in FIE     ← Warning.
FIE
> (foo 1)
ERROR! Unbound variable: X          ← Run time error.
When evaluating: X
(FAULTEVAL BROKEN)                    ← System error break point.
In FIE brk>:bt                        ← Make backtrace.
FIE
FOO
(FAULTEVAL BROKEN)
In FIE brk>:btv                       ← Make more detailed backtrace.
10:_ENV_ <-> 3
9:_ERRFORM_ <-> X
8:_ERROBJ_ <-> X
7:_ERRMSG_ <-> "Unbound variable"
6:_ERRNO_ <-> 1
5:--- (LAMBDA (_ERRNO_ _ERRMSG_ ...) "This function is called by system
whenever error detected" ...) --- @ 3
4:Y <-> 1
3:--- FIE --- @ 0
2:X <-> 1
1:--- FOO --- @ 0
0:--- *BOTTOM* --- @ 0
(FAULTEVAL BROKEN)
In FIE brk>y                          ← Investigate variable y in FIE scope
1
(FAULTEVAL BROKEN)
In FIE brk>:r                         ← Reset Lisp
14.343 s
>
```

In the break loop the following *break commands* are available:

:help Print summary of available debugging commands, i.e. this list.

?= Print variables bound by current frame

:lvars Names of local variables bound at current frame.

:fp Print file position of function at current frame.

:ub Unbreak the function at current frame.

:bt Print a backtrace of functions at current frame. The depth of the backtraces is controlled by the special variable `*backtrace-depth*` that tells how many function frames should be printed. Its default is 500.

:btv Print a detailed backtrace of the frames below the current frame.

:btv* Print a long backtrace including all stack contents.

eval Evaluate current frame.

!value Lisp variable bound to value of evaluating current frame with :eval. Its value is the symbol `!unevaluated` if :eval has not yet been called.

(return x) Return value x from the broken frame, i.e. the frame where the break loop was entered.

:c Continue evaluation from broken frame where the break loop was entered. The value of variable `!value` is used as return value from the break loop if :eval has been called beforehand.

:r Reset to aLisp REPL.

:a Change current frame to the previous broken frame or reset if there is no previous broken frame.

(:f FN) Set current frame to first frame down the stack calling FN.

:nx Set new current frame one step up the stack.

:pr Set new current frame one step down the stack.

:su Evaluate body and print report on how much storage was used during the evaluation.

(:arg N) Function that returns N:th argument in current frame.

(:b VAR) Enter new break loop when VAR becomes bound.

The variables bound in the current frame are inspectable in the break loop, because variables in a break loop are evaluated in the lexical environment of the current frame.

It is possible to explicitly insert a break loop around any Lisp form in a program by using the macro:

```
(help [tag])
```

When `help` is called, the break loop is entered to allow the user to investigate the environment with the usual break commands. The local variables in the environment where `help` was called are also available. The *`tag`* is printed to identify the `help` call. Used for debugging complex Lisp functions.

7.2. Breaking functions

Explicit break points can be put on the entry to and exit from Lisp functions *fn*... by the Lisp macro

```
(break fn...)
```

For example:

```
> (break foo fie)         ← Put break point on FOO and FIE
(FOO FIE)
lisp 1> (foo 1)
(FOO BROKEN)              ← In break point of FOO
In FOO brk>?=             ← Print parameters of FOO and their values
( X=1 )
(FOO BROKEN)
In FOO brk>:eval          ← Evaluate the body of FOO
(FIE BROKEN)              ← The broken function is FIE
In FIE brk>?=             ← The focused function is also FIE
( Y=1 )
(FIE BROKEN)
In FIE brk>y              ← Evaluate variable Y in scope of FIE
1
In FIE brk>(:f foo)       ← Move down the stack to FOO
2:X <-> 1
1:--- FOO --- @ 0
(FIE BROKEN)
In FOO brk>x              ← The focused function is FOO
1
(FIE BROKEN)
In FOO brk>:org           ← Move back to broken function
63:Y <-> 1
62:--- FIE --- @ 0
(FIE BROKEN)
In FIE brk>:args          ← Look at arguments of broken function
(Y)
(FIE BROKEN)
In FIE brk>:r             ← Reset Lisp
>
```

When such a *broken* function is called the system will also enter a break loop where the above break commands are available.

Breaks on macros mean testing how they are expanded. If you break an EXTFN the argument list is in the variable `!ARGS`.

The break points on functions can be removed with:

```
(unbreak fn...)
```

For example:

```
(unbreak foo fie)
```

To remove all current function breaks do:

```
(unbreak)
```

7.2.1. Conditional break points

The aLisp debugger also permits *conditional break points* where the break loop is entered only when certain conditions are fulfilled. A conditional break point on a function `fn` is specified by pairing `fn` with a *precondition function*, `precode`:

```
(break … (fn precond) …)
```

When `fn` is called `precond` is first called with the same parameters. If `precond` returns `nil` no break loop is entered, otherwise it is.

For example:

```
(break (+ floatp))
(break (createtype (lambda (tp)(eq tp ’person))) )
```

Then no break loop is entered by the call:

```
(+ 1 2 3)
```

By contrast, this call enters a break loop:

```
(+ 1.1 2 3)
```

7.3. Tracing functions

It is possible to trace Lisp functions `fn`... with the macro:

```
(trace fn…)
```

When such a *traced* function is called the system will print its arguments on entry and its result on exit. The tracing is indented to clarify nested calls.

Macros and special forms can also be traced or broken to inspect that they expand correctly.

Remove function traces with:

```
(untrace fn…)
```

To remove all currently active traces do:

```
(untrace)
```

Analogous to conditional break points, *conditional tracing* is supported by specifying a function name `fn` in `trace` with a pair of functions `(fn precond)`, for example:

```
> (trace (+ floatp))
(+)
> (+ 1 2)
3
> (+ 1.1 2)
--> + ( !ARGS=(1.1 2) )
```

```
<-- + = 3.1
3.1
> (+ 1 2.1)
3.1
>
```

7.4. Profiling

There are two ways to profile aLisp programs for identifying performance problems:

- The *statistical profiler* is the easiest way to find performance bottlenecks. It works by collecting statistics on what aLisp functions were executing at periodically sampled time points. It produces a ranking of the most commonly called aLisp functions. The statistical profiler has the advantage not to disturb the execution significantly, at the expense of not being completely exact.
- The *wrapping profiler* is useful when one wants to measure how much wall time is spent inside a particular function. By the function profiler the user can dynamically wrap Lisp functions with code to collect statistics on how much time is spent inside particular functions. The wrapping profiler is useful to exactly measure how much time is spent in specific functions. Notice that the wrapping makes the instrumented function run slower so the wrapping profiler can slow down the system significantly if the wrapped function does not use much time per call.

7.4.1. The Statistical Profiler

The statistical profiler is turned on by:

```
(start-profile)
```

After this the system will start a background timer process that regularly (default every millisecond) updates statistics on what function was executing at that time. After starting the statistical profiler you simply run the program you wish to profile.

When the statistics is collected, the percentage most called aLisp functions is printed with:

```
(profile)
```

You may collect more statistics to get better statistics by re-running the program and then call `profile` again.

Statistical profiling is turned off with:

```
(stop-profile)
```

The function `stop-profile` also clears the table of call statistics.

For example;

```
> (start-profile)
STAT-FUNCTION
> (defun fib (x)
     (if (< x 2) 1 (+ (fib (- x 1))(fib (- x 2)))))
FIB
> (fib 30)
1346269
```

```
   > (profile)
(120 (FIB . 99.1) (DEFUN . 0.8))
> (stop-profile)
```

The function `profile` returns a list where the first element is the number of samples and the rest lists the percentage spent in each function. Profile takes as argument an optional cut-off percentage. For example:

```
> (profile 1)
(120 (FIB . 99.1))
>
```

The sampling frequency is controlled with the global variable `_profiler-frequency_`. It is by default set to 0.001 meaning that up to 1000 samples are made per second. In practice the actual number of samples can be smaller.

The sampling is also influenced by the value of the global variable `_exclude-profile_` containing a list of functions excluded from sampling. The sampler registers the first call on the execution stack *not* in this list. For advanced profiling it is sometimes useful to exclude commonly called uninteresting functions by adding more functions to `_exclude-profile_`.

7.4.2. The Wrapping Profiler

To collect statistics on how much real time is spent in specific aLisp functions *fn*... and how many times they are called use the wrapping profiler:

```
(profile-functions fn...)
```

For example:

```
(profile-functions subset getfunction)
```

The calling statistics for the profiled functions are printed (optionally into a file) with:

```
(print-function-profiles [file])
```

The calling statistics are cleared with:

```
(clear-function-profiles)
```

Function profiling can be removed from specific functions *fn*... with:

```
(unprofile-functions fn...)
```

To remove all function profiles do:

```
(unprofile-functions)
```

Analogously to conditional break points, *conditional function profiling* is specified by pairs `(fn precond)` as arguments to `profile-functions`, e.g.

```
(profile-functions (createtype (lambda(x)(eq x 'person))) )
```

The function profiler does not double measure recursive functions calls. When a functions call causes error throws it is not measured.

7.5. System functions for debugging

We conclude this chapter with a list of all aLisp system functions useful for debugging:

Function	Type	Description
`(backtrace` *`depth`* `[`*`frame filtered`*`])`	EXTFN	Print a backtrace of the contents of the current variable binding stack. *`depth`* indicates how many function frames are printed. If *`filtered`* is true the arguments of EXTFNs are excluded from the backtrace. The parameter *`frame`* specifies at what stack frame number the backtrace shall start. Default is top of stack.
`_batch_`	GLOBAL	If this variable is true no break loop is entered after errors are detected. Instead the system make a backtrace (command :btv Sec. 7.1) and resets the system. Useful when running in batch or in servers.
`(break` *`fn`*...`)`	MACRO	Put break points on entries to Lisp functions *`fn`*... so that an interactive break loop is entered when any of the broken functions are called (Sec. 7.1).
`(clear-function-profiles)`	LAMBDA	Clear the statistics for wrapping profiling (Sec. 7.4.2).
`(debugging` *`flag`*`)`	EXTFN	If *`flag`* is true the system will start running in *debug mode,* where warning messages are printed and the system checks assertions. Turn off debug mode by calling with *`flag`* `nil`. Notice that the system by default is in debug mode when the aLisp REPL is entered.
`(dumpstack)`	EXTFN	Print all the contents of the variable binding stack.
`(help [`*`tag`*`])`	MACRO	To insert explicit break points in Lisp code. The *`tag`* indentifies the HELP call. For example: `(help foo)`
`(image-expansion` *`rate`* `[`*`move`*`])`	EXTFN	When the database image if full it is dynamically expanded by the system. This function controls the expansion. The parameter *`rate`* specifies how much the image is to be expanded (default 1.25). If *`move`* is true the image will always be copied to a different place in memory after image expansion. If *`move`* is false it may or may not be copied. To test system problems related to the moving of the image the following call will make the image move a lot when data is loaded: `(image-expansion 1.0001 t)`
`(loc` *`x`*`)`	EXTFN	Return the location (handle) of Lisp object *`x`* as an integer. The inverse is `(vag` *`x`*`)`.
`(print-function-profiles [`*`file`*`])`	LAMBDA	Print statistics on time spent in profiled functions (Sec 7.4.2).
`(printframe` *`frameno`*`)`	EXTFN	Print the variable stack frame numbered *`frameno`*.
`(printstat)`	EXTFN	Print storage usage since the last time `printstat` was called. Good for tracing storage leaks and usage.
`(profile)`	LAMBDA	Print statistics of time spent in aLisp functions after a statistical profiling execution (Sec 7.4.1).

`(profile-functions` *fn*...`)`	MACRO	Wrap the aLisp functions *fn*... with code to collect statistics on how much real time was spend inside them (Sec. 7.4.2).
`(refcnt` *x*`)`	EXTFN	Return the reference count of object *x*. Useful for debugging of storage leaks.
`(setdemon` *loc val*`)`	EXTFN	Set up a system trap so that when the word at image memory location *loc* becomes equal to integer *val* the system will call the Lisp function `(catchdemon` *loc val*`)`, which by default is defined to enter a break loop. The trap is immediately turned off when the condition is detected, or when a regular interrupt occurs. Useful for detecting memory corruption in C-code interfaced to the system.
`(start-profile)`	LAMBDA	Start statistical profiling of a Lisp program. (Sec. 7.4.1)
`(stop-profile)`	LAMBDA	Stop profiling the Lisp program. (Sec. 7.4.1)
`(storagestat` *flag*`)`	LAMBDA	If *flag* is true the aLisp REPL prints how much data was allocated and deallocated for every evaluated Lisp form in the aLisp REPL. Very useful for finding storage leaks.
`(storage-used` *form* `[`*tag*`])`	SPECIAL	Evaluate *form* and print a report on how many data objects of different types were allocated by the evaluation. The optional parameter *tag* TAG identifies the report. Good for finding storage leaks.
`(time` *form*`)`	*MACRO	Print the real time spent evaluating *form*. Returns value of *form*. Often used in combination with `rptq` (Sec. 3.13).
`(trace` *fn*...`)`	MACRO	Put a trace on the functions *fn*... (Sec 7.3). The arguments and the values will then be printed when any of these functions are called. Remove the tracing with `untrace`.
`(traceall` *flag*`)`	EXTFN	Trace *all* function calls if *flag* is true. The massive tracing is turned off with `(traceall nil)`.
`(trapdealloc` *x*`)`	EXTFN	Set up a demon so that the break loop is entered when the object *x* is deallocated. Good for finding out where objects are deallocated by the garbage collector.
`(unbreak` *fn*...`)`	MACRO	Remove the break points around the specified functions (Sec. 7.2).
`(unprofile-functions` *fn*...`)`	MACRO	Remove function profiles from the specified functions (Sec. 7.4.2).
`(vag` *x*`)`	EXTFN	Return the aLisp object at image location *x*. The inverse is `(loc` *x*`)`.
`(virginfn` *fn*`)`	LAMBDA	Get the original definition of the function *fn*, even if *fn* is traced or broken.

8. Code search and analysis

As Lisp code is also data it is stored in the internal database image. A number of system functions are available for searching and analyzing Lisp code in the image. This can be used for finding functions, printing function documentation, cross-referencing functions, analysing correctness of functions, etc.

8.1. Emacs subsystem

aLisp can run as a subprocess to Emacs . The most convenient way to develop aLisp code is to run from a shell within Emacs. Emacs should be configured using the file `init.el` that provides extensions to Emacs for finding Lisp code and for evaluating Lisp by aLisp. Place `init.el` in the initialization folder of Emacs (on Linux the file /home/.emacs).

When Emacs is started give the command:

```
M-x-shell
```

This will start a new Windows (or Unix) shell inside Emacs. You can there give the usual Windows (Unix) commands.

First check that the Emacs init file was loaded correctly by typing F1. If it was loaded correctly there should be a message:

Error: ‘’ is not a file

When Emacs initializes OK, run sa.engine in the Emacs shell by issuing the command:

```
   sa.engine
```

If you are developing Lisp code, enter to the aLisp REPL the command:

```
lisp;
```

8.2. Finding source code

The system contains many Lisp functions and it may be difficult to find their source code. To alleviate this, there are Lisp code search functions for locating the source codes of Lisp functions and macros loaded in the database image having certain properties. Most code search functions print their results as *file positions* consisting of file names followed by the line number of the source for the searched function. Only source code of LAMBDA functions and macros has file positions.

If Emacs is configured properly, the Emacs key F1 (defined in init.el) can be used for jumping to the source code of a file location at the mouse pointer. For example, the function (FP FN) prints the file position of a function:

```
> (fp 'printl)
PRINTL C:/AmosNT/lsp/orginit.lsp 530
T
```

If you place the pointer over the file name and press F1 you should be placed in a separate Emacs window at the file position where the function `printl` is defined. If F1 is undefined you have not installed `init.el` properly.

If you have edited a function with Emacs it can be redefined in aLisp by cut-and-paste. The key F2 will send the form starting at the pointer position in the file source window to the shell window for evaluation.

If you don’t have the source code you can still look at the definition using `pp`:

```
> (pp printl)
```

```
(DEFUN PRINTL (&REST L)
  "Print list of arguments on standard output"
  (PRINT L))
(PRINTL)
```

The macro pp prints the definitions of functions from their internal representation in the database image. The appearance in the source file is normally more informative, e.g. including comment lines and with no macros expanded.

Often you vaguely know the name of a function you are looking for. To search for a function where you only know a part of its name use the CommonLisp function `(apropos fn)`. For example:

```
> (apropos 'ddd)
CADDDR C:/AmosNT/lsp/orginit.lsp 47
   ""
CDDDDR C:/AmosNT/lsp/orginit.lsp 45
   ""
CDDDR
   EXTFN
```

Here we see that the function `cdddr` is an external function with no source code. We can inspect its definition and see that it is an EXTFN with:

```
> (pp cdddr)
(DEFC 'CDDDR #[EXTFN1 CDDDR])
(CDDDR)
```

The function `apropos` prints the documentation of LAMBDA functions and macros. For example:

```
> (apropos 'printl)
PRINTL C:/AmosNT/lsp/orginit.lsp 530
   "Print list of arguments on standard output"
NIL
```

The documentation of a function should be given as a string directly after the formal parameter list, as is done for `printl`.

To find where a structure is defined you can search for its construction. For example:

```
> (apropos 'make-selectbody)
MAKE-SELECTBODY C:/AmosNT/lsp/function.lsp 46
   ""
```

Summary of Lisp code documentation and search functions:

Function	Type	Description
`(doc` *fn*`)`	LAMBDA	Return the documentation string for function *fn*.
`(fp` *fn*`)`	LAMBDA	Print the file position of definition of function *fn*. The file position of the currently focused function in the break loop is printed with the command: `:fp`
`(grep` *string*`)`	LAMBDA	Print the lines matching the string in all source files currently

loaded in the database image.

`(calling fn [levels file])` LAMBDA Print the file positions for the functions calling the function *`fn`*. The optional parameter *`levels`* specifies how many levels of functions that call *`fn`* indirectly are printed (default 1). The optional *`file`* prints the report to a file.

`(calls fn [levels file])`

LAMBDA Print the file positions for the functions called from function *`fn`*. The optional *`levels`* specifies how many levels of functions that are called indirectly by *`fn`* are printed (default 1). Optional *`file`* prints report to a file.

`(using var)` LAMBDA Print the file positions for the functions whose definitions use the variable *`var`*.

`(matching pat)` LAMBDA Print the file positions of functions whose definitions match somewhere the code pattern *`pat`*. A pattern is an S-expression where the symbol * matches everything,. For example:
`(matching '(map* '* . *))`
matches functions containing, e.g., the form
`(mapcar 'print l).`

8.3. Code verification

aLisp has a subsystem for verifying Lisp code. The code verification goes through function definitions to search for code patterns that are seem erroneous. It also looks for calls to undefined functions, undefined variables, etc. The code verifier is automatically enabled incrementally when in debug mode. However, full code verification requires that all functions in the image are analyzed, e.g. to verify that all called functions are also defined. To verify fully all functions in the image, call:

```
(verify-all)
```

It goes through all code and prints a report when something incorrect is found. For example:

```
> (verify-all)
NIL                                    ← All Lisp functions in image OK
3.75 s
> (defun foo (x)(fie x))
FOO
> (verify-all)
Call to undefined function FIE in FOO. ← FOO was not OK
NIL
3.75 s
```

References

1 *Common Lisp HyperSpec* http://www.lispworks.com/documentation/HyperSpec/Front/.

2 Guy L.Steele Jr.: *Common LISP, the language*, Digital Press, http://www.cs.cmu.edu/Groups/AI/html/cltl/cltl2.html

3 *Read–eval–print loop*, https://en.wikipedia.org/wiki/Read–eval–print_loop.

4 *sa.engine Whitepaper*, https://docs.streamanalyze.com/whitepaper.html

Index