ALisp User’s Guide

Tore Risch
Uppsala Database Laboratory
Department of Information Technology
Uppsala University
Sweden
Tore.Risch@it.uu.se

2000-07-09
(Latest revision 2006-01-11)

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 in scalable end 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 ANSII 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.

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 ANSII 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. Since 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, since 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 to Amos II [2]. Amos II is started with the OS command:

```
amos2
```

There has to be a database file with a system Amos II database in the file:

```
amos2.dmp
```

in the same directory as where Amos II is started.

When Amos II is started it accepts AMOSQL [2] commands from the console. To enter the ALisp read-eval-print top loop from the AMOSQL top loop, give the AMOSQL command:

```
lisp;
```

To go back to AMOSQL from ALisp mode, enter the keyword:

```
:osql
```

to the ALisp top loop.

All data to be stored in an Amos II database is stored inside a dynamic main memory area called the *image*. When the database has been populated 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. 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 name. The symbolic name of the data type of an ALisp object O can be accessed with the ALisp function:

```
(TYPENAME 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 (Sec. 9.7 Storage types). The system tightly interoperates with C/C++ so that data structures can be shared between C and ALisp, the ALisp garbage collector is available from C, C and ALisp functions can call each other, etc.

3.1 Symbols

A *symbol* (data type name SYMBOL) is a *unique* and capitalized text string with which various system data can be associated. Symbols are used for representing *Lisp functions*, *Lisp macros*, *Lisp variables* and *property lists*. Lisp functions and macros represent executable ALisp code (Sec. 4. System Functions and Variables), Lisp variables bind symbols to values, and property lists associate data values with symbols and properties. Symbols are unique and therefore the system maintains a hash table of all symbols. Symbols are **not** garbage collected and their location in the 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, objects, 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:

1. The *print name* is the string representing the symbol. The print name of the symbol S can be accessed by the function:
   ```
   (MKSTRING S)
   ```
2. The *function definition* is non-nil when the symbol has an associated function definition. Functions can be system defined or defined in ALisp by the built-in functions DEFUN or DEFMACRO. The function definition of a symbol

S is accessed with (SYMBOL-FUNCTION S) and updated with (SYMBOL-SETFUNCTION S V). A function definition can be of various types as explained in Sec. 3.3 Functions.

3. The *global value* of a variable is bound by the Lisp functions SETQ or SET (Sec. 4.1 Control Structures and Variable Binding).

   Notice that, unlike most programming languages, global Lisp variables can 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 Alisp such dynamically scoped variables are declared using DEFVAR (Sec. 4.9 Function and Variable Definitions). It is also possible to declare not-scoped global variables (usually constants) with DEFGLOBAL. There are a number of built-in global variables that stores various system information and system objects.

4. The *property list* associates *property values* with the symbol and other symbols called *properties*. The property list of a symbol S is accessed with (SYMBOL-PLIST S). A property list is represented as a list with an even number of elements where every second element are property symbols 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, since atoms are not garbage collected, dynamic databases should not be represented with property lists. The Lisp functions PUTPROP and GETPROP are used for manipulating property lists (Sec. 4.3 Strings and Symbols).

3.2 Lists

Lists (data type name LIST) are objects having two fields, CAR and CDR (alt. FIRST and REST), pointing to other Lisp objects. CAR points to the head of the list and CDR points to its tail. There are many system functions for manipulating lists (see Sec. 3.2 Lists). Programs in Lisp are defined as lambda expressions using mainly lists and symbols.

3.3 Functions

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 function (SYMBOL-FUNCTION *nm*) (Sec. 4.9 Function and Variable Definitions) that returns the function definition of *nm* if there is one; otherwise it returns **nil** (The function GETD is equivalent to SYMBOL-FUNCTION). A function definition can be one of the following:

A. A ***LAMBDA*** *function* which is an interpreted function. It is defined by the system function DEFUN. A lambda function definition is represented as a list, (LAMBDA args . body).

B. A *Lisp* ***MACRO*** is defined by the system function DEFMACRO. A MACRO is a Lisp function that takes as its argument a functional expression (called *form* or *S-expression*) and produces a new equivalent functional expression (form). Many system functions are defined as macros. They are Lisp's *rewrite rules*. MACROs are made very efficient in ALisp since the first time the interpreter encounters a MACRO call it will modify the code and replace the original form with the macro-expanded one. 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 Lisp function (SYMBOL-MACRO *nm*) (Sec. 4.9 Function and Variable Definitions) returns the function definition of the MACRO *nm* if it is a MACRO; otherwise it returns NIL.

C. A ***SUBR*** *function* is a Lisp function defined in C. A SUBR is represented by special datatype named EXTFN and printed as #[SUBRn fn], where n is the arity of the function and fn is its name. E.g. the definition of CONS is printed as #[SUBR2 CONS]. The EXTFN data structure contains a pointer to the C definition.

D. A ***No-Spread*** *function* is a SUBR with varying number of arguments. It definition is printed as #[SUBR-1 fn], e.g. #[SUBR-1 LIST].

E. A ***Special Form*** is a SUBR with varying number of arguments and where the arguments are not evaluated the standard way. Special Forms are printed as #[SUBR-2 fn], e.g. #[SUBR-2 COND]

3.4 Numbers

Numbers represent numeric values. Numeric values can either be integers (data type name INTEGER) or floating point numbers (data type name REAL). Integers are entered to the ALisp 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

There are several arithmetic system functions (see Sec. 4.5 Arithmetic Functions).

3.5 Strings

Strings (data type name STRING) represent text strings of arbitrary length. The Lisp print functions PRIN1 and PRINT print strings enclosed within *string delimiter* (") characters. Strings containing the characters '"' or '\' must precede these with the escape character '\'. The ALisp reader (function READ) recognizes strings. E.g.

"This is a string" "This is a string containing the string delimiter \""

The Lisp function PRINC strips string delimiters and escape characters from a printed string.

Functions manipulating strings are described in Sec. 4.3 Strings and Symbols.

3.6 Arrays

Arrays (data type name ARRAY) represent 1 dimensional sequences. The elements of an array can be of any type.

Arrays are allocated with
(make-array size)

Array elements are accessible through
(aref array pos)

Array elements are updated through
(setf (aref array pos) value)

The size of an array is obtained by
(array-total-size array) or
(length array)

The record package of ALisp (see DEFSTRUCT in Sec. 4.10 Defining Structures) is implemented using arrays.

Adjustable arrays are arrays that can be dynamically increased in size. They are allocated with
(make-array size :adjustable t)

Arrays can be enlarged with
(adjustarray 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.

The system functions for manipulating arrays are described in Sec. 4.6 Array Functions.

3.7 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.

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))

Elements of a hash table are accessed with
(gethash key hashtab)

An element of a hash table is updated with
(setf (gethash key hashtab) value)

An element of a hash table is removed with
(remhash key hashtab)
or
(setf (gethash key hashtab) nil)

Iteration over all elements in a hash table is made with
(maphash (function(lambda (key val) ...)) hashtab)

The ALisp functions to manipulate hash tables are listed in Sec. 4.7 Hash Tables.

3.8 Main memory B-trees

Main memory B-trees (data type name 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-tree 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.

B-trees are allocated with

(make-btree)

Elements of a B-tree are accessed with
(get-btree key btree)

An element of a B-tree is updated with
(put-btree key btree value)

Iteration over the values in a B-tree whose keys are in a given interval is made with
(map-btree btree lower upper fn)

The key-value pairs of a B-tree are printed on the standard output with
(print-btree btree)

The B-tree manipulating functions are described in Sec. 4.8 B-Trees.

3.9 Streams

Streams (data type name STREAM) are used for reading and writing files.

A new stream is opened with
(openstream filename mode)
where mode can be "r" for reading, "w" for writing, or "a" for appending. The usual Lisp I/O functions (see Sec. 5. Input and Output) work on streams.

The stream is closed with
(closestream stream)

The standard output for PRINT etc. can be dynamically redirected to a stream by
(stdoutstream stream)
After stdoutstream is called all system messages and all calls to Lisp output functions to the standard output are redirected to the stream. stdoutstream returns the old standard output stream. If stdoustream is called without any arguments the current standard output stream is returned without redirection.

The standard input can be dynamically redirected to read from a stream by
(stdinstream stream)
After stdinstream is called the Lisp toploop will read from the specified stream. The function returns the old standard input stream. If stdinstream is called without any arguments the current standard input stream is returned without redirection.

There are several kinds of streams for reading and writing from files, string buffers, and sockets (See Sec. 5.2 Text streams).

The storage manager allows the user to define new kinds of streams.

4. System Functions and Variables

This section describes the built-in ALisp system functions, macros and variables. Those functions that are similar or equivalent to standard CommonLisp functions are marked with a ’*’ under *Type*. The reader should read e.g. [1] for further details on CommonLisp functions. When there are significant differences between an ALisp function and the corresponding CommonLisp function these differences are described. The *Type* of a function can be SUBR (defined in C), SPECIAL (special form), LAMBDA (interpreted), or MACRO (Lisp macro). A system variable can be either SPECVAR or GLOBAL. In the descriptions of arguments of functions X... indicates that the expression X can be repeated an arbitrary number of times.

The rest of this section contains short descriptions of the available built-in functions, ordered by subject. Notice, however, that I/O functions are described in Sec. 5. Input and Output, error management functions in Sec. 6. Error handling, and debugging functions in Sec. 7. Lisp Debugging.

4.1 Control Structures and Variable Binding

This subsection describes system functions, macros, and special forms for handling ALisp’s control structures, variable binding, and function application.

Function	Type	Description
(APPLY FN ARGS)	*SUBR	Apply the function FN on the arguments in the list ARGS.
(APPLYARRAY FN A)	SUBR	Apply the Lisp function FN on the arguments in the array A.
(BOUNDP VAR)	*SUBR	Return T if the variable VAR is bound. Unlike CommonLisp BOUNDP works not only for special and global variables but also for local variables.
(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.
(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.
(CATCH TAG FORM)	*SPECIAL	Evaluate 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.
(COND (TEST FORM...)...)	*SPECIAL	Classical Lisp conditional execution of forms.
(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.
(EVAL FORM)	*SUBR	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 #'(LAMBDA (X) FORM...).
(FLET ((FN DEF)...) FORM...)	*MACRO	Bind local function definitions and evaluate the forms FORM...
(FUNCALL FN ARG1...)	*SUBR	Call function FN with arguments ARG1...
(FUNCTION FN)	*SPECIAL	Make a closure of the function FN. Do not use QUOTE!
(IF P A B)	*SPECIAL	If P then evaluate A else evaluate B.
(LET ((VAR INIT...)...) FORM...)	*MACRO	Bind local variables in parallel and evaluate the forms FORM...
(LET* ((VAR INIT...)...) FORM...)	*MACRO	Bind local variables in sequence and evaluate the forms FORM....
(LOOP FORM...)	*MACRO	Evaluate the forms FORM... repeatedly. The loop can be terminated, and the result VAL returned, by calling (RETURN VAL).
(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)	As PROG-LET but binds the local variables as LET*.
(PROG1 X...)	*SUBR	Return the value of its first form among the arguments X...
(PROGN X...)	*SUBR	Return the value of the last form among the arguments X...
(QUOTE X)	*SPECIAL	Return X unevaluated.
(RESETVAR VAR VAL FORM...)	MACRO	Temporarily re-bind 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; the main difference being that look up of special variables in the current ALisp is rather expensive, while global variables are cheap to look up.
(RETURN VAL)	*LAMBDA	Return value VAL form the block in which RETURN is called. A block can be a PROG-LET, PROG-LET*, DOLIST, DOTIMES, DO, DO*, LOOP or WHILE expression.
(RPTQ N FORM)	SPECIAL	Evaluate FORM N times. Recommended for timing in combination with TIMER.
(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))
(SET VAR VAL)	*SUBR	Bind the value of the variable VAR to VAL.
(SETQ VAR VAL)	*SPECIAL	Change the value of the unevaluated variable VAR with VAL.
(THROW TAG VAL)	*SUBR	Return VAL as the value of a call to CATCH with the *catcher* TAG that has called THROW directly or indirectly.

(WHILE TEST FORM...)	MACRO	Evaluate the forms FORM... while TEST is true or until RETURN is called.

4.2 List Functions

Function	Type	Description
(ADJOIN X L)	*SUBR	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)	SUBR	Append two lists X and Y.
(ASSOC X ALI)	*SUBR	Search association list ALI for a pair (X .Y). Tests with EQUAL.
(ASSQ X ALI)	SUBR	Similar to ASSOC but tests with EQ.
(ATOM X)	*SUBR	True if X is not a list or if it is NIL.
(ATTACH X L)	SUBR	Similar to (CONS X L) but *destructive*, i.e. the head of the list L is modified so that all pointers to L will point to the extended list after the attachment. This does *not* work if L is not a list, in which case ATTACH works like CONS.
(BUILDL L X)	LAMBDA	Build a list of same length as L whose elements are all the same X.
(BUILDN X N)	LAMBDA	Build a list of length N whose elements all are all the same X:
(BUTLAST L)	*SUBR	Return a copy of list L minus its last element.
(CAAAR X)	*SUBR	(CAR (CAR (CAR X)))
(CAADR X)	*SUBR	(CAR (CAR (CDR X)))
(CAAR X)	*SUBR	(CAR (CAR X))
(CADAR X)	*SUBR	(CAR (CDR (CAR X)))
(CADDR X)	*SUBR	(CAR (CDR (CDR X))), same as (THIRD X)
(CADR X)	*SUBR	(CAR (CDR X)), same as (SECOND X)
(CAR X)	*SUBR	Return the head of the list X, same as (FIRST X).
(CDAAR X)	*SUBR	(CDR (CAR (CAR X)))
(CDADR X)	*SUBR	(CDR (CAR (CDR X)))
(CDAR X)	*SUBR	(CDR (CAR X))
(CDDAR X)	*SUBR	(CDR (CDR (CAR X)))
(CDDDDR X)	*LAMBDA	(CDR (CDR (CDR (CDR X))))
(CDDDR X)	*SUBR	(CDR (CDR (CDR X)))
(CDDR X)	*SUBR	(CDR (CDR X))
(CDR X)	*SUBR	Return the tail of the list X, same as (REST X).
(CONS X Y)	*SUBR	Construct new list cell.
(CONSP X)	*SUBR	Test if X is a list cell.
(COPY-TREE L)	*SUBR	Make a copy of all levels in list structure. To copy the top level only, use (APPEND L).
(DELETE X L)	*SUBR	Remove *destructively* 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.
(DMERGE X Y FN)	LAMBDA	Merge lists X and Y *destructively* with FN as comparison function. E.g. (DMERGE ’(1 3 5) ’(2 4 6) #’<) => (1 2 3 4 5 6)

		The value is the merged list; the merged lists are destroyed.
(EIGHT L)	*LAMBDA	Get the eight element from list L.
(FIFTH L)	*LAMBDA	Get fifth element in list L.
(FIRST L)	*SUBR	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)	*SUBR	Get value stored under the indicator I in the property list L.
(ILENGTH L)	LAMBDA	Compute the length of the list L. Returns 0 if L is atom.
(IN X L)	SUBR	IN returns T if there is some substructure in L that is EQ to X.
(INTERSECTION X Y)	*SUBR	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.
(ISOME L FN)	SUBR	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. E.g. (ISOME ’(1 2 2 3) #’(LAMBDA (X TL)(EQ X (CADR TL)))) => (2 2 3)
(KWOTE X)	SUBR	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)	SUBR	Return T if X is a quoted form. For example: (KWOTED 1) => T (KWOTED ’(QUOTE (1))) => T (KWOTED ’(1)) => NIL
(LAMBDAP X)	LAMBDA	Return T if X is a lambda expression.
(LAST L)	*SUBR	Return the last tail of the list L. E.g. (LAST ’(1 2 3)) => (3)
(LENGTH X)	*SUBR	Compute the number of elements in the list X, or the number of characters in the string X. See also ILENGTH.
(LIST X...)	*SUBR	Make a list with the elements X...
(LIST* X...)	*SUBR	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)	*SUBR	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
(MAPC FN ARGS...)	*MACRO	Apply FN on each of the elements of the lists ARGS...
(MAPCAN FN ARGS...)	*MACRO	Apply FN on each of the elements of the lists ARGS... and NCONC together the results.
(MAPCAR FN &REST ARGS)	*MACRO	Apply FN on each of the elements of the lists ARGS... and build a list of the results.
(MAPFILTER FILT LST &OPTIONAL OP)	SUBR	If OP=NIL return the subset of the elements for which the filter function FILT 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/l (x)(1+ x))) => (2 3) See also SUBSET!
(MAPL FN ARGS...)	*MACRO	Apply FN on each tail of the lists ARGS...
(MEMBER X L)	*SUBR	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)	SUBR	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. E.g. (MERGE '(1 3) '(2 4) (function <)) => (1 2 3 4)
(MKLIST X)	SUBR	X is returned if it is NIL or a list. Otherwise (LIST X) is returned.
(NCONC L...)	*MACRO	*Destructively* concatenate the lists L... and return the concatenated list.
(NCONC1 L X)	SUBR	Add X to the end of L *destructively*, i.e. same as (NCONC L (LIST X))
(NINTH L)	*LAMBDA	Get ninth element in list L.
(NREVERSE L)	*SUBR	*Destructively* reverse the list L. The value is the reversed list. L will be destroyed.
(NTH N L)	*SUBR	Get the Nth element of the list L with enumeration starting at 0.
(NTHCDR N L)	*SUBR	Get the Nth tail of the list L with enumeration starting at 0.
(NULL X)	*SUBR	True if X is NIL.
(PAIR X Y)	SUBR	Same as PAIRLIS.
(PAIRLIS X Y)	*SUBR	Form an association list by pairing the elements of the lists X and Y.
(POP L)	*SPECIAL	Same as (SETQ L (CDR L)).
(PROGNIFY FORML)	LAMBDA	Make a single form from a list of forms FORML.
(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)	SUBR	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)	*SUBR	Remove all occurrences of X from the list L. Tests with EQUAL.
(REST L)	*LAMBDA	Same as CDR.
(REVERSE L)	*SUBR	Return a new list whose elements are the reverse of the top level of L.
(RPLACA L X)	*SUBR	*Destructively* replace the head of list L with X.
(RPLACD L X)	*SUBR	*Destructively* replace the tail of list L with X.
(SECOND L)	*SUBR	Get second element in list L. Same as CADR.
(SET-DIFFERENCE X Y)	*SUBR	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.
(SMASH X Y)	LAMBDA	Replace *destructively* the list X with the list Y.
(SORT L FN)	*LAMBDA	Sort the elements in the list L using FN as comparison function.
(SUBLIS ALI L)	*SUBR	Substitute elements in the list structure L as specified by the association list ALI that has the format ((FROM . TO)...). E.g. (SUBLIS '((A . 1)(B . 2)) '((A) B A)) => ((1) 2 1)
(SUBPAIR FROM TO L)	SUBR	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. E.g. (SUBPAIR '(A B) '(1 2) '((A) B A)) => ((1) 2 1)
(SUBSET L FN)	LAMBDA	Return the subset of the list L for which the function FN returns true.
(SUBSETP X Y)	*LAMBDA	Return true if every element in the list X also occurs in the list Y.
(SUBST TO FROM L)	*SUBR	Substitute FROM with TO in the list structure L. Tests with EQUAL. E.g. (SUBST '1 'A '((A) B A)) => ((1) B 1)
(SWAP A E1 E2)	LAMBDA	Swap elements E1 and E2 in the array A.
(TENTH L)	*LAMBDA	Get the tenth element in the list L.
(THIRD L)	*SUBR	Get the third element of the list L. Same as CADDR.
(UNION X Y)	*SUBR	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)	SUBR	Remove all duplicate elements in the list L. Tests with EQUAL.

4.3 Strings and Symbols

Function	Type	Description
(ADDPROP S I V FLG)	SUBR	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.
(CONCAT STR...)	SUBR	Coerce the arguments STR... to strings and concatenate them.
(EXPLODE S)	SUBR	Unpack a symbol S into a list of single character symbols. Symbols are exploded into symbols and strings into strings. E.g. (EXPLODE 'ABC) => (A B C) (EXPLODE "abc") => ("a" "b" "c"))
(GENSYM)	*LAMBDA	Generate new symbols named G:1, G:2, etc.
(GET S I)	*SUBR	Get the property of symbol S having the indicator I.
(GETPROP S I)	SUBR	Same as GET.
(INT-CHAR X)	*SUBR	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 the character is returned.
(KEYWORDP X)	*SUBR	Return T if X is a keyword (i.e. symbol starting with ':').
(KEYWORD-TO-ATOM X)	SUBR	Convert a keyword X into a regular symbol without the ':'.
(LITATOM X)	SUBR	Return true if X is a symbol. Same as (SYMBOLP X).
(MKSTRING X)	SUBR	Coerce an atom to a string.
(MKSYMBOL X)	SUBR	Coerce a string to a symbol. The characters of X will be capitalized.
(NATOM X)	*SUBR	Return T if X is not an atom and not NIL.
(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)	*SUBR	Set the value stored on the property list of the symbol S under the indicator I to V.
(PUTPROP S I V)	SUBR	Same as PUT.
(REMPROP S I)	*SUBR	Remove property stored for the indicator I in the property list of symbol S.
(STRING-UPCASE STR)	*SUBR	Change all ASCII characters in the string STR to upper case.
(STRING< S1 S2)	*SUBR	Return true if the string S1 alphabetically precedes S2.
(STRING= S1 S2)	*SUBR	Return true if the strings S1 and S2 are the same. EQUAL works too.
(STRINGP X)	*SUBR	Return true if X is a string.
(SYMBOL-FUNCTION S)	*SUBR	Get the function definition associated with the symbol S. Same as GETD.
(SYMBOL-PLIST S)	*SUBR	Get the entire property list of the symbol S.
(SYMBOL-SETFUNCTION S D)	SUBR	Set the function definition of symbol S to D. Same as DEFC.
(SYMBOL-VALUE S)	*SUBR	Get the global value of the symbol S. Returns the symbol NOBIND if no global value is assigned.
(SYMBOLP X)	*SUBR	Return true if X is a symbol.

4.4 Logical Functions

Function	Type	Description
(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)	SUBR	Compare order of two objects. Return 0 if they are equal, -1 if less, and 1 if greater.
(EQ X Y)	*SUBR	Test if X and Y have the same address.
(EQUAL X Y)	*SUBR	Test if objects X and Y are equivalent [1]. **Notice** that, in difference to CommonLisp, non- arrays are equal if all their elements are equal.
(NEQ X Y)	*SUBR	(NOT (EQ X Y))
(OR X...)	*SPECIAL	Evaluate the forms X... until some form does not evaluate to NIL. Return the value of that form.
(NOT X)	*SUBR	True if X is NIL; same as NULL.
(EVERY FN ARG...)	*MACRO	Return T if FN returns non-nil result when applied on every element in the lists ARG... in parallel.
(SOME FN L...)	*MACRO	Apply FN on each of the elements in the lists L... in parallel. If FN returns non-nil value for some element then SOME will return T, otherwise NIL.

4.5 Arithmetic Functions

Function	Type	Description
(+ X...)	*SUBR	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...)	*SUBR	Multiply the numbers X...
(/ X Y)	*SUBR	Divide X with Y.
(< X Y)	*SUBR	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)	*SUBR	Tests if two numbers are the same. E.g. (= 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
(INTEGERP X)	*SUBR	True if X is an integer.
(MAX X Y)	*SUBR	Return the largest of the numbers X and Y. Exactly two arguments, unlike CommonLisp standard.
(MIN X Y)	*SUBR	Return the smallest of the numbers X and Y. Exactly two arguments, unlike CommonLisp standard.
(MINUS X)	*SUBR	Negate the number X.
(MOD X Y)	*SUBR	Return the remainder when dividing X with Y. X and Y can be integers or floating point numbers.

(NUMBERP X)	*SUBR	True if X is number.
(RANDOM N)	*SUBR	Generate a random integer between 0 and N.
(SQRT X)	*SUBR	Compute the square root of the number X.

4.6 Array Functions

Function	Type	Description
(ADJUST-ARRAY A NEWSIZE)	*SUBR	Increase the size of the adjustable array A to NEWSIZE. Only one-dimensional adjustable arrays are supported. 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. Unlike CommonLisp only one dimensional arrays are supported. See also SETF.
(ARRAY-TOTAL-SIZE A)	*SUBR	Return the number of elements in the (one-dimensional) array A.
(ARRAYP X)	*SUBR	True if X is an array (fixed or adjustable).
(ARRAYTOLIST A)	SUBR	Convert an array A to a list.
(CONCATVECTOR X Y)	LAMBDA	Concatenate arrays X and Y.
(COPY-ARRAY A)	*SUBR	Make a copy of the non-adjustable array A.
(ELT A I)	SUBR	Retrieve element I from array A. Enumeration starts at 0. Same as (AREF A I).
(LISTTOARRAY L)	SUBR	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)	SUBR	Adjusts the array A with one element X at the end.
(SETA A I V)	SUBR	Set the element I in the array A to V. Returns V.
(VECTOR X...)	*SUBR	Make an array of X..

4.7 Hash Tables

Function	Type	Description
(CLRHASH HT)	SUBR	Clear all entries from hash-table HT and return the empty table.
(GETHASH K HT)	*SUBR	Get value of element with key K in hash table HT.
(HASH-BUCKET-FIRSTVAL HT)	SUBR	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)	SUBR	Compute the number of buckets in the hash table HT.
(HASH-TABLE-COUNT HT)	*SUBR	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 parameter :SIZE is ignored since 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)	*SUBR	Apply (FN KEY VAL V) on each key and value of the hash table HT.

(PUTHASH K HT V)	SUBR	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)	SUBR	Remove the value stored in the hash table HT under the key K.

4.8 B-Trees

Function	Type	Description
(GET-BTREE K BT)	SUBR	Get value of element with key K in B-tree BT. Comparison uses COMPARE.
(MAKE-BTREE)	SUBR	Allocate a new B-tree.
(MAP-BTREE BT LOWER UPPER FN)	SUBR	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.
(PRINT-BTREE BT)	SUBR	Print all key-value pairs in B-tree BT.
(PUT-BTREE K BT V)	SUBR	Set the value stored in the B-tree BT under the key K to 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.

4.9 Function and Variable Definitions

The following system functions are used for defining and accessing function, macro, and variable definitions.

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 a SUBR the variable !ARGS is bound in CODE to a list of the actual arguments.
(DECLARE ...)	*MACRO	Dummy defined in ALisp for compatibility with CommonLisp.
(DEFC FN DEF)	SUBR	Associate the function definition DEF with the atom FN. Same as (SYMBOL-SETFUNCTION FN DEF).
(DEFGLOBAL VAR &OPTIONAL VAL)	MACRO	Declare VAR to be a *global variable* with optional initial value VAL. Global variables cannot be rebound locally with LET/LET*. They are much faster to look up than dynamically scoped variables (see DEFVAR).
(DEFMACRO NAME ARGS FORM...	*SPECIAL	Define a new MACRO.
(DEFVAR VAR &OPTIONAL VAL)	*SPECIAL	Declare VAR to be a *special variable* with optional initial value VAL. Special variables are dynamically scoped non-local variables [1]. See also DEFGLOBAL.
(DEFUN FN ARGS FORM...)	*SPECIAL	Define a new Lisp function.
(EXTFNP X)	LAMBDA	Return T if X is a function defined in C.
(GETD X)	SUBR	Get function definition of atom NM. Same as SYMBOL-FUNCTION

(MACRO-FUNCTION FN)	*SUBR	Return the function definition of FN if FN is a macro; otherwise return NIL.
(MACROEXPAND FORM)	*SUBR	If FORM is a macro return what it rewrites FORM into; otherwise FORM is returned unchanged.
(MOVD F1 F2)	SUBR	Make F2 have the same function or macro definition as F1.
(SPECIAL-VARIABLE-P V)	*SUBR	Return T if the variable V is declared as special with DEFVAR.

4.10 Defining Structures

ALisp includes a subset of the structure definition package of CommonLisp. The structure package is 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.

Function	Type	Description
(DEFSTRUCT NAME SLOT...)	*MACRO	Define a new record type NAME with fields named SLOT...The fields can only be pointers. DEFSTRUCT generates a number of macros and functions to create and update instances of the record type: New instances are created with (MAKE-NAME :SLOT VALUE ...) The fields of a record R are accessed using functions (NAME-SLOT R) To update a slot of record R with a new value V use (SETF (NAME-SLOT R) V) To test if an object O is a record named NAME, use (NAME-P O)
(SETF PLACE VAL)	*MACRO	Update the location PLACE to become VAL. The location can be specified with one of the functions AREF, GETL or GETHASH or access functions for user defined structures using DEFSTRUCT. A new *SETF macro* 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 form should return the update form with which to replace (SETF (FOO ...) V).

4.11 Miscellaneous Functions

Function	Type	Description
(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.
(CLOCK)	SUBR	Compute the number of wall clock seconds spent so far during the run as a floating point number.

(EVALLOOP)	SUBR	Enter a Lisp top loop. Exit when function (EXIT) is called.
(EXIT)	SUBR	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)	SUBR	Extend the system's data 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.
(IN-SYSTEM SYS FORM...)	MACRO	If the global variable _SYSTEM_ is EQ to atom SYS or member of list SYS then the forms FORM... are evaluated.
(QUIT)	*SUBR	Quit Lisp. If ALisp is embedded in another system it will terminate as well.
(ROLLOUT FILE)	SUBR	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.
(SLEEP N)	*SUBR	Suspend the execution for N seconds.
(TYPENAME X)	SUBR	Get the name of the datatype of object X.
(UNFUNCTION FORM)	LAMBDA	Get the function definition of a functional expression.

5. Input and Output

The I/O system of ALisp is stream oriented, but much simpler than the elaborate I/O system of CommonLisp. There are several kinds of streams: i) for terminal/file I/O, ii) *text streams* for reading and writing into text buffers, and iii) socket streams for communicating with other Amos II systems. The same I/O functions normally works for both regular I/O and text streams. The storage manager allows the programmer to define new kinds of stream.

5.1 File I/O

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

Function	Type	Description
(CLOSESTREAM STR)	SUBR	Close the I/O stream STR.
DEEP-PRINT	GLOBAL	(Default T). Normally the contents of fixed size arrays and structures are printed by PRINT etc. This allows I/O of such data types. However, when _DEEP-PRINT_==NIL the contents of arrays and structures are *not* printed. Good when debugging large or circular structures.
(DELETE-FILE FILE)	*SUBR	Delete the file named FILE. Returns T if successful.
(FILE-EXISTS-P NM)	*SUBR	Return T if file named NM exists.
(FILE-LENGTH NM)	*SUBR	Return the number of bytes in the file named NM.
(FORMATL FILE FORM...)	MACRO	Simple replacement of some of the functionality of FORMAT in CommonLisp. Print the values of the forms FORM... on file FILE. A marker T among FORM... indicates a line feed while the string "~PP" makes the next element pretty-printed.

		E.g. (FORMATL T "One: " 1 T) prints One: 1
(LOAD FILE)	*SUBR	Evaluate the forms in the file named FILE.
(OPENSTREAM FILE MODE)	SUBR	Open a stream against external file. MODE is the *file mode* i.e. "r", "w", or "rw". Since 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..., e.g. (PP PPS PPF). **Notice** that function symbols are not quoted.
(PPF L FILE)	LAMBDA	Pretty-print the functions and variables in L into the specified file, e.g. (PPF '(PPS PPF) "pps.lsp")
(PPS S &optional STREAM)	LAMBDA	Pretty-print expression S.
(PRIN1 S STREAM)	*SUBR	Print the object S in the stream STREAM with escape characters and string delimiters inserted so that the object can be read with (READ STREAM) later to produce a form EQUAL to S.
(PRINC X STREAM)	*SUBR	Print the object into the stream STREAM without escape characters and string delimiters.
(PRINT X STR)	*SUBR	(PRIN1 X STR) followed by a line feed.
(PRINTL X...)	LAMBDA	Print the objects X... as a list.
(READ &OPTIONAL STR)	*SUBR	Read expression from stream STR. If STR is T or NIL it reads from primary input. IF X is a string, the system reads an expression from the string, e.g. (READ "(A B C)") => (A B C)
(READ-CHARCODE STREAM)	SUBR	Read one byte from the stream STREAM and return it as an integer.
(SPACES N STREAM)	LAMBDA	Print N spaces on the stream STREAM.
(TERPRI &OPTIONAL STREAM)	*SUBR	Print a line feed on the stream STREAM. If STREAM is NIL or T the standard output is used.
(TYPE-READER TPE FN)	SUBR	Define the lisp function (FN TPE ARGS STREAM) to be a *type reader* for objects printed as #[TPE X...]. The type reader is evaluated by the ALisp reader when the pattern is encountered in an input 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 allow the I/O routines to work against dynamically expandible string buffers instead of files. This provides an efficient way to destructively manipulate large strings. Text streams can also be used for implementing large bit sequences ('blobs'). The following ALisp functions are available for manipulating text streams:

Function	Type	Description
(MAKETEXTSTREAM SIZE)	SUBR	Create a new text stream with an initial buffer size. The system automatically extends the initial size when necessary.
(TEXTSTREAMPOS TXTSTR)	SUBR	Get the position of the read/print cursor in a text stream TXTSTR.
(TEXTSTREAMPOS TXTSTR POS)	SUBR	Move the cursor to the specified position. This position is also updated by

		the regular Lisp I/O routines.
(TEXTSTREAMSTRING TXTSTR)	SUBR	Retrieve the text stream buffer of TXTSTR as a string.
(CLOSESTREAM TXTSTR)	SUBR	Reset the cursor to position 0, i.e. same as (TEXTSTREAMPOS TXTSTR 0).

6. Error handling

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

(FAULTEVAL ERRNO MSG O FORM FRAME)

where

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

The ALisp default behaviour of FAULTEVAL first prints the error message and then calls RESET to reset Lisp.

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.

The system may be run also in *debug mode* where assertions are checked and a break loop is entered by FAULTEVAL. (See Sec. 7. Lisp Debugging). The function (DEBUGGING T) enabled debug mode.

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

(CATCHINTERRUPT)

By default CATCHITERRUPT resets Lisp. In debug mode a break loop is entered when CTRL-C is typed. If ALisp profiling is turned on with (PROFILE-START) (Sec. 7.3.1 The Statistical Profiler), CATCHINTERRUPT will record what ALisp function was called when the interrupt occurred and then continue the evaluation.

6.1 Error Management

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

Function	Type	Description
(CATCH-AND-REPAIR TAG FORM REPAIRACTION)	MACRO	Evaluate FORM and return the result of the evaluation. Should THROW be called with the catcher TAG within FORM then the evaluation of FORM is abandoned and instead REPAIRACTION is evaluated and its result returned.

(CATCH-ERROR F1 F2)	MACRO	Evaluate F1 and return the result of the evaluation. Should an error occur during the evaluation of F1, then F2 is evaluated if supplied and an *error condition* is returned which looks like: (:ERRCOND (ERRNO "errmsg" XX)) where xx could be different things depending on the error. The function ERROR? can be used for testing if CATCH-ERROR returned an error condition. The functions ERRCOND-ARG, ERRCOND-NUMBER and ERRCOND-MSG are used for accessing error conditions.
(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.
(DEBUGGING FLAG)	SUBR	If FLAG!=NIL the system will start running in *debug mode* (See. Sec. 7. Lisp Debugging) where warning messages are printed and the system checks assertions. Turn off debug mode by calling with FLAG==NIL.
DEBUGGING	GLOBAL	Global flag indicating that the system runs in debug mode. Set by the function DEBUGGING.
(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. Lisp Debugging). If the system is not in debug mode FAULTEVAL prints the error message and calls (RESET).
(FRAMENO)	SUBR	Return the frame number of the top frame of the stack.
(HARDRESET)	SUBR	Does a 'hard' reset that cannot be caught to the top level reset point (usually the top loop). Called after fatal errors such as stack overflow.
(RESET)	SUBR	Return to the current reset point of ALisp. The rest point is either the ALisp top loop or the previous call to UNWIND-PROTECT.
(UNWIND-PROTECT FORM CL)	*SPECIAL	UNWIND-PROTECT enables the user to clean up after a local or non-local exit. 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. The FORM is evaluated as usual until it is terminated, whether naturally or by means of a regular exit or a RESET or THROW call. The cleanup form CL 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.

7. Lisp Debugging

This sections documents the debugging facilities of ALisp.

To enable run time debugging of ALisp programs the system should be put in *debug mode*. This is done by calling (DEBUGGING T). In debug mode the system checks assertions and analyses defined Lisp code 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 Alisp as described next.

7.1 The Break Loop

The break loop is a Lisp READ-EVAL-PRINT loop where some special debugging commands are available:

:help	print summary of available debugging commands
?=	print arguments of broken function
!value	Lisp variable bound to value of the evaluation after :eval or :rapply
:a	reset to previous break
(:arg N)	get N:th argument in current frame
(:b VAR)	continue evaluation until Lisp variable VAR is bound
:bt	print a backtrace of functions called
:btv	print a more detailed backtrace of the 10 top stack frames
:btv*	print a long backtrace including all stack contents
:c	continue the broken evaluation (only possible when current frame is where break loop was entered)
:eval	evaluate broken function body. Can only be done when current frame is frame where break loop was entered. Otherwise use :rapply. Variable !VALUE holds result from evaluation.
(:f FN)	search stack for call to function FN and set current frame to that frame
:FORM	variable bound to the Lisp form that was evaluated when the error occurred.
:fr	print current frame
:help	print this text
:nx	move current frame one step up the stack
:org	set current frame to the original current frame where the breakloop was entered
:pr	move current frame one step down the stack
:r	reset to ALisp top loop
:rapply	re-apply current frame (like :eval but can be done also when current frame is not frame where breakloop was entered. Variable !VALUE holds result from evaluation. Does not work for special forms or functions with &REST arguments.
(return x)	return value x from broken form
:ub	unbreak the present broken function

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. To print the complete variable binding stack use the function

```
(DUMPSTACK FRAME)
```

that print everything pushed on the stack starting at frame number FRAME.

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 FUM)
```

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

The arguments of the broken function are inspectable in the break loop, since variables in a break loop are evaluated in the lexical environment where the break loop was called.

You may put breaks around any kind of function, except special forms. Breaks on macros mean testing how they are expanded. If you break a SUBR the argument list is in the variable !ARGS.

The break points on functions can be removed with:

```
(UNBREAK FN...)
```

For example:

```
(UNBREAK FOO FIE FUM)
```

To remove all current function breaks do:

```
(UNBREAK)
```

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.

Conditional breaks

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 calls enters a break loop:

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

7.2 Tracing ALisp 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.

SPECIAL functions cannot be broken or traced. Macros can 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))
(TRACE (CREATETYPE (LAMBDA(TP)(EQ TP 'PERSON))) )
```

7.3 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 when user interrupts occurred. It produces a ranking of the most commonly called ALisp functions.
- The *function profiler* is useful when one wants to measure how much real time is spent inside a particular function. By the function profiler the user can dynamically wrap ALisp functions with code to collect statistics on how much time is spent inside particular functions.

7.3.1 The Statistical Profiler

The statistical profiler is turned on by:

```
(START-PROFILE)
```

After this the system will catch all user interrupts (CTRL-C) and update statistics on what code was executing when the interrupt occurred. Thus you should start the program you wish to profile and keep the CTRL-C button pushed down during the sections of the run you wish to profile.

When the statistics is collected a ranking of the 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.

Notice that regular interrupts cannot be caught when the statistical profiler is active.

7.3.2 The Function Profiler

To collect statistics on how much real time is spent in the specific ALisp functions FN.. and how many times they are called use:

```
(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 measure recursive functions calls or the time spent in functions causing error throws.

7.4 Debugging Functions

We conclude this section with a list of all ALisp system functions useful for debugging:

Function	Type	Description
(BACKTRACE DEPTH FRAME FILTERED)	SUBR	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 SUBRs are excluded from the backtrace. FRAME indicates at what stack frame number the backtrace shall start. Default is top of stack.
(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 The

		Break Loop).
(CLEAR-FUNCTION-PROFILES)	LAMBDA	Clear the statistics for function profiling (Sec. 7.3.2 The Function Profiler).
(DUMPSTACK &OPTIONAL FRAME)	SUBR	Print all the contents of the variable binding stack. If FRAME is provided it specifies the starting stack frame; otherwise the printing starts at the current top of the stack.
(HELP FORM)	MACRO	Put a break on FORM so that a break loop is entered when it is to be evaluated.
(IMAGE-EXPANSION RATE MOVE)	SUBR	When the database image if 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 not NIL the image will always be copied to a different place in memory after image expansion. If MOVE=NIL 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)	SUBR	Return the location (handle) of Lisp object X as an integer. The inverse is (VAG X).
(PRINT-FUNCTION-PROFILES &OPTIONAL FILE)	LAMBDA	Print statistics on time spent in profiled functions (Sec. 7.3.2 The Function Profiler).
(PRINTFRAME FRAMENO)	SUBR	Print the variable stack frame numbered FRAMENO.
(PRINTSTAT)	SUBR	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. 7.3.1 The Statistical Profiler).
(PROFILE-FUNCTIONS FN...)	MACRO	Wrap the ALisp functions FN... with code to collect statistics on how much real time was spend inside them (Sec. 7.3.2 The Function Profiler).
(REFCNT X)	SUBR	Return the reference count of X. For debugging of storage leaks.
(SETDEMON LOC VAL)	SUBR	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.
(START-PROFILE)	LAMBDA	Start statistical profiling of an ALisp program. See Sec. 7.3.1 The Statistical Profiler.
(STOP-PROFILE)	LAMBDA	Stop profiling the ALisp program. See Sec. 7.3.1 The Statistical Profiler.
(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).
(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.
(TIMER FORM)	SPECIAL	Print the real time spent evaluating FORM. Often used in combination with RPTQ.
(TRACE FN...)	MACRO	Put a trace on the functions FN... The arguments and the values will then be printed when any of these functions are called. Remove the tracing with UNTRACE.
(TRACEALL FLG)	SUBR	Trace *all* function calls if FLG is true. The massive tracing is turned off with (TRACEALL NIL).
(TRAPDEALLOC X)	SUBR	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.
(UNPROFILE-FUNCTIONS FN...)	MACRO	Remove function profiles from the specified functions (Sec. 7.3.2 The Function Profiler).
(VAG X)	SUBR	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. Time Functions

Time can be represented in ALisp either as *absolute time values* or *relative time values* (i.e. differences between time points).

2.1 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:

`(MKTIMEVAL SEC USEC)` creates a new TIMEVAL object.

`(TIMEVALP TVAL)` returns `T` if `TVAL` is a TIMEVAL object, otherwise `NIL`.

`(TIMEVAL-SEC TVAL)` returns the number of seconds for a given TIMEVAL object.

`(TIMEVAL-USEC TVAL)` returns the number of micro seconds for a given `TIMEVAL object`.

`(T< T1 T2)` returns `T` if the TIMEVAL `T1` is less than the TIMEVAL `T2`, otherwise `NIL`.

`(T<= T1 T2)` returns `T` if TIMEVAL `T1` is less than or equal to the TIMEVAL `T2`, otherwise `NIL`.

`(T> T1 T2)` returns `T` if the TIMEVAL `T1` is greater than the TIMEVAL `T2`, otherwise `NIL`.

`(T>= T1 T2)` returns `T` if the TIMEVAL `T1` is greater than or equal to the TIMEVAL `T2`, otherwise `NIL`.

`(GETTIMEOFDAY)` returns a TIMEVAL representing the wall time from a system call to C's `gettimeofday`. Useful for constructing time stamps.

`(TIMEVAL-TO-DATE TVAL)` translates a TIMEVAL `TVAL` 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.

`(DATE-TO-TIMEVAL D)` translates a date `D` to a timeval.

2.2 Relative Time Values

Relative time of day values are represented by the `TIME` datatype. It has three components, hour, minute, and second.

The following functions operate on relative times:

`(MKTIME HOUR MINUTE SECOND)` constructs a new TIME object.

`(TIMEP TM)` returns `T` if `TM` is a TIME otherwise `NIL`.

`(TIME-HOUR TM)` returns the number of hours given a TIME `TM`.

`(TIME-MINUTE TM)` returns the number of minutes given a TIME `TM`.

`(TIME-SEC TM)` returns the number of seconds given a TIME `TM`.

2.3 Relative Date Values

Relative date values are represented by the `DATE` datatype. It has three components, year, month, and day.

There are several functions that operates on dates and they are listed below:

`(MKDATE YEAR MONTH DAY)` constructs a new date.

`(DATEP DT)` returns `T` if `DT` is a date otherwise `NIL`.

`(DATE-YEAR DT)` returns the number of years.

`(DATE-MONTH DT)` Given a date `DT`, return the number of months given a date `DT`.

`(DATE-DAY DT)` returns the number of days given a date `DT`.

9. The Storage Manager

Amos II is tightly integrated with a storage management subsystem callable from C. 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 Lisp's `ROLLOUT`) and can be restored when restarting the system. 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. The storage manager is responsible for allocation and deallocation of *physical objects* inside the database image. However, while the system is running it is assumed that the entire database image is in the virtual memory. The system has primitives to save the image on disk or to restore persistent data from the disk.

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 since knowledge of the storage manager is needed for defining foreign ALisp 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 an 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.

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 Data Objects

With every handle there is an associated C structure, the *physical data 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 data type. 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 integercell
{
  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 unique *identification number* and a unique *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 (I/O 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.

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 data 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 in Amos II is very fast and does not involve any data copying; it involves just an offset computattion.

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 since 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 cause 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 is allowed to 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. Some built-in datatypes have allocation macros defined in `storage.h`:

`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);
   }
   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
`LISTP`	`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
`PIPE`	`PIPETYPE`	Internal
`TEXTSTREAM`	`TEXTSTREAMTYPE`	String 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 C library functions and macros are 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
`int getinteger(oidtype x)`	(macro)	Dereference a handle for an integer
`oidtype mkreal(double x)`	(macro)	Construct handle for a new real
`int realp(oidtype x)`	(macro)	`TRUE` if `X` is a handle for a real
`double getreal(oidtype x)`		Dereference a handle for a real
`oidtype mkstring(char *x)`	(macro)	Create handle for a new string
`int stringp(oidtype x)`	(macro)	`TRUE` if `X` is a handle for a string
`char *getstring(oidtype x)`	(macro)	Dereference a handle for a string
`oidtype new_array(int size,oidtype init)`		Construct handle for a new array with elements `init`
`int arrayp(oidtype x)`		`TRUE` if `X` is a handle for an array
`int a_arraysize(oidtype arr)`		return the array size
`oidtype a_seta(oidtype arr,int pos,oidtype val)`		Set an array element
`oidtype a_elt(oidtype arr,int pos)`		Retrieve array element
`oidtype a_vector(oidtype x1,...,xn,NULL)`		Create a new array and its elements x1 ... nn.
`oidtype cons(oidtype x,oidtype y)`		Create handle for a new list cell
`int listp(oidtype x)`	(macro)	`TRUE` if `X` is a list cell
`oidtype car(oidtype x)`	(macro)	Head of list cell
`oidtype cdr(oidtype x)`	(macro)	Tail of list cell
`oidtype a_list(oidtype x1,...,xn,NULL)`		Create new list of x1 ... xn
`oidtype mksymbol(char *x)`	(macro)	Create a new symbol
`int symbolp(oidtype x)`	(macro)	`TRUE` if `X` symbol
`char *getpname(oidtype x)`	(macro)	Get print name of symbol
`a_print(oidtype x)`		Print object of any type

`oidtype t`	Symbol `T` representing TRUE
`oidtype nil`	Symbol `NIL` 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.

`deallocfn`	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.
`printfn`	is an optional *print function* called by PRINT to provide a custoimized print function for every pysical object type.

10. Interfacing ALisp 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 A simple foreign function

As a very simple example of a foreign ALisp 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 making foreign ALisp functions in C; Foreign function definitions must always return ALisp objects of type `oidtype`. Do not forget the `return` statement!

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

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

A system convention is that a foreign ALisp function named `XXX` is named `xxxfn` in C, as for `HELLO`.

The call to register a foreign ALisp 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 **argc)
{
   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)
```

10.2 Defining foreign ALisp Functions

ALisp functions can be implemented as *foreign ALisp functions* in C. A foreign 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 *error 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); // Retrieves 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

```
   exfunction2("add",addfn);
```

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

Foreign ALisp functions need to be very careful to check the legality of the handles they receive. 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 `Into-Integer` is a convenient alternative).

Foreign Alisp 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 foreign ALisp function
`fn`	is the address of the C function.

Different versions of `extfunctionX` are available depending on the arity X of the foreign ALisp 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 foreign ALisp 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.

Many list manipulation functions are available as C functions and macros. For common list manipulations there are also simplifying macros defined in `storage.h`, e.g. the macros:

`cons(x,y)`	Lisp function CONS
`hd(x)`	The tail of a list.
`kar(x)`	Lisp function CAR
`tl(x)`	The head of a list.
`kdr(x)`	Lisp function CDR
`listp(x)`	TRUE if x is a list cell.

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 a foreign ALisp 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 foreign ALisp function where the system always checks the legality of data passed to ALisp from C. The debug mode is turned on with the ALisp call `(DEBUGMODE T)`.

10.2.1 No-spread foreign Lisp functions

No-spread foreign ALisp functions accept a variable number of arguments. Foreign ALisp functions with more than 5 arguments also need to be defined as no-spread functions. No-spread foreign functions always 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);
}
```

No-spread functions are the 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.

10.2.2 Special forms

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

C functions implementing special forms always have the signature:

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

Analogous to no spread 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 no-spread 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 since 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 `while` which also manages this memory deallocation correctly will be presented in the next section.

10.3 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 used the system error management, rather than e.g. directly calling C's *longjmp* or C++ error management.

10.3.1 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 CommonLisp [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 */
   }
}
```

10.3.2 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`:

`OfType(x,tpe,env)`	Raise a standard error if `x` is not of type `tpe`.
`IntoString(x,into,env)`	Dereference a symbol or string object `x` into a *pointer to* 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.
`IntoString0(x,into,env)`	Dereference a string object `x` into C string.
`IntoInteger(x,into,env)`	Convert integer object `x` into C integer.
`IntoDouble(x,into,env)`	Convert real object `x` 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.

10.4 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 in the call.
`a1,a2,...`	are the actual arguments in 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 = mksymbol("print"), lst = nil;

   a_setf(lst,l);
   while(listp(lst))
   {
      call_lisp(printsymbol,env,1,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.

10.4.1 Varying number of actual arguments

The following system function is used when calling an ALisp function with a dynamic number of arguments:

```
oidtype applyarrayfn(oidtype lfn, bindtype env, int arity, oidtype args[])
```

`lfn`	is the ALisp function to call.
`env`	is the error binding environment.
`arity`	is the actual arity.
`args`	is an array containing the actual arguments.

`call_lisp` is used when the exact number of arguments is known and the `applyarrayfn` when the number of arguments is dynamic and saved in the array `args[]`.

10.4.2 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 foreign ALisp 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 since 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;

}
```

10.5 Debugging functions

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 are 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.

10.6 Input/Output

AMOS II has two special datatypes for I/O streams:

`stream`	is for referencing C data streams.
`textstream`	is for streams over ALisp string buffers.

The following system functions and constants operate on streams:

`oidtype stdinstream` for C’s standard input stream

`oidtype stdoutstream` for C’s standard output stream

`oidtype stderrstream` for C’s standard error stream

10.7 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.ida.liu.se/imported/cltl/cltl2.html

2 Staffan Flodin, Martin Hansson, Vanja Josifovski, Timour Katchaounov, Tore Risch, and Martin Sköld,
Amos II Release 7 User's Manual,
http://www.it.uu.se/~udbl/amos/doc/amos_users_guide.html