

ALisp v2 User's Guide

**Tore Risch
Uppsala Database Laboratory
Department of Information Technology
Uppsala University
Sweden**

Tore.Risch@it.uu.se

2006-02-22

ALisp is an interpreter for a subset of CommonLisp built on top of the storage manager of the Amos II database system. The storage manager is 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 ANSI C 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 C or Lisp based data structures can be introduced to the system. ALisp runs under Windows and Linux. It is delivered as an executable and a C 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.....	4
2.	Starting ALisp.....	4
3.	Basic Primitives	5
3.1.	Data types.....	6
3.2.	Symbols.....	6
3.2.1.	Defining functions	7
3.2.2.	Binding variables	8
3.2.3.	Symbol manipulation	10
3.3.	Lists.....	11
3.3.1.	Destructive list functions	13
3.4.	Strings	14
3.5.	Numbers.....	15
3.6.	Logical Functions.....	16
3.7.	Arrays.....	17
3.8.	Hash Tables.....	19
3.9.	Main memory B-trees	20
3.10.	Functional arguments and dynamic forms.....	21
3.10.1.	Closures.....	21
3.10.2.	Applying functions with variable arity	22
3.10.3.	Dynamic evaluation	23
3.10.4.	System functions for run-time evaluation.....	23
3.11.	Map functions	23
3.12.	Control Structures	25
3.12.1.	Compound expressions	25
3.12.2.	Conditional expressions	25
3.12.3.	Iterative statements	27
3.12.4.	Non-local returns	27
3.13.	Macros.....	28
3.14.	Defining structures.....	30
3.15.	Miscellaneous functions.....	31
3.16.	Hooks	32
4.	Time Functions	33
4.1.	System clock.....	33
4.2.	Absolute Time Values.....	34
4.3.	Relative time values.....	34
4.4.	Relative Date Values.....	35
5.	Input and Output	35
5.1.	File I/O.....	36
5.2.	Text streams	37
5.3.	Sockets	38
5.3.1.	Point to point communication.....	38
5.3.2.	Remote evaluation	39
6.	Error handling.....	40
6.1.	Trapping exceptions.....	41

6.2.	Raising errors	41
6.3.	User interrupts.....	42
6.4.	Error management functions.....	42
7.	Lisp Debugging.....	43
7.1.	The break loop	43
7.2.	Breaking functions.....	45
7.2.1.	Conditional break points.....	46
7.3.	Tracing functions	46
7.4.	Profiling	47
7.4.1.	The Statistical Profiler	47
7.4.2.	The Wrapping Profiler	48
7.5.	System functions for debugging	49
8.	Code search and analysis	51
8.1.	Emacs subsystem	51
8.2.	Finding source code	51
8.3.	Code verification.....	53
9.	The Storage Manager.....	54
9.1.	Handles	54
9.2.	Physical Objects.....	54
9.3.	Logical Data Objects.....	55
9.4.	Dereferencing.....	55
9.5.	Assigning handles to locations.....	56
9.6.	Allocating physical objects.....	57
9.7.	Storage types.....	59
9.8.	Streams.....	60
9.8.1.	Marshalling objects.....	61
10.	Interfacing Lisp with C	62
10.1.	Calling C from Lisp	62
10.1.1.	Defining external Lisp functions in C.....	63
10.1.2.	Variable arity external Lisp functions.....	65
10.1.3.	Defining special forms.....	66
10.2.	Error management in C.....	67
10.2.1.	Unwind Protection	67
10.2.2.	Raising errors.....	68
10.3.	Calling Lisp from C	69
10.3.1.	Direct C calls.....	70
10.4.	C functions for debugging	71
10.5.	Interrupt handling.....	72

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 therefore tightly interfaced with ANSI C and can share data structures and code with C. ALisp supports a subset of CommonLisp and conforms to the CommonLisp standard when possible. However, it is not a full CommonLisp implementation, but rather such constructs are not implemented that are felt not being critical and difficult to implement efficiently. These restrictions make ALisp relatively small and light-weight, which is important when embedding it in other systems.

ALisp was designed to be embedded in the Amos II object-relational database kernel [2]. However, ALisp is a general system and can be used for many other applications as well. Because it is used in a database kernel it is very important that its storage manager is efficient and *scales* well. 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 and the size of the virtual memory backing file. 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 garbage collector. Lisp programs often generate large amounts of temporary data areas that need to be reclaimed by garbage collection. Furthermore, as ALisp was designed to be used in a DBMS kernel 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. Therefore, unlike many other implementations of Lisp (SmallTalk, Java, etc.) systems, 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 going to be used for managing large database objects as in ALisp's main application.

Section 2 describes the system functions in ALisp. The differences w.r.t. CommonLisp are documented. Section 3 gives an overview of the debugging facilities, while Section 4 describes the error handling mechanisms. Section 5 describes the I/O system and Section 6 overviews the storage manager interfaces.

2. Starting ALisp

ALisp is a subsystem of Amos II [2]. Amos II is started with the OS command:

```
amos2
```

The same directory as where Amos II is started must have a *database image* file with a system Amos II database:

```
amos2.dmp
```

When Amos II is started it is in *AmosQL mode* where it accepts and evaluates AmosQL commands from the console. To enter *Lisp mode* from the AmosQL top loop, give the AmosQL command:

```
lisp;
```

When in Lisp mode the system reads S-expressions, evaluates them, and prints the results from the evaluation, for example:

```

> (setq a '(a b c))
WARNING! Setting undeclared global variable: A
(A B C)
> (reverse a)
(C B A)
> (defun foo (x)(+ 1 a))
Undeclared free variable A in FOO
FOO
>

```

As you can see, ALisp warns the user when it encounters forms that may contain errors. If you make an error ALisp will enter a *break loop* where the error can be investigated. The simplest thing to do is to enter `:r` to reset the error. For example:

```

> your-age
Error 1, Unbound variable: YOUR-AGE
When evaluating: YOUR-AGE
(FAULT EVAL BROKEN)
In *BOTTOM* brk>:r
>

```

See Sec. 7.1 for documentation of all break loop commands.

The recommended way to learn about ALisp is to run a CommonLisp tutorial, e.g. <http://mypage.iu.edu/~colallen/lp/>. Notice that ALisp is a subset of CommonLisp so not all exercises there are applicable, in particular CommonLisp `FORMAT` function is replaced with a simplified `FORMATL` (Sec. 5) and arrays (Sec. 3.7) are one-dimensional.

To go back to AmosQL mode from Lisp mode, enter the keyword:

```
:osql
```

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 with the ALisp function:

```
(rollout filename)
```

which is equivalent to the AmosQL command:

```
save "filename";
```

To later connect Amos II to a previously saved image, issue the OS command:

```
amos2 filename
```

3. Basic Primitives

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

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

3.1. Data types

Every object in ALisp belongs to exactly one data type. There is a system provided *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 ALisp function:

```
(TYPEENAME O)
```

ALisp provides a set of built-in basic data types. However, through its C-interface ALisp can be in addition 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), ii) the ALisp garbage collector is available in C (Sec. 8), iii) ALisp can call functions implemented in C (Sec. 10.1.1), iv) ALisp functions can be called from C (Sec 10.3), and v) C can utilize ALisp's error management (Sec. 6 and 10.2).

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 dataase 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 special 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:

The *print name* is a string representing the symbol. The print name of a symbol can be accessed by the function MKSTRING. For example:

```
> (mkstring 'banana)
"BANANA"
```

1. 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.
2. Each symbol *nm* has an associated *function cell* where the ALisp function definition for the function named *nm* is stored. The function cell of a Lisp symbol *nm* is retrieved with the CommonLisp function (SYMBOL-FUNCTION *nm*) that 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 system special function DEFUN, e.g.:

```
> (defun rev (x)(cons (cdr x)(car x)))
```

```

REV
> (rev '(1 2))
((2) . 1)

```

- ii) A *macro* (Sec.3.13) is defined by the system special function 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 (Sec. 10.1). A external Lisp function is represented by a special data type named EXTFN and printed as #[EXTFNn fn], where n is the arity of the function and fn is its name. E.g. the definition of CONS is printed as #[EXTFN2 CONS]. The EXTFN data structure contains a pointer to the C definition. For example:

```

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

```

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

```

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

```

- v) A *special form* is a 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]

```

3. 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. Described functions that are similar or equivalent to standard CommonLisp functions are marked with a '*' under *Type*. The *Type* of a function can be EXTFN (defined in C), SPECIAL (special form), LAMBDA (defined in Lisp), or MACRO (Lisp macro) (Sec. 3.13). A system variable can be either SPECIAL (dynamically scoped) or GLOBAL (Sec. 3.2.2).

3.2.1. Defining functions

This section describes the system functions to define and manipulate Lisp functions.

Function	Type	Description
(DEFC FN DEF)	EXTFN	Associate the function definition DEF with the atom FN. Same as SYMBOL-SETFUNCTION.
(DEFUN FN ARGS FORM...)	*SPECIAL	Define a new Lisp function.
(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...
(GETD X)	EXTFN	Get function definition of atom NM. Same as SYMBOL-FUNCTION
(LAMBDA P X)	LAMBDA	Return T if X is a lambda expression.

(MOVD F1 F2)	EXTFN	Make F2 have the same function or macro definition as F1.
(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 special system function SETQ.

In ALisp global variables should be declared before they are used, using the system 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:

```

> (setq x 1)
WARNING! Setting undeclared global variable: X ← because X is undeclared
1
lisp 1> (defglobal _g_) ← declare _G_ as global
NIL
lisp 1> (setq _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_ initialized 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 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 [1]. In CommonLisp dynamically scoped variables are declared using the special system function DEFVAR. 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. Usually 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
> (setq *angle-unit* (/ 3.1415926 180)) ← Let's use degrees instead
0.0174533
> (hl 45 10)
14.1421
> (setq *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.
(DEFGLOBAL VAR &optional VAL)	MACRO	Declare VAR to be a <i>global variable</i> with optional initial value VAL. Unlike dynamically scoped variables global variables are not temporarily reset locally with LET/LET*. They are much faster to look up than dynamically scoped variables (see DEFVAR).
(DEFVAR VAR &optional VAL)	*SPECIAL	Declare VAR to be a <i>special variable</i> with optional initial value VAL. 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. The classical PROG and GO are NOT implemented in ALisp. The most common use of PROG is as a LET

with a RETURN, which is supported by PROG-LET.

(PROG-LET* ((VAR INIT...)...) FORM...)	MACRO	This function is similar to PROG-LET, but binds the local variables sequentially like LET*.
(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. This is similar to declaring VAR being special with DEFVAR as specified by the CommonLisp standard. DEFVAR should normally be used.
(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 unevaluated variable VAR to VAL.
(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 often 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. The Lisp function GET is used for accessing property lists (Sec. 3.2.3).

Function	Type	Description
(ADDPROP S I V FLG)	EXTFN	Add a new value V to the list stored on the property I of the symbol S. If FLG = NIL the new value is added to the end of the old value, 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 indicator I.
(GETPROP S I)	EXTFN	Same as GET.
(KEYWORDP X)	*EXTFN	Return T if X is a keyword (i.e. symbol starting with ':').
(KEYWORD-TO-ATOM X)	EXTFN	Convert a keyword X into a regular symbol without the ':'.
(MKSYMBOL X)	EXTFN	Coerce a string to a symbol. The characters of X will be capitalized.
(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 indicator I to V. Same as (SETF (GET S I) V).

(PUTPROP A I V)	EXTFN	Same as PUT.
(REMPROP S I)	*EXTFN	Remove property stored for the indicator I in the property list of symbol S.
(SYMBOL-PLIST S)	*EXTFN	Get the entire property list of the symbol S.
(SYMBOLP X)	*EXTFN	Return true if X is a symbol.

3.3. Lists

Lists (data type name LIST) represent list structures as binary trees. 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:

```
> (setq 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).
(ANDIFY L)	LAMBDA	Make an AND form of the forms in L.
(APPEND L...)	*MACRO	Make a copy of the concatenated lists L... (APPEND X) copies the top level elements of the list X.
(APPEND2 X Y)	EXTFN	Append two lists X and Y.
(ASSOC X ALI)	*EXTFN	Search association list ALI for a pair (X .Y). Tests with EQUAL.
(ASSQ X ALI)	EXTFN	Similar to ASSOC but tests with EQ.
(ATOM X)	*EXTFN	True if X is not a list or if it is NIL.
(BUILDL L X)	LAMBDA	Build a list of same length as L whose elements are all the same X.
(BUILDN N X)	LAMBDA	Build a list of length N whose elements all are all the same X:
(BUTLAST L)	*EXTFN	Return a copy of list L minus its last element.
(CAAAR X)	*EXTFN	(CAR (CAR (CAR X)))
(CAADR X)	*EXTFN	(CAR (CAR (CDR X)))
(CAAR X)	*EXTFN	(CAR (CAR X))
(CADAR X)	*EXTFN	(CAR (CDR (CAR X)))
(CADDR X)	*EXTFN	(CAR (CDR (CDR X))), same as (THIRD X)
(CADR X)	*EXTFN	(CAR (CDR X)), same as (SECOND X)
(CAR X)	*EXTFN	Return the head of the list X, same as (FIRST X).
(CDAAR X)	*EXTFN	(CDR (CAR (CAR X)))
(CDADR X)	*EXTFN	(CDR (CAR (CDR X)))
(CDAR X)	*EXTFN	(CDR (CAR X))
(CDDAR X)	*EXTFN	(CDR (CDR (CAR X)))
(CDDDDR X)	*LAMBDA	(CDR (CDR (CDR (CDR X))))

(CDDDR X)	*EXTFN (CDR (CDR (CDR X)))
(CDDR X)	*EXTFN (CDR (CDR X))
(CDR X)	*EXTFN Return the tail of the list X, same as (REST X).
(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. To copy the top level only, use (APPEND L).
(EIGHT L)	*LAMBDA Get the eight element from list L.
(FIFTH L)	*LAMBDA Get fifth element in list L.
(FIRST L)	*EXTFN Get first element in list L. Same as (CAR L).
(FIRSTN N L)	LAMBDA Return a new list consisting of the first N elements in the list L.
(FOURTH L)	*LAMBDA Get fourth element in list L.
(GETF L I)	*EXTFN Get value stored under the indicator I in the property list L.
(IN X L)	EXTFN IN returns T if there is some substructure in L that is EQ to X.
(INTERSECTION X Y)	*EXTFN Build 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.
(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 with 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 This function returns true if X is a list cell or NIL.
(LOCATEPOS L POSL)	LAMBDA Select the substructure in L specified by the position chain POSL. For example: (LOCATEPOS '(A (B (C))) '(1 1 0)) => C
(MEMBER X L)	*EXTFN Tests if element X is member of list L, Tests with EQUAL. 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 A B FN)	*LAMBDA Merge the two lists A and B with FN as comparison function. For example: (MERGE '(1 3) '(2 4) (function <)) => (1 2 3 4)
(MKLIST X)	EXTFN X is returned if it is NIL or a list. Otherwise (LIST X) is returned.
(NATOM X)	*EXTFN Return T if X is not an atom and not NIL. Anything not being a list is an atom.
(NINTH L)	*LAMBDA Get ninth element in list L.
(NTH N L)	*EXTFN Get the Nth element of the list L with enumeration starting at 0.
(NTHCDR N L)	*EXTFN Get the Nth tail of the list L with enumeration starting at 0.
(NULL X)	*EXTFN True if X is NIL.

(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	Same as (SETQ L (CDR L)).
(PUSH X VAR)	*MACRO	Add X to the front of the list bound to the variable VAR, same as (SETQ VAR (CONS X VAR)).
(PUTF L I V)	EXTFN	Set the value of the indicator I on the property list L to V.
(RECONS X Y L)	LAMBDA	Similar to (CONS X Y) but if the result object has the same head and tail as L then L is returned. Useful for avoiding to copy substructures.
(REMOVE X L)	*EXTFN	Remove all occurrences of X from the list L. Tests with EQUAL.
(REST L)	*LAMBDA	Same as CDR.
(REVERSE L)	*EXTFN	Return a new list whose elements are the reverse of the top level of L.
(SECOND L)	*EXTFN	Get second element in list L. Same as CADR.
(SET-DIFFERENCE X Y)	*EXTFN	Return a list of the elements in X which are not member of the list Y. Tests with EQ.
(SEVENTH L)	*LAMBDA	Get the seventh element of the list L.
(SIXTH L)	*LAMBDA	Get the sixth element of the list L.
(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	Return 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	Get the tenth element in the list L.
(THIRD L)	*EXTFN	Get the third element of the list L. Same as CADDR.
(UNION X Y)	*EXTFN	Construct a list of the elements occurring in both the lists X and Y. Tests with EQ.
(UNIONL L)	LAMBDA	Construct a list of the elements occurring in all the elements of the list of lists L.
(UNIQUE L)	EXTFN	Remove all duplicate elements in the list L. Tests with EQUAL

3.3.1. Destructive list 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 garbage collector that immediately discards no longer used objects. However, sometimes one needs to actually modify list structures by 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 maintained as lists are updated. Therefore Lisp has a number of destructive list manipulating functions that replace

pointers rather than extensively copying lists. Notice that such destructive functions may cause bugs that are difficult to find. Therefore destructive functions should be avoided if possible. As an example of a destructive list, the function (RPLACA X Y) replaces (CAR X) with Y. For example:

```

> (setq a '(1 2 3 4))
(1 2 3 4)
> (rplaca (caddr a) 8)      ← Replace (CAR (CADDR X)) with 8
(8 4)
> a                          ← The list held in A has changed
(1 2 8 4)
> (rplaca a a)              ← This makes a circular list. Don't do this!
((((((((((((((((((((((((((((((((((((((((((((((((((((((((
← That makes the printer loop! (use CTRL-C)

```

CommonLisp makes it very easy to make destructive list operations using SETF [1]. ALisp does not support such SETF on lists to not encourage destructive list operations.

The following destructive system list functions are supported:

Function	Type	Description
(ATTACH X L)	EXTFN	Similar to (CONS X L) but <i>destructive</i> , 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. This does <i>not</i> work if L is not a list, in which case ATTACH works like CONS.
(DELETE X L)	*EXTFN	Remove <i>destructively</i> the elements in the list L that are EQ to X. Value is the updated L. 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 <i>destructively</i> 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.
(NCONC L...)	*MACRO	<i>Destructively</i> concatenate the lists L... (destructive APPEND) and return the concatenated list.
(NCONC1 L X)	EXTFN	Add X to the end of L <i>destructively</i> , i.e. same as (NCONC L (LIST X))
(NREVERSE L)	*EXTFN	<i>Destructively</i> reverse the list L. The value is the reversed list. L will be destroyed. Very fast reverse.
(RPLACA L X)	*EXTFN	<i>Destructively</i> replace the head of list L with X.
(RPLACD L X)	*EXTFN	<i>Destructively</i> replace the tail of list L with X.
(SMASH X Y)	LAMBDA	Replace <i>destructively</i> the list X with the list Y.

3.4. Strings

Strings (data type name STRING) represent text strings of arbitrary length. Strings containing the characters “ or \ must precede these with the *escape character*, \. Examples of strings:

```

> (setq a "This is a string")
"This is a string"
> (setq 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
(CONCAT STR...)	EXTFN	Coerce the arguments STR... to strings and concatenate them.
(INT-CHAR X)	*EXTFN	If possible, return the character string with the encoding integer X, otherwise return NIL. Unlike CommonLisp there is no special data type for characters in ALisp and instead a string with a single character is used.
(MKSTRING X)	EXTFN	Coerce an atom to a string.
(STRING-DOWNCASE STR)	*EXTFN	Change all ASCII characters in string STR into lower case.
(STRING-UPCASE STR)	*EXTFN	Change all ASCII characters in the string STR to upper case.
(STRING< S1 S2)	*EXTFN	Return true if the string S1 alphabetically precedes S2.
(STRING= S1 S2)	*EXTFN	Return true if the strings S1 and S2 are the same. EQUAL works too.
(STRING-LIKE PAT S)	EXTFN	Returns true if PAT matches string S. 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.
(STRINGP X)	*EXTFN	Return true if X is a string.

3.5. Numbers

Numbers represent numeric values. Numeric values can either be 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)	*LAMBDA	Subtract Y from X.

(1+ X)	*MACRO	Add one to X which can be both integer and real.
(1++ X)	MACRO	Increment the variable X.
(1- X)	*MACRO	Subtract one from X which can be both integer and real.
(1-- X)	MACRO	Decrement the variable X.
(* X...)	*EXTFN	Multiply the numbers X...
(/ X Y)	*EXTFN	Divide X with Y.
(ACOS X)	*EXTFN	Compute arc cosine of X.
(ASIN X)	*EXTFN	Compute arc sine of X.
(ATAN X)	*EXTFN	Compute arc tangent of X.
(COS X)	*EXTFN	Compute cosine.
(CEILING X)	*EXTFN	Compute ceiling of X.
(EXP X)	*EXTFN	Exponent e^X
(EXPT X Y)	*EXTFN	Compute exponent X^Y .
(FLOOR X)	*EXTFN	Compute largest integer $\leq X$.
(INTEGERP X)	*EXTFN	True if X is an integer.
(LOG X)	*EXTFN	Compute 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.
(MOD X Y)	*EXTFN	Return the remainder when dividing X with Y. X and Y can be integers or floating point numbers.
(NUMBERP X)	*EXTFN	True if X is number.
(RANDOM N)	*EXTFN	Generate a random integer between 0 and N.
(ROUND X)	*EXTFN	Round X.
(SQRT X)	*EXTFN	Compute the square root of the number X.
(SIN X)	*EXTFN	Compute sinus.
(TAN X)	*EXTFN	Compute tangent.

3.6. *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:

```

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

```



```
> (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 less than Y.
(<= X Y)	*LAMBDA	True if the number X is less than or equal to Y.
(= X Y)	*EXTFN	Tests if two numbers are the same. For example: (= 1 1.0) => T, while (EQUAL 1 1.0) => NIL
(> X Y)	*LAMBDA	True if the number X is greater than Y.
(>= X Y)	*LAMBDA	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	Test if X and Y have the same address.
(EQUAL X Y)	*EXTFN	Test 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 too (Sec. 9.7).
(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.

3.7. Arrays

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

```
> (setq 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 only supports 1-dimensional arrays (vectors) while CommonLisp allows arrays of any dimensionality.

Adjustable arrays (datatype ADJARRAY) 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:

```
> (setq a (make-array 3))
#(NIL NIL NIL)
> (adjust-array a 6)
#(NIL NIL NIL NIL NIL NIL)
> a
#(NIL NIL NIL)
> (setq 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.
(AREF A I)	*MACRO	Access element I of the array A. Enumeration starts at 0. Unlike CommonLisp, only one dimensional arrays are supported.
(ARRAY-TOTAL-SIZE A)	*EXTFN	Return the number of elements in the (one-dimensional) array A.
(ARRAYP X)	*EXTFN	True if X is an array (fixed or adjustable).
(ARRAYTOLIST A)	EXTFN	Convert an array A to a list.
(CONCATVECTOR X Y)	LAMBDA	Concatenate arrays X and Y.
(COPY-ARRAY A)	*EXTFN	Make a copy of the non-adjustable array A.
(ELT A I)	EXTFN	Same as (AREF A I).
(LISTTOARRAY L)	EXTFN	Convert a list to a non-adjustable array.
(MAKE-ARRAY SIZE :INITIAL-ELEMENT V :ADJUSTABLE FLG)	*MACRO	Allocate a one-dimensional array of pointers with SIZE elements. :INITIAL-ELEMENT specifies optional initial element values. If :ADJUSTABLE is true an adjustable array is created; the default is a non-adjustable array.
(PUSH-VECTOR A X)	EXTFN	Adjusts the array A with one element X at the end.
(SETA A I V)	EXTFN	Set the element I in the array A to V. Returns V. Same as (SETF (AREF A I) V).
(SWAP A E1 E2)	LAMBDA	Swap elements E1 and E2 in the array A.
(VECTOR X...)	*EXTFN	Make an array with elements X..

3.8. 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:

```
> (setq ht1 (make-hash-table))
#[HASHTAB 2547944]
> (setf (gethash "hello" ht1) "world")
"world"
> (gethash "hello" ht1)
NIL
> (setq ht2 (make-hash-table :test (function equal)))
#[HASHTAB 2548104]
> (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 empty table.
(GETHASH K HT)	*EXTFN	Get value of element with key K in hash table HT.
(HASH-BUCKET-FIRSTVAL HT)	EXTFN	Return the value for the first key stored in the hash table HT. What is the first key is undefined and depends on the internal hash function used.

(HASH-BUCKETS HT)	EXTFN	Compute the number of buckets in the hash table HT.
(HASH-TABLE-COUNT HT)	*EXTFN	Compute the number of elements stored in the 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 VAL V) on each key and value of the hash table HT.
(PUTHASH K HT V)	EXTFN	Set the value stored in the hash table HT under the key K to V. Same as (SETF (GETHASH K HT) V).
(REMHASH K HT)	EXTFN	Remove the value stored in the hash table HT under the key K.

3.9. *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 table 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:

```
> (setq bt (make-btree))
#[BTREE 3396632]
> (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 of element with key K in B-tree BT. Comparison uses COMPARE.
(MAKE-BTREE)	EXTFN	Allocate a new B-tree.

(MAP-BTREE BT LOWER UPPER FN)

EXTFN Apply ALisp 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 are 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. Same as (SETF (GET-BTREE K BT) V). V=NIL => marking element as deleted. **NOTICE** that the elements are not physically removed when V=NIL; they are just marked as deleted. To physically remove them you have to copy the B-tree.

3.10. 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 actually can replace many control structures in conventional programming languages. The map functions in Sec. 3.11 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 argument FN with X and Y as actual parameters and then adds together the result (i.e. $\text{sum2} = \text{fn}(x) + \text{fn}(y)$):

```
> (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 FUNCTION should be used (rather than QUOTE) when passing a functional argument, to be explained next.

3.10.1. Closures

In the example FUNCTION is used when passing a functional argument. 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
              (lambda (z)
                (expt z pow)))))) ← FUNCTION must be used here
SUMPOW
                                  ← lambda expression = anonymous function
                                  ← POW is free variable in lambda
> (sumpow 1 2 2)
5
```

Free lambda expressions [1] 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
             (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 1, Unbound variable: POW
When evaluating: POW
(FAULT EVAL BROKEN)
In SUM2 brk>:r

```

As you can see, the system warns that QUOTE is used instead of FUNCTION and then the variable POW is unbound when SUMPOW is called. The reason is that the system QUOTE returns its argument unchanged while FUNCTION makes a *closure* of its argument if it is a lambda expression. A closure is a special 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 in SUMPOW. Thus always use FUNCTION when passing functional arguments.

3.10.2. Applying functions with variable arity

FUNCALL does not work if we don't know until at run time the number of arguments of the function to call. In particular FUNCALL cannot be used if we want to call a function with variable arity, like + (plus). What we need is a way to construct a dynamic argument list before we call the function. For this the system function APPLY is used. For example, the function (COMBINEL X Y FN) applies FN on the elements of 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. COMBINEL could also have been written less efficiently 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

```

3.10.3. *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:

```
> (setq a 1)
1
> (eval '(list a))
(1)
> (eval (list a))      ← This fails because we are trying to evaluate the form (1)
Error 15, Undefined function: 1
When evaluating: (1)
(FaultEVAL BROKEN)
In *BOTTOM* brk>:r
> (list a)            ← This gives the same result as the Lisp top loop calls eval
(1)
```

EVAL is actually very seldomly used. It is useful when writing Lisp programming utilities, like e.g. the top loop or remote evaluation (Sec. 5.3.2). Avoid using EVAL unless you really need to, as the code executed by EVAL is not known until run-time and this is very unpredictable and prohibits compilation and program analysis. If possible, use FUNCALL and APPLY instead. In most other cases macros (Sec. 3.13) replace need for EVAL while at the same time producing compilable and analyzable programs.

3.10.4. *System functions for run-time evaluation*

Function	Type	Description
(APPLY FN ARGS)	*EXTFN	Apply the function FN on the arguments in the list ARGS.
(APPLYARRAY FN A)	EXTFN	Apply the Lisp function FN on the arguments in the array A.
(EVAL FORM)	*EXTFN	Evaluate FORM. 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...)) is equivalent to the CommonLisp read macro (also supported): #'(LAMBDA (X) FORM...).
(FUNCALL FN ARG1...)	*EXTFN	Call function FN with arguments ARG1...
(FUNCTION FN)	*SPECIAL	Make a closure of the function FN. Do not use QUOTE!

3.11. *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 in a functional programming style over data structures. They are often a good alternative to the more conventional iterative statements (Sec. 3.12.3). They are also often a good alternative to recursive functions as they don't eat stack as recursive functions do.

The classical 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 arguments 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
(ISOME L FN)	EXTFN	FN is function with two arguments X and TAIL. Apply FN on each element and its tail in list L. If FN returns true for some element in L then ISOME will return the corresponding tail of L. For example: (ISOME '(1 2 2 3) #'(LAMBDA (X TL)(EQ X (CADR TL)))) => (2 2 3) See also SOME.
(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.
(MAPFILTER FILT LST &optional OP)		

	EXTFN	If OP=NIL return the subset of the elements for which the filter function FILTER returns true. If OP is specified the result is transformed by applying OP on each element of the subset. For example: (MAPFILTER (FUNCTION NUMBERP) '(A 1 B 2) (F/1 (x)(1+ x))) => (2 3) See also SUBSET!
(MAPL FN L...)	*MACRO	Apply FN on each tail of the lists L...
(SUBSET L FN)	LAMBDA	Return the subset of the list L for which the function FN returns true.
(EVERY FN L...)	*MACRO	Return T if FN returns non-nil result when applied on every element in the lists L... in parallel.
(NOTANY FN L...)	*MACRO	Apply FN on elements in the lists L... in parallel. Return T if FN does not return true for any element.
(SOME FN L...)	*MACRO	Apply FN on elements in the lists L... in parallel. Return T if FN returns non-NIL value for some element.

3.12. Control Structures

Syntactic sugar and control structures are implemented in Lisp as macros and special functions. This subsection describes system functions, macros, and special forms.

3.12.1. Compound expressions

The compound 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 are also implicitly formed by lambda and LET expressions and many control structures described in the next section.

Function	Type	Description
(PROG1 X...)	*EXTFN	Return the value of the first form in X...
(PROG2 X...)	*EXTFN	Return the value of the second form in X...
(PROGN X...)	*EXTFN	Return the value of the last form in X...

3.12.2. Conditional expressions

Conditional expressions are special forms that evaluate expressions conditional on the truth value of some condition. The classical Lisp conditional expression is COND. For example:

```
> (setq x 1)
```

```

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

```

An alternative to COND is (IF PRED THEN ELSE). For example, the COND expression above can also be written:

```

> (if z (print "NO")
      (if y (progn (print "YES") 5)
           (print "NO")))
"YES"
5

```

The function PROG N has to be used to form compound expressions inside the IF. Such nested IFs are not recommended as they make the code difficult to read.

The CASE macro selects forms to evaluate conditional on the value of a test form. For example:

```

> (case (+ x 1)
    ((0 2) "YES")
    (1 "NO")
    (otherwise "NO"))
"YES"

```

← The test form
← Succeeds if (+ x 1) is either 0 or 2
← Succeeds if (+ x 1) is 1
← Default case
← X is 1

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

```

Evaluate TEST and match the value with each of the WHEN expressions. For the WHEN expression matching the value, the corresponding forms THEN... 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 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 P A B)

*SPECIAL If P then evaluate A else evaluate B.

(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.

3.12.3. Iterative statements

As in other programming languages Lisp provides iterative control structures, normally as macros. However, in most cases map functions (Sec. 3.11) 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]. Loop can be terminated with (RETURN VAL) in addition to the ENDTEST.
(DO* INITS ENDTEST FORM...)	*MACRO	As DO but the initializations INIT are done in sequence rather than in parallel.
(DOLIST (X L) FORM...)	*MACRO	Evaluate the forms FORM... for each element X in list L.
(DOTIMES (I N) FORM...)	*MACRO	Evaluate the forms FORM... N times. I varies from 0 to N-1.
(LOOP FORM...)	*MACRO	Evaluate the forms FORM... repeatedly. The loop can be terminated, and the result VAL returned, by calling (RETURN VAL).
(RETURN VAL)	*LAMBDA	Return value VAL from the block in which RETURN is called. A block can be a PROG-LET, PROG-LET*, DOLIST, DOTIMES, DO, DO*, LOOP or WHILE expression.
(RPTQ N FORM)	SPECIAL	Evaluate FORM N times. Recommended for timing in combination with the function TIMER.
(WHILE TEST FORM...)	MACRO	Evaluate the forms FORM... while TEST is true or until RETURN is called.

3.12.4. Non-local returns

Non-local returns allows to bypass the regular function application order. The classical functions for this are CATCH and THROW. (CATCH TAG FORM) evaluates TAG to a *catcher* which must be a symbol. Then FORM is evaluated and if the function (THROW TAG VALUE) is called 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)
```

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 call to CATCH with the <i>catcher</i> TAG that has called THROW directly or indirectly.

3.13. 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 make for loops, e.g. (for i 2 10 (print i)) prints the natural numbers from 2 to 10. FOR can be defined as a macro:

```
> (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
        (setq _var (1+ _var))))))

FOR
lisp 1> (for i 2 4 (print i)) ← Macros expanded by interpreter
2
3
4
NIL ← Value of FOR
```

When defining macros as in the example one normally have a code skeleton in which one replaces elements with actual arguments. In the example we use SUBAIR to do the substitution. A more convenient CommonLisp facility to define code skeleton is to use backquote (“`”), which is a variant of QUOTE where pieces can be marked for evaluation. Using backquote FOR could also have been written as:

```
> (defmacro for (var from to do)
  `(let ((, var , from))
    (while (<= , var , to)
      , do
      (setq , var (1+ , var))))

(FOR REDEFINED)
FOR
> (for i 2 4 (print i))
2
3
4
NIL
```

The backquote character ‘`’ marks the succeeding expression to be back quoted. In a back quoted expression the character ‘,’ indicates that the next expression is to be evaluated.

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 function MACROEXPAND can be used, normally in combination with pretty-printing with PPS (Sec 5). For example:

```
> (macroexpand '(for i 2 4 (print i)))
((LAMBDA (I) (WHILE (<= I 4) (PRINT I) (SETQ I (1+ I)))) 2)
> (pps (macroexpand '(for i 2 4 (print i))))
((LAMBDA (I)                                     ← PPS makes more readable printing of code
  (WHILE
    (<= I 4)
    (PRINT I)
    (SETQ I
      (1+ I))))
  2)
NIL
```

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. For example, it is not unusual to use macros to define functions where arguments are automatically quoted. For example, (PP FN1...Fn) pretty-prints function definitions and the following expression pretty-prints the definition of PP itself:

```
> (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 is a lambda function
```

MACROS are made 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, but each symbol has a special flag indicating that its definition is a macro.

The following functions are useful when defining macros:

Function	Type	Description
(BQUOTE X)	MACRO	BQUOTE implements ALisp's variant of the CommonLisp read macro ‘(back-quote). X is substituted for a new expression where ‘,’ (comma) and ‘,@’ (at sign) are recognized as special markers. A comma is replaced with the value of the evaluation of the form following the comma. The form following an at-sign is evaluated and ‘spliced’ into the list. For example, after evaluating (setq a '(1 2 3))

```
(setq b '(3 4 5))
then
'(a (b , a ,@ b))
or equivalently
(bquote(a (b , a ,@ b)))
both evaluate to
(a (b (1 2 3) 3 4 5))
Very useful for making Lisp macros.
```

(DEFMACRO NAME ARGS FORM...)

*SPECIAL Define a new MACRO.

(KWOTE X)

EXTFN Make X a quoted form. Good for making Lisp macros.

For example,

```
(KWOTE T) => T
```

```
(KWOTE 1) => 1
```

```
(KWOTE 'A) => (QUOTE A)
```

```
(KWOTE '(+ 1 2)) => (QUOTE (+ 1 2))
```

(KWOTED X)

EXTFN Return T if X is a quoted form. For example:

```
(KWOTED 1) => T
```

```
(KWOTED '(QUOTE (1))) => T
```

```
(KWOTED '(1)) => NIL
```

(MACRO-FUNCTION FN)

*EXTFN Return the function definition of FN if FN 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.

(PROGNIFY FORML)

LAMBDA Make a single form from a list of forms FORML.

(UNFUNCTION FORM)

LAMBDA Get the function definition of a functional expression.

3.14. 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 FIELD1...), for example:

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

DEFSTRUCT defines a new structure S with fields named FIELD1... pointing to arbitrary objects. DEFSTRUCT generates a number of macros and functions to create and update instances of the structure. New instances are created with

```
(MAKE-S :FIELD1 VALUE1 ...)
```

for example:

```
> (setq 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-FIELD S)
```

for example:

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

Fields are updated by combining SETF with an accessor function:

```
(SETF (S-FIELD S) VAL)
```

For example:

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

An object O can be tested to be a structure of type S using the generated function:

```
(S-P O)
```

For example:

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

3.15. *Miscellaneous functions*

Function	Type	Description
(ADVISE-AROUND FN CODE)	LAMBDA	Replace the body of function FN with form CODE where each * is substituted for the original body of FN. (So called aspect-oriented programming). CODE must be a single form. The variables of FN are available in CODE. If the function is an EXTFN the variable !ARGS is bound in CODE to a list of the actual arguments.
(CHECKEQUAL TEXT (FORM VALUE)...)	SPECIAL	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. If some evaluation of some FORM is not EQUAL to the corresponding VALUE an error notice is printed.
(DECLARE ...)	*MACRO	Dummy defined in ALisp for compatibility with CommonLisp.
(EVALLOOP)	EXTFN	Enter a Lisp top loop. Return to caller when function (EXIT) is called.
(EXIT)	EXTFN	Return from the Lisp top loop to the program that called it. In a stand-alone Amos II system EXIT is equivalent to QUIT. When Amos II is embedded in some other system EXIT will pass the control to the embedding system.
(ID X)	LAMBDA	ID is the identity function.
(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)	*EXTFN	Quit Lisp. If ALisp is embedded in another system it will terminate as well.
(ROLLOUT FILE)	EXTFN	Save the ALisp memory area (image) in file FILE. It can be restored by specifying FILE on the command line the next time ALisp is started.
(SETF PLACE VAL)	*MACRO	Update the location PLACE to become VAL. The location can be specified with, e.g., one of the functions AREF, GETL or GETHASH. It can also be a symbol, in which case SETF behaves like SETQ. A new <i>SETF macro</i> can be defined for a function call (FOO ...) by putting a form on the property list of FOO under the indicator SETFMETHOD. The SETF macro should have the format (LAMBDA (PLACE VAL) . . .). It is executed when (SETF (FOO ...) V) is called and thereby PLACE is bound to the list (FOO ...) and VAL to V. The SETF method should return the form to do the update.
(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 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 passing NIL as arguments. Notice that the SIZE can never be increased beyond the initial setting 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 too small C stack size.
(TYPENAME X)	EXTFN	Get the name of the datatype of object X.

3.16. Hooks

Hooks are lists of Lisp functions 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 &optional WHERE)
```

The Lisp expression FORM is inserted into a list of forms stored in the global variable AFTER-ROLLIN-FORMS, which 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 use the system function:

```
(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.
c:\torer>amos2 myimage.dmp ← Start Amos II with the saved image
amos2 myimage.dmp
Welcome! ← The initialization hook is evaluated.
Amos II Release 7, v7
Amos 1>
```

4. Time Functions

Time can be represented in ALisp either as *absolute time values* or *relative time values* (i.e. differences between time points). Time values are used for storing time stamps, measuring time intervals, or control system behaviour.

4.1. System clock

The behaviour of the system can be influenced based on time either by i) making the system sleep for time period, or ii) by running a background *timer function* at regular time intervals.

Function	Type	Description
(CLOCK)	EXTFN	Compute the number of wall clock seconds spent so far during the run as a floating point number.
(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.
(SET-TIMER FN PERIOD)	EXTFN	The function function SET-TIMER starts a <i>timer function</i> , which is a Lisp function called regularly by the system kernel. PERIOD specified the minimal interval 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.

4.2. Absolute Time Values

The ALisp datatype `TIMEVAL` represents time points. A `TIMEVAL` object has two components, *sec* and *usec*, representing seconds and micro seconds, respectively. A `TIMEVAL` object is printed as `#[TIMEVAL sec usec]`, e.g. `#[TIMEVAL 2 3]`.

The following Lisp functions operate on time points:

Function	Type	Description
<code>(MKTIMEVAL SEC USEC)</code>	EXTFN	Create a new <code>TIMEVAL</code> object.
<code>(TIMEVALP TVAL)</code>	EXTFN	Return T if <code>TVAL</code> is a <code>TIMEVAL</code> object, otherwise NIL.
<code>(TIMEVAL-SEC TVAL)</code>	EXTFN	Return the number of seconds for a given <code>TIMEVAL</code> object.
<code>(TIMEVAL-USEC TVAL)</code>	EXTFN	Return the number of micro seconds for a given <code>TIMEVAL</code> object.
<code>(T< T1 T2)</code>	EXTFN	Return T if the <code>TIMEVAL</code> T1 is less than the <code>TIMEVAL</code> T2, otherwise NIL.
<code>(T<= T1 T2)</code>	LAMBDA	Return T if <code>TIMEVAL</code> T1 is less than or equal to the <code>TIMEVAL</code> T2, otherwise NIL.
<code>(T> T1 T2)</code>	EXTFN	Return T if the <code>TIMEVAL</code> T1 is greater than the <code>TIMEVAL</code> T2, otherwise NIL.
<code>(T>= T1 T2)</code>	LAMBDA	Return T if the <code>TIMEVAL</code> T1 is greater than or equal to the <code>TIMEVAL</code> T2, otherwise NIL.
<code>(GETTIMEOFDAY)</code>	EXTFN	Return a <code>TIMEVAL</code> representing the wall time from a system call to C's <code>gettimeofday</code> . Useful for constructing time stamps.
<code>(TIMEVAL-TO-DATE TVAL)</code>	EXTFN	Translate a <code>TIMEVAL</code> <code>TVAL</code> into a date. A date is a seven element array where the first element is the Year, the second Month, the third Date, the fourth Hour, the Minutes, the Seconds, and the seventh Micro Seconds. All elements in the array are integers.
<code>(DATE-TO-TIMEVAL D)</code>	EXTFN	Translates a date <code>D</code> to a <code>TIMEVAL</code> .

4.3. Relative time values

Relative time values are represented by the datatype `TIME`. It has three components, hour, minute, and second. The following functions operate on relative times:

Function	Type	Description
<code>(MKTIME HOUR MINUTE SECOND)</code>	EXTFN	Construct a new <code>TIME</code> object.
<code>(TIMEP TM)</code>	EXTFN	Return T if <code>TM</code> is a <code>TIME</code> otherwise NIL.
<code>(TIME-HOUR TM)</code>	EXTFN	Return the number of hours given a <code>TIME</code> <code>TM</code> .
<code>(TIME-MINUTE TM)</code>	EXTFN	Return the number of minutes given a <code>TIME</code> <code>TM</code> .
<code>(TIME-SECOND TM)</code>	EXTFN	Return the number of seconds given a <code>TIME</code> <code>TM</code> .

4.4. *Relative Date Values*

Relative date values are represented by the datatype `DATE`. It has three components, year, month, and day. The following functions operate on dates:

Function	Type	Description
(MKDATE YEAR MONTH DAY)	EXTFN	Construct a new date.
(DATEP DT)	EXTFN	Return T if DT is a date otherwise NIL.
(DATE-YEAR DT)	EXTFN	Return the number of years.
(DATE-MONTH DT)	EXTFN	Given a date DT, return the number of months given a date DT.
(DATE-DAY DT)	EXTFN	Return the number of days given a date DT.

5. *Input and Output*

The I/O system is based on various kinds of *streams*. A 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 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 Amos II/Alisp systems. The storage manager allows the programmer to define new kinds of stream (Sec. 9.8). The default (= nil) file stream *standard output* prints to the console and *standard input* reads from the console.

Streams normally have functions providing the following operations:

Open a new stream, e.g. (`OPENSTREAM FILE MODE`) creates a new file stream.

Print bytes to the stream buffer. For example, (`PRINT FORM STR`) prints a form to a stream open for output and iterates through `FORM` converting encountered data structures to byte strings that are printed to the stream.

Read bytes from the stream buffer. For example, (`READ STR`) reads of form from a 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 a stream to its destination when (`FLUSH STR`) is called.

Close the stream, when (`CLOSESTREAM STR`) is called.

The following functions work on any kind of stream:

Function	Type	Description
(CLOSESTREAM STR)	EXTFN	Close stream STR.
<code>_DEEP-PRINT_</code>	GLOBAL	(Default T). Normally the contents of fixed size arrays and structures are printed by <code>PRINT</code> etc. This allows I/O of such

datatypes. However, when `_DEEP-PRINT_==NIL` the contents of arrays and structures are *not* printed. Good when debugging large or circular structures.

(FORMATL STR FORM...)	LAMBDA	This function is a simple replacement of some of the functionality of <code>FORMAT</code> in CommonLisp [1]. It prints the values of the forms <code>FORM...</code> on stream <code>STR</code> . A marker <code>T</code> among <code>FORM...</code> indicates a line feed while the string <code>"~PP"</code> makes the next element pretty-printed. For example: (FORMATL T "One: " 1 T) prints One: 1
(PPS S STR)	LAMBDA	Pretty-print expression <code>S</code> .
(PRIN1 S STR)	*EXTFN	Print the object <code>S</code> in the stream <code>STR</code> with escape characters and string delimiters inserted so that the object can be read with <code>(READ STR)</code> later to produce a form <code>EQUAL</code> to <code>S</code> .
(PRINC X STR)	*EXTFN	Print the object into the stream <code>STR</code> without escape characters and string delimiters.
(PRINT X STR)	*EXTFN	(<code>PRIN1 X STR</code>) followed by a line feed.
(READ STR)	*EXTFN	Read expression from stream <code>STR</code> . If <code>STR</code> is a string, the system reads an expression from the string. For example: (READ "(A B C)") => (A B C)
(READ-CHARCODE STR)	EXTFN	Read one byte from the stream <code>STR</code> and return it as an integer.
(REDIRECT-BASIC-STDOUTPUT FILE)	EXTFN	Redirect the standard output of ALisp to the specified file. In case the system is run under another system, e.g. inside a web server, standard output is often disabled and this function allows logging in a file instead. Notice that in such cases it is sometimes necessary to run this function when the system is started. One way to achieve this is to make an ALisp image where the <code>AFTER-ROLLIN-FORMS</code> (Section 3.16) redirects standard output.
(SPACES N STR)	LAMBDA	Print <code>N</code> spaces on the stream <code>STR</code> .
(TERPRI STR)	*EXTFN	Print a line feed on the stream <code>STR</code> .

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. For example:

```
> (setq s (openstream "foo.txt" "w"))
#[STREAM 3396656]
> (print '(hello world 1) s)
(HELLO WORLD 1)
> (closestream s)
#[STREAM 3396656]
> (setq 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 FILE)	*EXTFN	Delete the file named FILE. Returns T if successful.
(FILE-EXISTS-P NM)	*EXTFN	Return T if file named NM exists.
(FILE-LENGTH NM)	*EXTFN	Return the number of bytes in the file named NM.
(LOAD FILE)	*EXTFN	Evaluate the forms in the file named FILE.
(OPENSTREAM FILE MODE)	EXTFN	Open a file stream against an external file. MODE is the <i>file mode</i> 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 macros WITH-INPUT-FILE, and WITH-OUTPUT-FILE.
(PP FN...)	MACRO	Pretty-print the functions and variables FN... on standard output. Notice that arguments of PP are not quoted. For example: (PP PPS PPF).
(PPF L FILE)	LAMBDA	Pretty-print the functions and variables in L into the specified file. For example: (PPF '(PPS PPF) "pps.lsp")
(PRINTL X...)	LAMBDA	Print the objects X... as a list on standard output.
(TYPE-READER TPE FN)	EXTFN	Define the lisp function (FN TPE ARGS STREAM) to be a <i>type reader</i> for objects printed as #[TPE X...]. The type reader is evaluated by the ALisp reader when the pattern is encountered in an input stream. TPE is the type tag, ARGS is the list of argument of the read object (X...), and STREAM is the input stream.
(WITH-INPUT-FILE STR FILE FORM)	MACRO	Open stream STR for reading from FILE, evaluate FORM, and always close stream afterwards.
(WITH-OUTPUT-FILE STR FILE FORM)	MACRO	Open stream STR for writing to FILE, evaluate FORM, and always close stream afterwards.

5.2. Text streams

Text streams (datatype TEXTSTREAM) allow the I/O routines to work against dynamically expanded buffers instead of files. This provides an efficient way to destructively manipulate large strings. Text streams can also store bit sequences ('blobs'). The following ALisp functions are available for manipulating text streams:

Function	Type	Description
(MAKETEXTSTREAM SIZE)	EXTFN	Create a new text stream with an initial buffer size. The system automatically extends the initial size when necessary.
(TEXTSTREAMPOS TXTSTR)	EXTFN	Get the position of the read/print cursor in a text stream TXTSTR.

(TEXTSTREAMPOS TXTSTR POS) EXTFN Move the cursor to the specified position. This position is also updated by the regular Lisp I/O routines.

(TEXTSTREAMSTRING TXTSTR) EXTFN Retrieve the text stream buffer of TXTSTR as a string. **Notice** that this function cannot be used if the buffer contains binary data.

(CLOSESTREAM TXTSTR) EXTFN Reset the cursor to position 0, i.e. same as (TEXTSTREAMPOS TXTSTR 0).

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 thus used for sending forms between ALisp systems.

There is also a higher level *remote evaluation* mechanisms where system can be set up as a server evaluating incoming Lisp forms.

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)
```

Server side:

The first step on the server (receiving) side is to open a socket listening for establishments of 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==0 it means that the OS assigns a free port for incoming messages. If the OS assigns the port number of socket S can be obtained with the function:

```
(SOCKET-PORT S)
```

Then the server must then wait for clients to request connections by calling:

```
(ACCEPT-SOCKET S &optional TIMEOUT)
```

ACCEPT-SOCKET waits for the next OPEN-SOCKET call to the server 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 returns a new socket stream

to use for communication with the client issuing 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. HOSTNAME must not be NIL (which would indicate a server connection socket). The result of OPEN-SOCKET is a SOCKET object, which is a regular ALisp I/O stream that can be used by any I/O function. Thus, once OPEN-SOCKET is called the regular Lisp I/O functions can be used for communication. A SOCKET stream behaves like any other I/O stream.

Notice that 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)
```

POLL-SOCKET returns T if something 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 it can be closed and deallocated with:

```
(CLOSE-SOCKET S)
```

Notice that all pending data is lost when CLOSE-SOCKET is called. The garbage collector automatically calls CLOSE-SOCKET when a socket object is deallocated.

5.3.2. Remote evaluation

Amos II peers communicate through a protocol called *remote evaluation*. With remote evaluation Lisp forms are sent from one 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 Amos II peer behave as a remote evaluation server, accepting incoming forms to evaluate remotely.

An Amos II *name server* must be started on some host. The name server is an Amos II peer that keeps track of what peers listen to what ports for remote evaluation (see 2). To start a name server run on the desired host, execute the shell command:

```
amos2 -n
```

The peer needs to be registered in the Amos II name server used by the peer under some name NAME. This is done with:

```
(REGISTER-AMOS NAME &optional REREGISTER)
```

For example:

```
(REGISTER-AMOS "ME")
```

The NAME is a short *nick name* for the peer. The name server keeps track of the nick names of the peers and makes sure that no name collisions occur. REREGISTER==T means that the system should reregister NAME for this peer even if another peer is registered with the same nick name.

The OS environment variable AMOS-NAMESERVERHOST should be set to the IP host name of the computer where the name server is running. Default is the same host as the peer.

The remote evaluation server must be listening for incoming remote evaluation requests on the port on which it has been assigned for that purpose. This is done with:

```
(RUN-SERVER)
```

After RUN-SERVER is called the peer enters a remote evaluation server loop. The loop continues forever, or until interrupted with CTRL-C. If an error occurs during the remote evaluation the default behaviour is that the error message is shipped back to the caller. However, if the server is in debug mode (Sec . 7.1) server errors will be trapped there.

Client side:

On the client side, to ship a FORM for evaluation on an Amos II peer with nick name PEER, simply call:

```
(REMOTE-EVAL FORM PEER)
```

The result of the remote evaluation is shipped back to REMOTE-EVAL. REMOTE-EVAL blocks until the result is received. Errors occurring on server are shipped back to client.

For non-blocking messages use instead:

```
(SEND-FORM FORM PEER)
```

The difference to REMOTE-EVAL is that FORM is evaluated on PEER on its own; the client does not wait for the result and is thus non-blocking. Errors are NOT sent back. SEND-FORM is faster than REMOTE-EVAL, in particular when the messages are large. If you want to synchronize after many non-blocking messages sent with SEND-FORM, end with a REMOTE-EVAL. For example, the following form will return the number 1000, assuming that an Amos II peer named FOO is running:

```
(progn (send-form '(setq xx 0) 'foo)
      (dotimes (i 10000)(send-form '(1++ xx) 'foo))
      (remote-eval 'xx 'foo))
```

6. Error handling

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

```
(FAULT-EVAL 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 a unwind protection to be explained next.

6.1. Trapping exceptions

The special form UNWIND-PROTECT enables trapping error signals and clean up after error signals.

```
(UNWIND-PROTECT FORM CLEANUP)
```

The FORM is evaluated as usual until it is terminated, whether naturally or by means of a regular exit or an error signal. The cleanup form CLEANUP is then evaluated before control is handed back. Note that the cleanup form of an UNWIND-PROTECT is not protected by that UNWIND-PROTECT so errors produced during evaluation of CL can cause problems. The solution is to nest UNWIND-PROTECT. The function (HARDRESET) bypasses UNWIND-PROTECT and directly resets the system.

UNWIND-PROTECT traps any local or non-local exit, including error signals and THROW (Sec 3.12). For example, a throw form 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. UNWIND-PROTECT can be used to achieve this.

It is possible to trap all errors raised during the evaluation of a form by using the macro (CATCH-ERROR FORM CLEANUP). It evaluates FORM and returns the result of the evaluation, if successful. Should an error occur during the evaluation of FORM, then REPAIR is evaluated if supplied and an *error condition* is returned which looks like:

```
(:ERRCOND (ERRNO "errmsg" X))
```

For example:

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

The function (ERROR? X) tests if X is an error condition. It can be used for testing if CATCH-ERROR returned an error condition. The functions ERRCOND-ARG (the object causing the error), ERRCOND-NUMBER (the error number), and ERRCOND-MSG (the error message) are used for accessing error condition properties. The form REPAIR is evaluated if an error is raised. In REPAIR the variable `_ERROR-CONDITION_` is bound to the error condition.

6.2. Raising errors

The function (ERROR MSG X) print and error message MSG and raises an error for X. The error number is always -1 (user error).

To cause an error signal without any error message call (RESET).

As any other error these functions will go through the regular error management mechanisms. User errors can be caught with UNWIND-PROTECT or CATCH-ERROR.

6.3. *User interrupts*

After an interrupt is generated (e.g. CTRL-C) the system calls the Lisp function

```
(CATCHINTERRUPT)
```

By default CATCHINTERRUPT resets Lisp. In debug mode a break loop is entered when CTRL-C is typed.

For disable (delay) CTRL-C during evaluation of a FORM, use:

```
(DOUNITERRUPTED 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. CLEANUP is optional.
(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 an error condition.
(ERRCOND-MSG EC)	LAMBDA	Get the error message of an error condition.
(ERRCOND-NUMBER EC)	LAMBDA	Get the error number of an error condition.
(ERROR MSG X)	LAMBDA	Print message MSG followed by ': ' and X and then generates an error.
(ERROR? X)	LAMBDA	True if X is an error condition.
(FAULTEVAL ERRNO ERRMSG X FORM ENV)	LAMBDA	FAULTEVAL is called whenever the system detects an error. If the system runs in debug mode FAULTEVAL then enters a break loop (Sec. 7.1). If the system is not in debug mode FAULTEVAL prints the error message and calls (RESET).
(FRAMENO)	EXTFN	Return the frame number of the top frame of the stack.
(HARDRESET)	EXTFN	Does a 'hard' reset ignoring UNWIND-PROTECT. Called after fatal errors such as stack overflow.
(RESET)	EXTFN	Signals an error. The control is returned to the latest reset point. The reset point is either the ALisp top loop or the latest call to

UNWIND-PROTECT.

(UNWIND-PROTECT FORM CL) *SPECIAL UNWIND-PROTECT enables the user to clean up after a local or non-local exit.

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 Lisp mode. To enable Lisp debugging also in AmosQL mode call (DEBUGGING T). To disable debugging in Lisp mode 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 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: (SETQ `_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 1, Unbound variable: X          ← Run time error.
When evaluating: X
(FAULT EVAL BROKEN)                  ← System error break point.
In FIE brk>:bt                       ← Make backtrace.
FIE
FOO
(FAULT EVAL 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 arguments and values of broken function. The broken function is the function where the break loop is first entered.
:lvars Print argument bound at focused frame. It is the same as the broken function when the break loop is entered. Using stack moving commands the user can move down the stack to investigate other frames. The focused function is then the function at the focused frame. The focused function is part of break loop prompter.
:fp    Print file position of the focused function.
:ub    Unbreak the broken function.
:bt    Print a backtrace of functions called at the focused 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 10.
:btv   Print a backtrace of functions and arguments of functions.
:btv*  Print a long backtrace including all stack contents. To print the complete variable binding stack use the function (DUMPSTACK FRAME) that print everything pushed on the stack starting at frame number FRAME.
:c     Continue the broken evaluation. Possible only when current frame is at the broken function.
:eval  Evaluate broken function body. Possible only when current frame is at the broken function; otherwise see :rapply.
!value Lisp variable bound to value of evaluating the body of a broken function with :eval or :rapply.
:r     Reset to ALisp top loop.
:a     Reset to previous break point.

(:f FN) Move focused frame down the stack to first call to function FN and make it the focused function.
:nx    Move focused frame one step up the stack.
:pr    Move focused frame one step down the stack.
:org   Reset to frame of broken function.
:fr    Print focused frame.
:rapply Re-apply focused frame. This is like :eval but can be done also when focused frame is at not the broken function. Variable !VALUE holds result from evaluation as for :eval.

```

Does not work for special forms or functions with &REST arguments.

(:arg N) Get N:th argument in focused frame.

(:b VAR) Enter break loop when Lisp variable VAR is becoming bound after :c is specified.

(return x) Return value x from broken function.

The variables bound in the focused 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 FORM)
```

When HELP is called a break loop is entered where the user can investigate the environment with the usual break commands. The local variables in the environment where HELP was called are also available. The :EVAL command will evaluate the FORM. Its value is then inspectable through the variable !VALUE. Very good for debugging complex Lisp functions.

7.2. *Breaking functions*

Explicit break points can be put on the entry to and exit from Lisp functions 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

ALisp 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*, PRECOND:

```
(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)
```

However, 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 functions 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 replacing 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 periodic 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) update statistics on what code 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)
```

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 the most commonly called 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 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 &optional FILE)
```

The calling statistics are cleared with:

```
(CLEAR-FUNCTION-PROFILES)
```

Function profiling can be removed from specific functions with:

```
(UNPROFILE-FUNCTIONS FN...)
```

To remove all function profiles do:

```
(UNPROFILE-FUNCTIONS)
```

Analogously to conditional break points, *conditional function profiling* is supported by specifying 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 then arguments of EXTFNs are excluded from the backtrace. FRAME indicates 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 <i>debug mode</i> , where warning messages are printed and the system checks assertions. Turn off debug mode by calling with FLAG false. Notice that the system by default is in debug mode when Lisp mode is entered, but can be turned on by calling (DEBUGGING NIL).
(DUMPSTACK &optional FRAME)	EXTFN	Print all the contents of the variable binding stack. If FRAME is provided it specifies the starting stack frame number; otherwise the printing starts at the current top of the stack.

(HELP FORM)	MACRO	To insert explicit break points in Lisp code. Puts a break around evaluation of FORM. For example: (HELP "at FOO")
(IMAGE-EXPANSION RATE MOVE)	EXTFN	When the database image is full it is dynamically expanded by the system. This function controls the expansion. RATE is 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). FILE is optional.
(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 spent inside them (Sec. 7.4.2).
(REFCNT X)	EXTFN	Return the reference count of X. 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 the 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. Very useful for detecting memory corruption in C-code interfaced to the system. See also Section 10.4.
(START-PROFILE)	LAMBDA	Start statistical profiling of a Lisp program. (Sec. 7.4.1)
(STOP-PROFILE)	LAMBDA	Stop profiling the ALisp program. (Sec. 7.4.1)
(STORAGESTAT FLAG)	LAMBDA	If FLAG is true the top loop prints how much data was allocated and deallocated for every evaluated ALisp form in the ALisp top loop (or AmosQL statement in the Amos II top loop). 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. TAG is an optional title for the report. Good for finding storage leaks.
(TIMER FORM)	SPECIAL	Print the real time spent evaluating FORM. Often used in combination with RPTQ (Sec. 3.12).
(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 FLG)	EXTFN	Trace <i>all</i> function calls if FLG 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 associated with the symbol 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 or XEmacs. The most convenient way to develop Alisp code is to run from a shell within XEmacs. Emacs should be configured using the file `init.el`. It provides extensions to Emacs for finding Lisp code and for evaluating Lisp by ALisp. Place `init.el` in the initialization folder of Emacs (usually `/home/.emacs`) or XEmacs (usually `%HOME%\xemacs`).

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. Run there Amos II by issuing the shell command:

```
amos2
```

If you are developing Lisp code, enter Lisp mode with 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.

The code search facility requires the system to be installed with code search enabled. It is enabled if the environment variable `RELEASING` is NOT set when system is installed. Furthermore, in order to actually edit the source code all source files must reside where they were when the system was installed. Thus certain installations may not have access to the source code or to the program database.

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 of PRINTL using PP:

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

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)
CADDR C:/AmosNT/lsp/orginit.lsp 47
" "
CDDDDR C:/AmosNT/lsp/orginit.lsp 45
" "
CDDDR
EXTFN
```

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

```
> (pp cdddr)
(DEFUN 'CDDDDR #[EXTFN1 CDDDDR])
(CDDDDR)
```

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 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
" "
```

Function	Type	Description
(DOC FN)	LAMBDA	Return the documentation string for a function.
(FP FUNCTION)	LAMBDA	Print the file position of a function definition. 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. This can be slow.
(CALLING FN &optional LEVELS FILE)	LAMBDA	Print the file positions for the functions calling the function FN. LEVELS specifies how many levels of functions that call FN indirectly are printed (default 1). FILE prints to a file.
(CALLS FN &optional LEVELS FILE)	LAMBDA	Print the file positions for the functions called from function FN. LEVELS specifies how many levels of functions that are called indirectly by FN are printed (default 1). FILE prints to a file.
(USING S)	LAMBDA	Print the file positions for the functions whose definitions contain the symbol S. S is usually a variable name.
(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 1).

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
```

9. The Storage Manager

ALisp is integrated with a storage management subsystem callable from C. The storage manager is responsible for allocation and deallocation of *physical objects* inside the database image. The C implementor has the choice of allocating data *persistently* inside the database image by using a set of primitives provided by the storage manager. Alternatively data can be allocated *transiently* by using the usual C routines malloc, etc. Persistency in this case means that data allocated in the database image can be saved on disk and later restored. Persistent data is stored on disk when the user issues the AmosQL statement save or call Lisp's ROLLOUT function. The image is restored when restarting the system with the image file as command line argument. By contrast, transient data disappears when the system is exited. The C/C++ programmer can define own persistent data structures by using a set of storage manager primitives.

Another important service of the storage manager is to provide a garbage collection subsystem that automatically deallocates persistent memory no longer in use in the database.

The storage manager is actually independent of the ALisp interpreter. Its description is in this document as knowledge of the storage manager is needed for defining external Lisp function.

9.1. Handles

All access to physical objects is made through *handles* which are indirect identifiers for physical data records in C. The representation of handles is currently unsigned integers. In order to make the application code both fast and independent on the internal representation of handles, the handles are always manipulated through a set of C macros and utility functions. The interface is furthermore connected to an automatic garbage collector so that data no longer used is reclaimed when using those macros/functions. The interface between the storage manager is defined by the header file `storage.h`.

Handles to persistent objects must be declared of C type `oidtype` and *must* always be initialized to the global C constant `nil.oidtype` and `nil` are defined in `storage.h`. For example:

```
oidtype myhandle = nil;
```

This equivalent to the C macro `dcloid`:

```
dcloid(myhandle);
```

9.2. Physical Objects

With every handle there is an associated C structure, the *physical object*, stored in the database image and holding the *value* of the handle. The physical data objects are C structures containing the data to be stored persistently together with a *physical type* identifier identifying the type of the object. The layout of the physical data object depends on the datatype. However, the first two bytes of a physical object are *always* reserved for the system; the succeeding bytes are used for storing the data. Every persistent data item must be represented as physical objects, including literals such as integers and strings. For example, integers are represented by this structure:

```
struct integercelltype
{
    objtags tags;
    short int filler;
```

```
    int integer;
};
```

The field `tags` is used by the system, the field `integer` stores the actual integer value, and `filler` aligns the integer to a full-word.

Every physical type has an associated *type identifier* number and a unique *type name* string known to the storage manager. A number of physical types are predefined, including LIST, SYMBOL, INTEGER, REAL, EXTFN (ALisp function defined in C), CLOSURE (internal ALisp closures), STRING, ARRAY (1D fixed size arrays), STREAM (file streams), TEXTSTREAM (streams to text buffers), HASHTAB (hash tables), ADJARRAY (dynamically extensible arrays), and BINARY (bit strings). In `storage.h` there are structure definitions defined for the physical representation of most of the built-in types. The convention is used that if the type is named `xxx` the template has the name `xxxcell`, e.g. REAL has a template named `realcell`, etc. The type identification numbers for most built-in types are also defined as C macros in `storage.h`, with the convention that a type named `xxx` has a corresponding identification number `XXXTAG` if it is defined as a C macro or `xxxtag` if it is a global C variable.

The C/C++ programmer can extend the built-in set of physical datatypes with new persistent data structures through the C function `a_definetype`, explained below. It defines to the storage manager the properties of the new data type.

9.3. **Logical Data Objects**

Notice the difference between *physical* and *logical objects*: Physical data objects are C record structures stored in the database image while logical data objects are object descriptors references through AmosQL. Logical data objects are internally represented by one or several physical data objects. For example, Amos II objects of logical data type INTEGER are directly represented by the above mentioned physical data objects also named INTEGER. Similarly, other simple literal objects (e.g. real numbers and strings) are internally represented as directly corresponding physical objects. More complex objects, e.g. the logical datatype FUNCTION, are represented by data structures consisting of several physical objects of different types. *Surrogate objects* in AmosQL are represented as physical objects of a particular kind named `OID` describing properties of the logical object identifier of the surrogate. One property of an `OID` object is a numeric identifier maintained by the storage manager; another one is a handle referencing the Amos II type(s) of the logical object. References to `OID` objects are very common in the database, e.g. to represent arguments or values of functions, extents of types, etc.

The AmosQL user cannot directly manipulate physical objects; they can only be manipulated in C/C++ or ALisp.

9.4. **Dereferencing**

In order to access or change the contents a physical object given a handle it has to be converted from a handle into a C pointer to the corresponding physical object in the database image. This process is called to *dereference* the handle. The dereferencing of physical data objects is very fast and does not involve any data copying; it involves just an offset computation.

Once the physical object has been dereferenced its contents can be investigated by system provided C macros and functions or directly by C pointer operations. The header of a physical object (type `objtags`) is maintained by the storage manager. It contains the identification of its physical type (1 byte) and a *reference counter* (1 byte) used by the automatic garbage collector.

The following C macro dereferences a handle:

```
dr(x, str)
```

`dr` returns the address of the record referenced by the handle `x` casted as a C struct named `str`.

For example, the following C function prints an integer referenced by the handle `x`:

```
void printint(oidtype x)
{
    struct integercell dx = dr(x, integercell);
    printf("x = %d\n", dx->integer);
    return;
}
```

Notice that the parameter `x` must be a handle referencing an object of type `INTEGER`, otherwise the system might crash. To make `printint` safe it therefore should check that `x` actually references an integer. The following C macro can be used for investigating the type of a physical object handle:

```
a_datatype(x)
```

`a_datatype` returns the type identifier of a handle `x`.

For example, the function `printint2` checks that `x` actually is an integer before printing its value:

```
void printint2(oidtype x)
{
    if(a_datatype(x) == INTEGERTAG)
        printf("x = %d\n", dr(x, integercell)->integer);
    else printf("X is not an integer\n");
    return;
}
```

WARNING: Storage manager operations may invalidate dereferenced C pointers because the dereferenced objects might move to other memory locations when the image is expanded. Thus dereferenced pointers may become incorrect once a system feature that causes the image to expand is called. Object allocation is the only system operation that may cause this. Thus, if a system function is called that is suspected to do object allocation (most do), the dereferencing *must* be redone. It is therefore safer to always dereference through `dr` as in `printint2` rather than saving the C pointer as in `printint`.

9.5. Assigning handles to locations

In order for the storage manager and garbage collector to function correctly, assignments of C locations (variables or fields) of type `oidtype` *must* use the C macro

```
a_setf(location, value);
```

`a_setf` corresponds to an assignment, `location=value`, but, unlike an assignment, it also updates the reference counter of `value` so it is increased after the assignment. The reference counter increment indicates to the system that the C location `x` holds a reference to the physical object and it therefore cannot be deallocated until the handle `location` is *released*. A handle location `x` is released with the C macro:

```
a_free(x)
```


If the reference counter of a physical object reaches 0 the physical object is passed to the garbage collector for deallocation. Thus, unlike the C function *free*, *a_free* will deallocate *x* only when there is no other location holding a reference to it.

To handle reassignments of locations correctly, *a_setf* releases the handle *previously* referenced from the handle *x*; thus it *decreases* the reference counter of handles *previously* referenced from the handle *x*.

Lisp symbols (e.g. *nil*) are not garbage collected and thus not reference counted.

Notice that locations *must* be assigned to some handle before *a_setf* can be used, otherwise the system is likely to crash when trying to release a non-existing handle. It is therefore necessary to *always* initialize C handle locations to the constant *nil* (referencing the symbol *NIL*) when declaring them. An alternative it to use the macro *a_let* the first time a location is assigned a handle. It assumes the old value of *x* was uninitialized:

```
a_let(location,value)
```

9.6. *Allocating physical objects.*

Physical objects can be allocated only through a number of storage manager primitives (not through e.g. *malloc*). When a physical object is allocated it initializes the reference counter to 0. The built-in datatypes have allocation macros and functions defined in *storage.h*, e.g.:

```
mkinteger (xx)  allocates a new integer object.
mkreal (xx)    allocates a new double precision real number object.
mkstring (xx)  allocates a new string object.
mksymbol (xx)  allocates or gets the symbol named capitalized xx.
cons (x,y)     allocates a new list cell.
```

For example, the following C function adds two integers:

```
oidtype add(oidtype x, oidtype y)
{
    int sum;

    if(a_datatype(x) != INTEGERTAG ||
       a_datatype(y) != INTEGERTAG)
    {
        printf("Cannot add non-integers\n");
        exit(1);           ← Should call error manager here instead.
    }
    sum = dr(x,integercell)->integer + dr(y,integercell)->integer;
    return mkinteger(sum);
}
```

The following code fragment allocates two integers, calls *add*, and prints the sum.

```
{
    oidtype x=nil, y=nil, s=nil; // handles must be initialized to nil!

    a_setf(x,mkinteger(1)); // assign x to new integer 1
    a_setf(y,mkinteger(2)); // assign y to new integer 2
```

```

    a_setf(s,add(x,y));    // assign s to new integer being sum of a x and y
    printf("The sum is %d\n",dr(s,integercell)->integer);
    a_free(s);           // release locations s, x, y
    a_free(x);
    a_free(y);
}

```

In `storage.h`, for each built-in storage type there is a C constant (upper case) or a variable (lower case) containing the identifier for the type.

Type-name	constant/variable	Short description
LIST	LISTTYPE	Lists
SYMBOL	SYMBOLTYPE	Symbols
INTEGER	INTEGERTYPE	Integers
REAL	REALTYPE	Double precision reals
EXTFN	EXTFNTYPE	ALisp function in C
CLOSURE	CLOSURETYPE	ALisp function closure
STRING	STRINGTYPE	Strings
ARRAY	ARRAYTYPE	1D Arrays
STREAM	STREAMTYPE	File streams
TEXTSTREAM	TEXTSTREAMTYPE	String streams
SOCKET	sockettype	Socket streams
HASHTAB	HASHTYPE	Hash tables
HASHBUCKET	HASHBUCKETTYPE	Internal to hash tables
OID	objecttag	Object identifiers
HISTEVENT	histeventtype	Update events

For most built-in datatypes there are C macros or functions for construction and access. For example, to allocate a new handle of type `STRING` with the content "Hello world" you can use the macro `mkstring` that returns a handle to the new string:

```

{
    dcloid(mystring);
    ...
    a_setf(mystring,mkstring("Hello world"));
    ...
    a_free(mystring);
};

```

To *dereference* a handle referencing a `STRING` object the macro `getstring` can be used:

```

{
    dcloid(mystring);
    char *mystringcont;

    a_setf(mystring,mkstring("Hello world"));
    mystringcont = getstring(mystring);
    printf("%s\n",mystringcont);
    a_free(mystring);
};

```

The following are examples of C library functions and macros used for manipulating the built-in data types:

```

oidtype mkinteger(int x)           (macro) Construct handle for a new integer
int integerp(oidtype x)          (macro) TRUE if X is a handle for an integer

```

<code>int getinteger(oidtype x)</code>	(macro) Dereference a handle for an integer
<code>oidtype mkreal(double x)</code>	(macro) Construct handle for a new real
<code>int realp(oidtype x)</code>	(macro) TRUE if <code>x</code> is a handle for a real
<code>double getreal(oidtype x)</code>	Dereference a handle for a real
<code>oidtype mkstring(char *x)</code>	(macro) Create handle for a new string
<code>int stringp(oidtype x)</code>	(macro) TRUE if <code>x</code> is a handle for a string
<code>char *getstring(oidtype x)</code>	(macro) Dereference a handle for a string
<code>oidtype new_array(int size,oidtype init)</code>	Construct handle for a new array with elements <code>init</code>
<code>int arrayp(oidtype x)</code>	TRUE if <code>x</code> is a handle for an array
<code>int a_arraysize(oidtype arr)</code>	return the array size
<code>oidtype a_seta(oidtype arr,int pos,oidtype val)</code>	Set an array element
<code>oidtype a_elt(oidtype arr,int pos)</code>	Retrieve array element
<code>oidtype a_vector(oidtype x1,...,xn,NULL)</code>	Create a new array and its elements <code>x1 ... xn</code> .
<code>oidtype cons(oidtype x,oidtype y)</code>	Create handle for a new list cell
<code>int listp(oidtype x)</code>	(macro) TRUE if <code>x</code> is a list cell
<code>oidtype hd(oidtype x)</code>	(macro) Head of list cell
<code>oidtype tl(oidtype x)</code>	(macro) Tail of list cell
<code>oidtype a_list(oidtype x1,...,xn,NULL)</code>	Create new list of <code>x1 ... xn</code>
<code>oidtype mksymbol(char *x)</code>	(macro) Create a new symbol
<code>int symbolp(oidtype x)</code>	(macro) TRUE if <code>x</code> symbol
<code>oidtype globval(oidtype x)</code>	(macro) Get global value of symbol.
<code>char *getpname(oidtype x)</code>	(macro) Get print name of symbol
<code>a_print(oidtype x)</code>	Print object of any type
<code>oidtype t</code>	Symbol <code>T</code> representing TRUE
<code>oidtype nil</code>	Symbol <code>NIL</code> representing empty list and FALSE

9.7. Storage types

This subsection describes how to introduce new physical storage types into Amos II. This is required when new C data structures need to be defined for Amos II.

In `storage.h` the basic built-in physical storage type tags are declared as macros. The include file also contains the record templates for each storage type.

There is a global *type table* which associates a number of optional C functions with each physical object type. A new storage type is introduced into the system (thus expanding the type table) with a call to the C function `a_definetype`:

```
int a_definetype(char *name,
                void (*dealloc_function) (oidtype),
```

```
void (*print_function) (oidtype,oidtype,int))
```

`a_definetype` adds a new type named `name` to the type table and returns the new type identifier as an integer.

`dealloc_function` is a required C function taking an object of the new type as argument. It is a *destructor* called by the garbage collector when the object is deallocated. It should then release all locations referenced by the object and call storage manager primitives to deallocate the storage occupied by the object.

`print_function` is an optional *print function* called by PRINT to provide a customized printing of physical objects of the new type. See section x.

9.8. Streams

ALisp has several datatypes representing streams:

STREAM	represents regular C file streams.
TEXTSTREAM	represents streams over buffers in the database image.
SOCKET	represents socket streams for communication with other ALisp systems.
PIPE	represent pipes for communication between child and parent processes in Unix.

The following system standard streams are defined:

<code>oidtype stdinstream</code>	for C's standard input stream
<code>oidtype stdoutstream</code>	for C's standard output stream
<code>oidtype stderrstream</code>	for C's standard error stream

Streams are physically represented as other data types but with some special stream attributes in the beginning of the structure template:

```
struct xxxcell
{
  objtags tags;
  short int bytes;           /* Total size of object in bytes, incl. header */
  char autoflush;           /* Flush after each item and new line */
  char filler[3];           /* Unused flags */
  int line_num;             /* Current line number */
  oidtype logstream;       /* Stream to copy input to if non-NIL */
  /*** end of stream header ***/
}
```

The attributes above must always be present for stream templates. Additional specific attributes can be added after the end of the stream header. Once the data type has been defined using `definetype` the newly created type can be made into a stream by a call to `a_define_stream_implementation`:

```
int a_define_stream_implementation(int tag, /* Storage type */
                                   int(*getc)(oidtype),
                                   int(*ungetc)(int,oidtype),
                                   int(*feof)(oidtype),
```

```

int(*puts)(char*,oidtype),
int(*putc)(int,oidtype),
int(*fflush)(oidtype),
int(*fclose)(oidtype));

```

The first argument, `tag`, is the type tag (returned by `defintype`) of the type to be made a stream. Each stream should have the following associated functions (methods):

```

int getc(oidtype stream)    Returns the next character in stream.
int ungetc(int c, oidtype stream)
                            Put back character c in stream.
int feof(oidtype stream)   Return TRUE if end-of-file reached.
int putc(int c, oidtype stream)
                            Write character c to the stream
Int readbytes(oidtype stream, void *block, unsigned int len)
                            Read a block of data from the stream. The slower putc method is used if
                            this method is NULL.
int writebytes(oidtype stream, void *block, unsigned int len)
                            Write a block of data to the stream. The slower getc method is used if this
                            method is NULL.
int fflush(oidtype stream) Flush stream buffer contents.
int fclose(oidtype stream) Close the stream.

```

Once these methods are defined and registered the user can use the following generic stream functions to manipulate the new stream:

```

int a_getc(oidtype stream);           Read one character
int a_ungetc(int c, oidtype stream);  Unread one character
int a_puts(char *str,oidtype stream); Write string
int a_writebytes(oidtype stream, void *buff, unsigned int len);
                                       Write block
int a_putc(int c, oidtype stream);     Write a character
int a_readbytes(oidtype stream, void *buff, unsigned int len);
                                       Read block
int a_fclose(oidtype stream);          Close stream
int a_feof(oidtype stream);           Test for end-of-file
int a_fflush(oidtype stream);         Flush stream buffer

```

The performance of stream management can be improved by moving bulks of data to or from the stream through calls to `a_printbytes` and `a_readbytes`. If the corresponding methods are not registered with a stream, writing to and reading from the stream is slower.

9.8.1. Marshalling objects

Streams are often used for writing object in such a format that they can later be restored by reading. This is particularly important when using streams to communicate data between ALisp peers, e.g. using sockets (Section 5.3). The Lisp function `PRINT` prints object structures on a stream in such a format (S-expression) that copies of the objects can later be restored by using `READ`. This `PRINT` and `READ` are Lisp's generic (de-)marshalling functions. Lisp's S-expression notation provides marshalling and demarshalling for the basic Lisp datatypes. In addition

customized (de-)marshalling can be specified for user defined storage type, as will be described below.

In C the following functions can be used for (de-)marshalling S-expressions:

```
oidtype a_read(oidtype stream)
    Read (unmarshal) S-expression from stream
oidtype a_print(oidtype s) Print S-expression a followed by a line feed on stdoutstream, normally
    for debugging.
oidtype a_printobj(oidtype s, oidtype stream)
    Print S-expression s followed by a line feed as delimiter on stream.
oidtype a_prinl(oidtype s, oidtype stream, int princflg)
    Print S-expression s on stream. If princflg is FALSE the printout be
    marshalled for subsequent reading; if princflg is TRUE object will be
    written as PRINC (sec. xx) and not readable. Notice that, as no delimiter is
    inserted as with a_printobj, it is up to the user to ensure proper object
    delimitation is.
oidtype a_terpri(oidtype stream)
    Write a line feed on stream.
```

10. *Interfacing Lisp with C*

An ALisp function can be implemented as a C function and C functions can call ALisp functions. ALisp and C can also share data structures without data copying or transformations. The error management in ALisp can be utilized in C as well for uniform and efficient error management.

In order to interface ALisp with C/C++ you need to download the Amos II development version. You must include the file `alisp.h` in your C program. In the development version, the file `democpp.cpp` contains a simple C program that calls ALisp and where ALisp also calls C.

This section describes how to call C functions from ALisp, and how to call ALisp functions from C.

10.1. *Calling C from Lisp*

As a very simple example of an external Lisp function we define an ALisp function HELLO which prints the string 'Hello world' on the standard output. It has the C implementation:

```
#include "alisp.h"
oidtype hellofn(bindtype env)
{
    printf("Hello world\n");
    return nil;
}
```

The include file `alisp.h` contains all necessary declarations for implementing external Lisp functions in C; External Lisp function definitions must always return handles of type `oidtype`. Do not forget the return statement!

In order to be called from Lisp, an external Lisp function implementation has to be registered with a symbolic ALisp name, in this case the symbol HELLO, by calling:

```
extfunction0("HELLO",hellofn);
```

A system convention is that an external Lisp function named XXX is named xxxfn in C, as for HELLO.

The call to register an external Lisp function should be done in a main C program, the *driver program*, after the system has been initialized (i.e. after `init_amos` or `a_initialize` is called). The following driver program initializes the system, registers HELLO, and calls the ALisp read-eval-print loop with prompter string 'Lisp>'.
'

```
#include "alisp.h"

oidtype hellofn(bindtype env)
{
    printf("Hello world\n");
    return nil;
}

void main(int argc, char **argv)
{
    init_amos(argc,argv);
    extfunction0("HELLO",hellofn);
    evalloop("Lisp>");
}
```

When the above program is run the user can call HELLO from the read-eval-print loop by typing

```
(hello)
```

10.1.1. *Defining external Lisp functions in C*

Lisp functions can be implemented as *external Lisp functions* in C. An external ALisp function `fn` with optional arguments `x1`, `x2`, ..., `xn` must have the following signature in C:

```
oidtype fn(bindtype env,oidtype x1,oidtype x2,..,oidtype xn)
```

The first argument `env` is the *binding environment* to be used by the system for error handling, memory management, and other things.

For example, the following function implements an ALisp function to add two numbers:

```
oidtype addfn(bindtype env, oidtype x, oidtype y)
{
    int ix, iy, r; // will hold integer values of x, y and result

    IntoInteger(x,ix,env); // Retrieve value of integer x into ix and raises
                          // ALisp error if x is not an integer object
    IntoInteger(y,iy,env);
    r = ix + iy;
    return mkinteger(r);
}
```

`addfn` is registered with

```
extfunction2("add", addfn);
```

The number '2' after 'extfunction' indicates that this ALisp function takes two arguments.

External Lisp functions need to be very careful to check the legality of the handles they receive, so that the system never crashes. To check that a handle is of an expected type use the C macro:

```
OfType(x, tpe, env)
```

A standard error will be generated if *x* does not have the type tag *tpe*. (For integers the above used macro `IntoInteger` is a convenient alternative).

External Lisp functions are *registered* (assigned to ALisp symbols) by calling a system C function:

```
extfunctionX(char *name, Cfunction fn);
```

name is the ALisp name for the external Lisp function

fn is the address of the C function.

Different versions of `extfunctionX` are available depending on the arity *X* of the external Lisp function. For example,

```
extfunction2("add", addfn);
```

There are corresponding ALisp registration functions for functions with arity 0, 1, 2, 3, 4, 5 named `extfunction0`, `extfunction1`, etc.

When a physical object handle whose reference counter has been managed by `a_setf` is to be returned from a C-function the following C-macro should be used:

```
a_return(x);
```

`a_return` returns *x* from the C-function after the reference counter of *value* has been decreased *without* deallocating *x* if the counter reaches 0.

For example, the following external Lisp function calls `addfn` twice to sum three integers:

```
oidtype add3fn(bindtype env, oidtype x, oidtype y, oidtype z)
{
    oidtype s=nil;

    a_setf(s, addfn(env, x, y));
    a_setf(s, addfn(env, s, z));
    a_return(s);
}
```

The variable *s* holds the result from `add3fn`. If it had been returned by the usual statement

```
return(s);
```

the result object would never be released from the location *s* and would therefore never be deallocated.

For example, the following function reverses a list:


```

oidtype reversefn(bindtype env, oidtype l)
{
    oidtype lst=nil, res = nil;

    a_setf(lst,l);
    while(listp(lst))
    {
        a_setf(res,cons(hd(lst),res));
        a_setf(lst,tl(lst));
    }
    a_free(lst);
    a_return(res);
}

```

Register REVERSE with:

```
extfunction1("REVERSE",reversefn);
```

WARNING: Never try to assign C function parameters (such as `l` in the example) with `a_setf`; it will clobber the garbage collector. Instead the parameter `l` is assigned to the local variable `lst` in order to subsequently use `a_setf`.

WARNING: The C implementation of an external Lisp function must always return a legal handle, otherwise the system might crash. It is therefore recommended to run the system in 'debug mode' while testing external Lisp function where the system always checks the legality of data passed to ALisp from C.

10.1.2. Variable arity external Lisp functions

Variable arity external functions accept any number of arguments. External Lisp functions with more than 5 arguments also need to be defined as variable arity functions. Variable arity external Lisp functions have the signature:

```
oidtype fn(bindtype args,bindtype env)
```

where `env` is the binding environment for errors, and `args` is a binding environment representing the actual arguments of the function call. To access argument number `i` use the C macro:

```
nthargval(args,i)
```

The arguments are enumerated from 1 and up.

The C function

```
int envarity(bindenv args)
```

returns the actual arity of the function call.

For example, the following ALisp function `sumfn` adds an arbitrary number of integer arguments:

```

oidtype sumfn(bindtype args,bindtype env)
{
    int sum=0, arity = envarity(args), i, v;

```

```

    for(i=1;i<=arity;i++)
    {
        IntoInteger(nthargval(args,i),v,env);
        sum = sum + v;
    }
    return mkinteger(sum);
}

```

Variable arity functions are registered to the system with `extfunctionn`:

```
extfunctionn("SUM",sumfn);
```

The Lisp function `LIST` has the following implementation:

```

oidtype listfn(bindtype args,bindtype env)
{
    oidtype res=nil;
    int arity=envarity(args), i;

    for(i=arity;i>=1;i--)
    {
        a_setf(res,cons(nthargval(args,i),res));
    }
    a_return(res);
}

```

Notice how the iteration over the arguments is done in reverse order to get the correct list element order.

10.1.3. *Defining special forms*

Special forms are external Lisp functions whose arguments are not evaluated by the ALisp interpreter when the C implementation function is called.

C functions implementing special forms have the signature:

```
oidtype fn(bindtype args,bindtype env)
```

Analogous to no variable arity functions the macros `envarity` and `nthargval` can be used to investigate the actual arguments. The difference is that `nthargval` here returns the *unevaluated* value, unlike for variable arity functions where evaluated values are returned.

For example, the following C function implements the ALisp special form `QUOTE`:

```

oidtype quotefn(bindtype args, bindtype env)
{
    return nthargval(args,1);
}

```

Special forms are registered using `extfunctionq`:

```
extfunctionq("QUOTE",quotefn);
```

For evaluating unevaluated forms this system function can be used:

```
oidtype evalfn(bindtype env, oidtype form)
```

For example, the following C function implements the special form (WHILEA PRED FORM1 FORM2 ...) that iteratively executes FORM1 etc. while PRED is non-nil:

```
oidtype whileafn(bindtype args, bindtype env)
{
    oidtype cond = nil, v = nil;
    int arity = envarity(args), i;

    a_setf(cond, nthargval(args, 1));
    for(;;)
    {
        a_setf(v, evalfn(env, cond)); /* Evaluate condition */
        if(v == nil) /* Condition false */
        {
            a_free(v); /* Release v and cond before returning */
            a_free(cond);
            return nil;
        }
        for(i=2; i<=arity; i++)
        {
            a_setf(v, evalfn(env, nthargval(args, i)));
        }
    }
}
```

Notice that `v` and `cond` must be released before the function is exited. Furthermore, the above definition is not fully correct because if `evalfn` fails, because of some logical error in the evaluated form, the error management system will make `evalfn` never return. Thus, in case of an error in the evaluation, the storage referenced by `v` and `cond` will never be deallocated. Another version of `whilea` which also manages this memory deallocation correctly will be presented in the next section.

10.2. Error management in C

ALisp has its own error management system integrated with the storage manager. In order for the storage manager to correctly release data after failures, abnormal function exits should always use the system error management, rather than e.g. directly calling C's `longjmp` or C++ error management.

10.2.1. Unwind Protection

To unconditionally catch failed operation the *unwind protect* mechanism is used. This is necessary sometimes to guarantee that certain actions are performed even if some called function terminates abnormally. For example, space may need to be deallocated or files be closed. For this purpose ALisp provides an *unwind-protect* feature in C, similar to what is provided in Lisp (Sec. 6.1). Unwind protection is provided through the following three macros:

```
{unwind_protect_begin; /* Always new block */
    main code
    unwind_protect_catch; /* This statement MUST ALWAYS be executed */
    unwind code
unwind_protect_end;} /* Will continue abnormal evaluation */
```

The `main` code is the code to be unwind protected. The `unwind` code is always executed both if the main code fails or succeeds. In the `unwind` code, a flag, `unwind_reset`, is set to `TRUE` if the code is executed as the result of an exception. The `unwind` code is executed outside the scope of the current unwind protection. Thus, exceptions occurring during the execution of `unwind` code is unwound by the next higher unwind protection.

WARNING: The `unwind_protect_end` code *must* be executed; never return directly out of the main code block. If `unwind_protect_end` is not executed after an exception, then the exception is not continued. Always execute `unwind_protect_end`, unless you want to catch all possible exceptions.

For example, a correct version of `while` that releases memory also in case of an error in the evaluation can be defined as follows:

```
oidtype whilebfn(bindtype args, bindtype env)
{
  oidtype cond = nil, v = nil;
  int arity = envarity(args), i;

  {unwind_protect_begin
    a_setf(cond, nthargval(args, 1));
    for(;;)
    {
      a_setf(v, evalfn(env, cond)); /* Evaluate condition */
      if(v == nil) /* Condition false => exit for loop */
        break;
      for(i=2; i<=arity; i++)
      {
        a_setf(v, evalfn(env, nthargval(args, i)));
      }
    }
  unwind_protect_catch;
  a_free(v); /* Release v and cond before exiting function */
  a_free(cond);
  unwind_protect_end;
  return nil; /* This statement not executed in case of an error */
}
}
```

10.2.2. Raising errors.

Every kind of error has an *error number* and an associated *error message*. There are predefined error numbers for common errors defined in `storage.h`. To raise an ALisp error condition use the system function:

```
oidtype lerror(int no, oidtype form, bindtype env);
```

`no` is the error number.

`form` is the failed expression.

`env` is the binding environment for the error.

For example, the following code implements the Lisp function `CAR`:

```
oidtype carfn(bindtype env, oidtype x)
```

```

{
  if(x==nil) return nil; // (CAR NIL) = NIL
  if(a_datatype(x) != LISTTYPE) return lerror(ARG_NOT_LIST,x,env);
  return hd(x);
}

```

A few convenience macros for common error checks are defined in `storage.h`:

<code>OfType(x, tpe, env)</code>	Raise a standard error if <code>x</code> is not of type <code>tpe</code> .
<code>IntoString(x, into, env)</code>	Dereference a symbol or string object <code>x</code> into a <i>pointer to</i> the C string in the image representing the object. Notice that the storage manager might invalidate this pointer as for other dereferences if the image moves.
<code>IntoString0(x, into, env)</code>	Dereference a string object <code>x</code> into C string.
<code>IntoInteger(x, into, env)</code>	Convert integer object <code>x</code> into C integer.
<code>IntoDouble(x, into, env)</code>	Convert real object <code>x</code> into C double.

To register a new error to the system use:

```
int a_register_error(char *msg);
```

`a_register_error` gets a unique error number for the error string `msg`. If `msg` has been registered before its previous error number is returned.

10.3. Calling Lisp from C

An ALisp function can be called from C by using the following C function:

```
oidtype call_lisp(oidtype lfn, bindtype env, int arity,
                 oidtype a1, oidtype a2,...)
```

`lfn` is the ALisp functional expression to call.
`env` is the error binding environment.
`arity` is the actual arity of the call.
`a1, a2, ...` are the actual arguments of the call.

For example, the following code implements an ALisp function (`MYMAP L FN`) which applies `FN` on each element in `L`:

```
oidtype mymapfn(bindtype env, oidtype l, oidtype fn)
{
  oidtype res = nil, lst=nil;

  {unwind_protect_begin;
    a_setf(lst,l);
    while(listp(lst))
    {
      a_setf(res,call_lisp(fn,env,1,hd(lst)));
      a_setf(lst,tl(lst));
    }
  unwind_protect_catch;
  a_free(res);
  a_free(lst);
}
```

```

    unwind_protect_end;
}
return nil;
}

```

Notice that unwind protection has to be used here to guarantee that the temporary memory locations are always released even if the call to `fn` causes an ALisp error.

Also notice that the called ALisp function might allocate new data objects and these have to be freed correctly.

Symbols are convenient for calling named ALisp functions from C. For example, the following function prints each element in a list:

```

oidtype mapprintfn(bindtype env, oidtype l)
{
    oidtype printsymbol = mk-symbol("print"), lst = nil;

    a_setf(lst,l);
    while(listp(lst))
    {
        call_lisp(printsymbol,env,l,hd(lst));
        a_setf(lst,tl(lst));
    }
    return nil;
}

```

Notice that symbols like `PRINT` are permanent and when a symbol is referenced from a location (here `printsymbol`) it need not be reference counted. Also the call to `PRINT` is here guaranteed to not generate any new objects and need not be released.

To call Lisp functions with variable arity use:

```

oidtype apply_lisp(oidtype fn, bindtype env, int arity, oidtype args[]);

```

The difference to `call_lisp` is that the arguments are passed in the array `args`. Don't forget to release the result.

To evaluate a C string of Lisp forms use:

```

oidtype eval_forms(bindtype env, char *forms);

```

All forms in `forms` are evaluated. The value of the last evaluation is returned as value. Don't forget to release the result.

10.3.1. *Direct C calls*

If the name of a C function implementing an ALisp function is known, it is more efficient to directly call the C function. However, arguments and results of direct C calls must be handled carefully to avoid storage leaks or system crashes. The automatic deallocation of temporary storage is NOT performed with direct C function calls. For example, the following correctly defined external Lisp function prints 'hello world' by directly calling the ALisp function `PRINT`:

```

oidtype hellofn(bindtype env)

```

```

{
  oidtype msg = nil;

  a_setf(msg, mkstring("hello world"));
  printfn(env, msg, nil); // PRINT has two arguments
  a_free(msg);
  return nil;
}

```

By contrast, the following incorrect implementation would cause a storage leak because the 'hello world' string is not deallocated:

```

oidtype hellofn(bindtype env)
{
  printfn(env, mkstring("Hello world"), nil);
  return nil;
}

```

Notice that `call_lisp` automatically garbage collects its arguments upon return; thus temporary objects among the arguments are automatically freed. For example, the following definition of `myhello` would be correct but slower than the previous definitions:

```

oidtype hellofn(bindtype env)
{
  call_lisp(mksymbol("print"), 2, env, mkstring("Hello world"), nil);
  return nil;
}

```

10.4. *C functions for debugging*

The reference counter of a physical storage object referenced by a handle is obtained with:

```
int refcnt(oidtype x)
```

Any ALisp object can be printed on the standard output with:

```
oidtype a_print(oidtype x);
```

When defining new physical storage type it is important to make sure that object allocation and deallocation works OK. Therefore there is a hook to the Amos II and Alisp top loops to trace how many objects are allocated, or deallocated, respectively. Turn on that hook by evaluating the form

```
(STORAGESTAT T)
```

The system will then make a report of how many objects have been (de)allocated for each physical storage type. Make sure that the same number of objects is deallocated as allocated if that is expected. Notice that object references might be saved in the database log and therefore you should rollback database updates when necessary to get the balance between allocated and deallocated objects.

Turn off storage usage tracing with:

```
(STORAGESTAT NIL)
```

In C memory leaks can be traced also by calling the system function:

```
void a_printstat(void)
```

It prints a report on how much storage was allocated since the previous time it was called.

Trapping memory corruption

When adding C-code to the system it is too easy to create corrupted database images. If not all the conventions for writing C-code is not followed by all modules errors typically occur in a completely different place of the system. When the system finds a corrupted memory location in the image it will print an error message:

```
Memory corruption in location 134000 (= 12345)
```

The two numbers *134000* and *12345* indicate that memory location denoted by handle (oidtype) *134000* is corrupt and points to a word containing the integer *12345*. To trap this when it actually happens can be done by calling the function

```
a_setdemon(oidtype loc, int val)
```

for example

```
a_set_demon(134000, 12345);
```

It causes the ALisp interpreter to continuously check if *loc* is equal to *val*. Whenever *loc* becomes equal to *val* the demon is turned off and an error is raised. See also Section 7.5.

10.5. Interrupt handling

The interrupt handling system is managed by the ALisp function (CATCHINTERRUPT). This function is called whenever an interrupt has occurred. It either prints a message or catches the interrupt. The following C macro checks if an error has occurred and calls CATCHINTERRUPT if that is the case:

```
CheckInterrupt;
```

An interrupt is indicated when the global C variable `InterruptHasOccurred` is set to `TRUE`. `CheckInterrupt` is called by the ALisp interpreter after every function call. If you write long-running C code you should insert calls to `CheckInterrupt` to allow interrupts to be managed.

References

- 1 Guy L.Steele Jr.: *Common LISP, the language*, Digital Press,
<http://www.cs.cmu.edu/Groups/AI/html/cltl/cltl2.html>
- 2 Staffan Flodin, Martin Hansson, Vanja Josifovski, Timour Katchaounov, Tore Risch, and Martin Sköld:
Amos II Release 7.x User's Manual,
http://www.it.uu.se/~udbl/amos/doc/amos_users_guide.html

Index

-	15	analyze code.....	51
!ARGS.....	46	AND.....	17
!VALUE.....	45	ANDIFY.....	11
*	16	APPEND.....	11
BACKTRACE-DEPTH.....	44	APPEND2.....	11
/	16	APPLY.....	23
:ERRCOND.....	41	apply function from C.....	70
:osql.....	5	apply_lisp.....	70
BATCH.....	49	APPLYARRAY.....	23
DEEP-PRINT.....	35	APROPOS.....	52
ERROR-CONDITION.....	41	arc cosine.....	16
EXCLUDE-PROFILE.....	48	arc sine.....	16
PROFILER-FREQUENCY.....	48	arc tangent.....	16
+	15	AREF.....	18
<	17	arguments of broken function.....	46
<=.....	17	array element.....	17
=	17	array dimensionality.....	17
>	17	ARRAYP.....	18
>=.....	17	ARRAYTOLIST.....	18
1-16		ARRAY-TOTAL-SIZE.....	18
1--.....	16	ASIN.....	16
1+.....	16	aspect-oriented programming.....	31
1++.....	16	ASSOC.....	11
a_datatype.....	56	association list.....	11
a_definetype.....	59	ASSQ.....	11
a_free.....	56	ATAN.....	16
a_let.....	57	atom.....	12
a_print.....	59, 71	ATOM.....	11
a_printstat.....	72	ATTACH.....	14
a_register_error.....	69	backquote.....	28
a_return.....	64	backtrace.....	43, 44
a_setf.....	56	BACKTRACE.....	49
a_stacksize.....	32	batch mode.....	43
absolute time values.....	33	binary tree.....	11
ACCEPT-SOCKET.....	38	binding environment.....	63
accessor functions.....	31	BOUNDP.....	9
ACOS.....	16	BQUOTE.....	29
ADDPROP.....	10	BREAK.....	45, 46, 49
ADJOIN.....	11	break commands.....	44
Adjustable arrays.....	17	break loop.....	5, 32, 43, 46
ADJUST-ARRAY.....	18	break point.....	43, 45
ADVISE-AROUND.....	31	break point on external Lisp function.....	46
AFTER-ROLLIN-FORMS.....	32	break point on function.....	45
AMOS-NAMESERVERHOST.....	40	break point on macro.....	46
AmosQL mode.....	5	broken function.....	43, 44, 46

B-trees	20	CommonLisp tutorial	5
BUILDL	11	COMPARE	17
BUILDN	11	CONCAT	15
BUTLAST	11	CONCATVECTOR	18
CAAAR	11	COND	25, 26
CAADR	11	conditional break points	46
CAAR	11	conditional tracing	47
CADAR	11	connection	38
CADDR	11	CONS	12
CADR	11	CONSP	12
call with variable arity	22	control structures	21, 28
call_lisp	69	copy with APPEND	11
CALLING	53	COPY-ARRAY	18
calling C from Lisp	62	COPY-TREE	12
calling Lisp from C	69	COS	16
CALLS	53	cosine	16
CAR	11	cross referencing functions, CALLING....	53
CASE	26	cross referencing functions, CALLS.....	53
CATCH	27, 28	cross-referencing	51
CATCHDEMON	42	CTRL-C	33, 38, 39, 40, 42
catcher	27	database image	4, 5, 54
CATCH-ERROR	41, 42	database image size	31
CATCHINTERRUPT	42	datatype	6, 32
CDAAR	11	datatype ADJARRAY	17, 55
CDADR	11	datatype ARRAY	17, 55
CDAR	11	datatype BINARY	55
CDDAR	11	datatype BTREE	20
CDDDDR	11	datatype CLOSURE	22, 55
CDDDR	12	datatype DATE	35
CDDR	12	datatype EXTFN	55
CDR	11, 12	datatype HASHTAB	19, 55
ceiling	16	datatype INTEGER	15, 55
CEILING	16	datatype LIST	11, 55
CHECKEQUAL	31	datatype REAL	15, 55
circular list	14	datatype SOCKET	35
cleanup form	41	datatype STREAM	35, 36, 55
CLEAR-FUNCTION-PROFILES	49	datatype STRING	14, 55
CLOCK	33	datatype SYMBOL	6, 55
close stream	35	datatype TEXTSTREAM	35, 37, 55
CLOSE-SOCKET	39	datatype TIME	34
CLOSESTREAM	35, 38	datatype TIMEVAL	34
closure	22, 23	date values	35
CLRHASH	19	DATE-DAY	35
code pattern	53	DATE-MONTH	35
code search	51, 53	DATEP	35
code verification	53	DATE-TO-TIMEVAL	34

DATE-YEAR.....	35	ERRCOND-NUMBER.....	41, 42
debug mode.....	40, 42, 43, 53	ERROR.....	41, 42
DEBUGGING.....	43, 49	error condition.....	41, 68
debugging C.....	71	error message.....	41, 68
debugging macros.....	29	error number.....	41, 68
DECLARE.....	31	error signal.....	41
DEFC.....	7, 8	ERROR?.....	41, 42
DEFGLOBAL.....	8, 9	escape character.....	14, 36
DEFMACRO.....	7, 30	EVAL.....	23
DEFSTRUCT.....	30	eval_forms.....	70
DEFUN.....	6, 7	EVALLOOP.....	31
DEFVAR.....	8, 9	evaluate C forms.....	70
DELETE.....	14	EVENP.....	17
DELETE-FILE.....	37	EVERY.....	25
dereferencing objects.....	55	exfunction.....	64
destructive CONS.....	14	EXIT.....	31
destructive list concatenation.....	14	EXP.....	16
destructive list element removal.....	14	explicit break point.....	45
destructive list manipulation.....	13	EXPLODE.....	10
destructive list merge.....	14	exponent.....	16
destructive reverse.....	14	EXPT.....	16
destructor.....	60	external function registration.....	63, 64
direct C call.....	70	external Lisp function.....	7, 62
DMERGE.....	14	EXTFNP.....	7
DO.....	27	extfunction.....	64
DO*.....	27	F/L.....	23
DOC.....	53	false.....	6, 16
documentation.....	51, 52	FAULTEVAL.....	41, 42
DOLIST.....	27	FIFTH.....	12
DOTIMES.....	27	file position.....	44, 51, 53
double precision.....	15	file streams.....	35, 36
DOUNITERRUPTED.....	42	FILE-EXISTS-P.....	37
dr, dereferencing objects.....	56	FILE-LENGTH.....	37
DUMPSTACK.....	44, 49	finding functions.....	51
dynamic argument list.....	22	finding source code.....	51
dynamic expressions.....	23	FIRST.....	12
dynamic scoping.....	8	FIRSTN.....	12
EIGHT.....	12	FLET.....	7
ELT.....	18	floating point numbers.....	15
Emacs.....	51	FLOOR.....	16
envarity.....	65	FLUSH.....	35, 39
environment variable RELEASING.....	51	focused frame.....	44
EQ.....	17	focused function.....	44
EQUAL.....	17	FORMAT.....	36
ERRCOND-ARG.....	41, 42	FORMATL.....	36
ERRCOND-MSG.....	41, 42	FOURTH.....	12

FP	51, 53	INTERSECTION	12
FRAMENO	42	INTERSECTIONL	12
free variables	21	IntoInteger	64
FUNCALL	21, 23	ISOME	24
FUNCTION	21, 23	iteration	23
function cell	6	keyword	10
function definition	6	KEYWORDP	10
function statistics	49	KEYWORD-TO-ATOM	10
function type	7	KWOTE	30
functional arguments	21	KWOTED	30
functions	70	LAMBDA	6
functions excluded from sampling	48	lambda expression	21, 24
garbage collection	4, 10, 50, 54, 65, 71	LAMBDA function	6
GENSYM	10	LAMBDA P	7
GET	10	LAST	12
GET-BTREE	20	LDIFF	12
GETD	7	LENGTH	12
GETF	12	lerror	68
GETHASH	19	LET	8, 9
GETHOSTNAME	38	LET*	9
GETTIMEOFDAY	34	lexical environment	45
global value	6, 59	Lisp function defined in C	7
global variable	9	Lisp macro	7
GO	9	list	11
GREP	53	LIST	12
handle assignment	56	LIST*	12
handle initialization	57	LISTP	12
handle release	56	LISTTOARRAY	18
handles	54	LOAD	37
HARDRESET	41, 42	LOC	50
hash table keys	19	local variables	8, 9
HASH-BUCKET-FIRSTVAL	19	LOCATEPOS	12
HASH-BUCKETS	20	LOG	16
HASH-TABLE-COUNT	20	logical objects	55
HELP	45, 50	LOOP	27
higher order functions	21	lower case	15
hooks	32	macro	7, 30
ID	31	macro expansion	29, 30
IF26		MACROEXPAND	30
image expansion	32, 50, 56	MACRO-FUNCTION	30
IMAGE-EXPANSION	50	macros	28
IMAGESIZE	31	MAKE_ARRAY	18
IN	12	MAKE_BTREE	20
indicator	11	MAKE-ARRAY	17, 18
INT-CHAR	15	MAKE-BTREE	20
INTEGERP	16	MAKE-HASH-TABLE	19, 20

MAKETEXTSTREAM	37	NTHCDR	12
malloc.....	54	NULL.....	12, 17
map function	23	NUMBERP	16
MAP-BTREE.....	21	numeric values	15
MAPC	24	objtags.....	55
MAPCAN	24	ODDP.....	17
MAPCAR.....	23, 24	OfType	64
MAPFILTER	24	oidtype, declaring handle	54
MAPHASH.....	19, 20	open stream.....	35
MAPL	25	OPEN-SOCKET	38, 39
MATCHING.....	53	OPENSTREAM.....	36, 37
MAX	16	OR.....	17
MEMBER	12	PACK.....	10
memory corruption.....	50, 72	PACKLIST	10
MEMQ.....	12	PAIR	13
MERGE.....	12	PAIRLIS	13
MIN.....	16	parameters.....	45
MINUS.....	16	peer.....	39
MKDATE	35	pending data.....	39
MKLIST.....	12	percentage spent in function	48
MKSTRING.....	6, 15	performance profiling	47
mksymbol.....	59	persistent data.....	54
MKSYMBOL	10	physical objects.....	54, 55
MKTIME	34	point-to-point communication	38
MKTIMEVAL	34	POLL-SOCKET.....	39
MOD	16	POP	13
MOVD	8	PP	37, 52
Move down the stack	45	PPF.....	37
name server	39	PPS.....	36
NATOM.....	12	pretty-print	29, 36
natural logarithm.....	16	Pretty-print	37
NCONC.....	14	PRIN1	36
NCONC1.....	14	PRINC.....	36
NEQ	17	PRINT	35, 36, 38
nick names	40	print function.....	60
NIL.....	6, 16	print name	6, 59
nil, C handle	54, 59	PRINTFRAME	50
NINTH	12	PRINT-FUNCTION-PROFILES.....	49, 50
NOBIND.....	10	PRINTL.....	37
non-blocking messages	40	PRINTSTAT	50
non-local returns	27	PROFILE	48, 50
NOT	17	PROFILE-FUNCTIONS.....	48, 50
NOTANY.....	25	PROG.....	9
NREVERSE.....	14	PROG1	25
NTH	12	PROG2.....	25
nthargval	65	PROG-LET	9

PROG-LET*	10	RPLACA	14
PROGN	25, 26	RPLACD	14
PROGNIFY	30	RPTQ	27
property indicator	10	RUN-SERVER	40
property list	6, 7, 10, 11, 12	samples	48
property value	10	sampling frequency	48
PUSH	13	scope	44
PUSH-VECTOR	18	search code	51
PUT	10	SECOND	13
PUT-BTREE	21	SELECTQ	27
PUTF	13	SEND-FORM	40
PUTHASH	20	sequences	17
QUIT	32	SET	10
QUOTE	10, 21, 23, 66	SETA	18
raising error	41	SETDEMON	50, 72
RANDOM	16	SET-DIFFERENCE	13
READ	35, 36, 38	SETF	14, 19, 20, 31, 32
READ-CHARCODE	36	SETFMETHOD	32
RECONS	13	SETQ	8, 10
recursive functions	23	SET-TIMER	33
REDIRECT-BASIC-STDOUTPUT	36	SEVENTH	13
REFCNT	50	side effects	29
reference counter	50, 55, 71	SIN	16
REGISTER-AMOS	40	sinus	16
REGISTER-INIT-FORM	32	SIXTH	13
REGISTER-SHUTDOWN-FORM	33	SLEEP	33
regression testing	31	SMASH	14
regular expression	15	socket stream	35
relative time values	33, 34	SOCKET-PORT	38
REMHASH	20	sockets	38
remote evaluation	38, 39	SOME	25
REMOTE-EVAL	40	SORT	13
REMOVE	13	sorting lists	13
remove break point	46	source code	51, 52
REMPROP	11	SPACES	36
RESET	41, 42	special forms	7, 66
reset Lisp	41, 44, 46	special variable	8, 9
reset point	41	SPECIAL-VARIABLE-P	10
RESETVAR	10	SQRT	16
REST	13	stack overflow	32, 42
RETURN	27	STACKSIZE	32
REVERSE	13	standard error	60
rewrite rule	28	standard input	35, 36, 60
rewrite rules	7	standard output	35, 36, 60
ROLLOUT	5, 32, 33, 54	START-PROFILE	47, 50
ROUND	16	statistical profiler	47

STOP-PROFILE	48, 50	THROW	27, 28, 41
storage leaks.....	50, 71	time functions.....	33
storage manager	35, 54, 67	time points.....	34
storage types.....	59	TIME-HOUR	34
storage usage.....	50	TIME-MINUTE.....	34
storage.h.....	54	TIMEP.....	34
STORAGESTAT	50, 71	TIMER	50
STORAGE-USED	50	timer function.....	33
streams	35	TIME-SECOND	34
string delimiter.....	36	TIMEVALP	34
STRING<.....	15	TIMEVAL-SEC.....	34
STRING=.....	15	TIMEVAL-TO-DATE.....	34
STRING-DOWNCASE	15	TIMEVAL-USEC	34
STRING-LIKE.....	15	TRACE	46, 50
STRINGP.....	15	TRACEALL.....	50
STRING-UPCASE	15	transform Lisp programs.....	28
structures.....	30	transient data.....	54
SUBLIS.....	13	TRAPDEALLOC.....	50
SUBPAIR.....	13	true	6, 16
SUBSET.....	25	truth value	6
SUBSETP	13	type identifier	54, 55
SUBST	13	type name	6, 55
surrogate objects	55	type reader.....	37
SWAP	18	type table.....	59
SYMBOL-FUNCTION	6, 8	type tag.....	6
SYMBOLP.....	11	TYPENAME.....	6, 32
SYMBOL-PLIST.....	11	TYPE-READER	37
symbols	6	UNBREAK	46, 50
SYMBOL-SETFUNCTION	7, 8	undeclared global variables.....	8
SYMBOL-VALUE.....	10	undefined functions.....	53
syntactic sugar.....	28	undefined variables	53
t, C handle	59	UNFUNCTION.....	30
T, global Lisp variable	16	UNION.....	13
T<.....	34	UNIONL	13
T<=.....	34	UNIQUE	13
T>.....	34	UNLESS	27
T>=.....	34	UNPROFILE-FUNCTIONS.....	49, 50
TAN	16	UNTRACE.....	47
tangent.....	16	unwind protection in C.....	67, 70
TCP/IP.....	38	UNWIND-PROTECT.....	28, 41, 43
TENTH	13	upper case.....	15
TERPRI.....	36	USING	53
text streams	37	VAG.....	51
TEXTSTREAMPOS.....	37	value of handle.....	54
TEXTSTREAMSTRING.....	38	variable.....	8
THIRD	13	variable arity	22

variable arity external Lisp functions ...	7, 65	WHILE.....	27
variable number of arguments	7, 65	WITH-INPUT-FILE	37
VECTOR.....	18	WITH-OUTPUT-FILE	37
VERIFY-ALL.....	53	wrapping profiler	47, 48
VIRGINFN	51	XEmacs.....	51
WHEN.....	27		