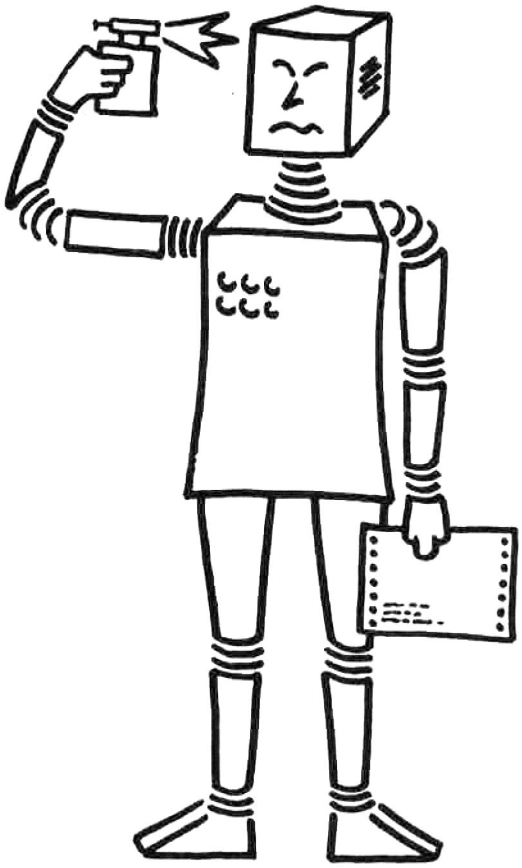
SEKI-PROJEKT

SEKI MEMO

Fachbereich Informatik
Universität Kaiserslautern
Postfach 3049
D-6750 Kaiserslautern 1, W. Germany



LISP - SP
A Portable INTERLISP Subset Interpreter for Mini-Computers

Elisabeth Hülsmann

MEMO SEKI-83-03

Abstract

This paper describes the implementation of LISP-SP, a decendant of LISPF3, an INTERLISP comaptible LISP system originating from DATALOGILABORIET, Uppsala, Sweden 1978.

Two data types have been added to LISPF3, namely floating point numbers and arrays, and a swapping algorithm as been implemented for data arrays.

Besides those new data types, a different method of storing lists internally has been adopted which is based on the idea that in LISPF3 each list cell consists of two memory cells, one for CAR and one for CDR, where in most cases, CDR contains only a pointer to the successor list element, and list space can be saved when keeping list elements ´physically´ adjacent as much as possible.

The CDR pointer is then available implicitly by incrementing the address to the internal structure carrying lists, except when the sequence of elements is broken through application of LISP functions. In those cases, special pointers are used which do not represent CDR values, but are only inspected to determine the location of the next element.

This method was selected to avoid program addressing space problems on mini-computers, resulting from the expansion of LISPF3´s internal data structures from 16-bit to 32-bit width, as required by LISPF3 users asking for ´more user address space´.

These enhancements of LISPF3 lead to a complete redesign of the system, where special attention was put on producing readable and self-documenting software. The resulting system differs enough from its predecessor to justify a different name, therefore, the name LISP-SP was selected, reflecting the fact that sublist structures are marked internally by (S)tored (P)arentheses.

When handling lists in this way, examples show that the number of memory cells allocated for a given package of LISP functions is approximately the same as in LISPF3. If, however, LISPF3´s data structures had been widened to 32-bit elements, then twice the amount of program addressing space for lists would be neccessary to store the same package - too much for most of the mini-computers currently available.

The system is available at a nominal charge from

Prof. Dr. P. Raulefs
Universität Kaiserslautern
Fachbereich Informatik
Postfach 3049
D-6750 Kaiserslautern
West Germany

Acknowledgements

Implementing a piece of software like this requires a lot of resources - computer time, printer paper, magnetic tapes,... and patience of the other timesharing users. Also, any help in testing or discussing ideas is welcome.

I want to tell my thanks to all, who were involved in this work: those wo helped me to understand the two computer systems I worked on, at the University of Bonn as well as at ATM Computer GmbH in Koeln, and to those, who developed and ran the test programs.

Then, I have to thank ATM Computer GmbH for providing unlimited computer time, the sine-qua-non condition for this project.

And thanks to Prof. P. Raulefs, who gave me the opportunity to do this diploma thesis work, and who supported my idea for implementing the system in the way it is described here.

Finally, thanks to C. Wedell at ATM, who - besides discussing ideas with me all the time - reviewed this paper.

Table of Contents

1 Preface

Although never supported officially by any computer manufacturer, LISP has maintained its role in the world of programming languages since its beginning in the late 1950's. Beeing the second oldest programming language after FORTRAN, LISP not only survived many attempts to replace it (e.g, by ALGOL), but gained importance in several application areas, especially in artificial intelligence, through an increasing number of implementations.

What, however, is LISP? Does it exist at all? These questions are raised in [DE79], entitled

THE LANGUAGE LISP DOES NOT EXIST ?

The problem discussed in this paper is the large set of different implementations of LISP systems, each of which contains a dialect of the language although most of them are based on McCARTHY's LISP 1.5. The main objection to classifying LISP as a programming language following [DE79] is

- lack of standardization
- lack of reference manuals
- minimal syntax
- large degree of freedom for implementors

No solution to these problems has been found so far, and there is still a variety of LISP systems around. To enlighten the roots of LISP-SP, a short overwiew on the LISP history will be given here.

The "creator" of LISP was J.McCARTHY, who gives a more detailed review of the LISP development in [MC78]. The original aim of his activities was to create a programming language for algebraic list processing purposes to support artificial intelligence work.

The first approach was FLPL (Fortran List Processing Language) which - as can be seen from its name - was based on early FORTRAN systems and contained some of the key ideas, but did not have features like conditional expressions or recursion.

At that time, not even the LISP notation in use today was known. Instead, a so-called "M-notation" was used in pencil-and-paper work, and using this notation for input-output purposes was not even possible due to te selection of symbols used in that notation.

Several new ideas, as for instance the desire for recursion, lead to the implementation of the first FLPL-independent compiler system in 1958, LISP 1. Meanwhile, the parenthesized prefiz notation was in use for external representation of lists. Also, "garbage collection" was introduced, and the function 'eval', discussed in recursion theory, turned out to provide a LISP interpreter.

Still a lot a features known today were not available in LISP 1, and in the early 1960's, LISP 1.5 was implemented which introduced property lists, list element insertion and -deletion, free variables and more efficient handling of numbers than available before. This was also the first compiler written in the language to be compiled.

After 1962, LISP had found its place in 'computer science', and as a consequence, different ideas were pursued at different places, leading to today's LISP babylon - INTERLISP, MACLISP, the Swedish LISPF1 and others.

Of all these systems, INTERLISP is probably the largest. It offers a set of features like syntax extensions, error correction and type declarations, resulting in availability of that system on larger computer systems only.

The LISPF1 system mentioned above, was written in 1970-1971 as an implementation of LISP 1.5, and was then rewritten conforming to INTERLISP as much as possible until 1978. This version was named LISPF3 ([MN78]), and this is the immediate predecessor of the LISP-SP described in this paper.

In summer 1981, together with two other students, on a practical course on software, I worked on enhancing LISPF3 by making available an additional data type, namely floating point numbers. As a prerequisite for this, we had to dive into the system structure to completely understand it, and thus becoming "LISPF3 experts", we were faced to the question by the LISPF3 users, if it would be possible to enlarge the "user space".

This problem was not only known to the students working with LISPF3, but evidently is a general one:

> "By far the most pressing problem for the user of a symbolic computing system is the problem of storage"

- words spoken at the 1980 LISP conference at Stanford University [SU80].

The LISPF3 system, running on a 32-bit ATM 80-60 computer, is implemented in FORTRAN IV. List storage is implemented by using two 'parallel' arrays, representing the CAR and CDR of list elements. These FORTRAN structures use 16-bit words, and enlarging the user space would involve two different modifications:

- larger array dimensioning

- pointer expansion to 32-bit

since larger arrays require the latter due to the address encoding necessary to distinguish the data types available.

What however would happen to overall system throughput, when reserving considerably more virtual memory?

This question was raised when reviewing the results of the practical course, and also the idea came up to further extend the features of LISPF3. One of the enhancements in question was making available arrays, and additionally, a swapper mechanism.

As we had experienced during the practical course, the LISPF3 system structure would become overloaded by implementing these features, and I decided to completely rewrite the system, based only on upward language compatibility, so that any LISPF3 program can run on the new system.

Besides, a different method of internally representing lists (with hopefully less storage space requirements) and the addition of the array data type and the swapper, a number of minor enhancements and error corrections to LISPF3 have been implemented in LISP-SP.

One of the goals in rewiting the system was to implement the software in a way that making further enhancements would be supported by well structured software - as much structured and clear as possible in FORTRAN, the language chosen for the implementation for portability reasons. Additionally, for this reason, this paper is not a LISP-SP 'reference-manual', but is a guide through the structure of its implementation.

On the other hand, a reference guide containing all function definitions had to be compiled anyway during the system implementation, since the LISPF3 documentation gives information on the differences to INTERLISP only, and the user is either faced to have two or three "large books" at hand ,when working at the terminal, or act as described by some other LISPF3 user:

> "First, I suppose the function I want to use exists. If the interpreter returns a message '--- UNDEFINED FUNCTION', I write the function myself."

The lack of documentation, the need to "try it out", is what "seems to be one of the characteristics of LISP systems", as [DE79] says. To assist the LISP-SP user as well as people who want to change LISP-SP in the future, the compilation of function definitions has been included as appendix to this paper. It is a brief description of the functions, however, and the user may have to refer to the definitions given in the literature.

In Chapter 2, the system structure is presented as a set of 'modules' or 'subsystems', each implementing a set of functions available to the other subsystems through certain 'primitives' like 'create-an-atom' or 'make-an-array-swappable' or 'get-the-value-of-a-number'.

Chapter 3 then describes each subsystem's functions and interfaces in detail, to enable the reader to understand system operation from a functional point of view.

As IISPF3, the system can be ported to other computers, since it is written in FORTRAN. Also, the space reserved for data storage, and several system parameters can be adjusted, when installing the system, to make it fit to a given computer's resources. Steps involved in 'system configuration' are described in Chapter 4.

Chapter 5 includes a description of some of the problems I faced during system implementation, some notes on the testing strategy and some proposals for enhancements. Also, the differences to INTERLISP are described briefly.

There are several appendices, giving more detailed information on LISP- as well as implementation aspects:

- app.1 -- List of Global System Variables and their Meaning
- app.1 -- FORTRAN Elements
- app.3 -- LISP-SP Reference Guide
- app.4 -- INTERLISP vs. LISP-SP Function Index
- app.5 -- References

2 The LISP Software System

This chapter is intended to give a general idea about the structure of this LISP implementation. It is not the my to give an introduction into LISP itself however; therefore the emphasis is to present the software structure on a global level.

The system is described as a set of modules or subsystems, where every subsystem provides a certain set of functions to the other subsystems. For each subsystem, its general purpose and the major access primitives are explained.

2.1 System Structure Overview

The general principle of operation of LISP systems which can be presented in a very few words, namely

```
LOOP  (PRINT (EVAL (READ)))
      (GO LOOP)
```

involves several steps of operation. Informally, these steps are:

a) accept a character stream from an input device, split it into tokens and recognize an s-expression,

b) store all tokens by conserving the s-expression's structure,

c) analyze the s-expression by recursively searching for "executable" sub-expressions and execute them, until the whole expression has been evaluated, and

d) print the resulting value,

e) then, repeat this sequence.

From these tasks, some of the general functions of a LISP system can already be classified as belonging to a software subsystem, as for instance "input-output" or "expression analysis". Other functions, however, can better be identified through a more technical view on the system functions, as for instance those parts of the software which actually perform the operations on user defined data. Subsystems resulting from that are the "storage management" and the "subr" subsystem.

Then, there are "subsystems resulting from the fact that certain "special" features" are desired which may be too complex or too "special" to be integrated into other subsystems. Here, the "roller" and the "swapper" belong to that class of functions.

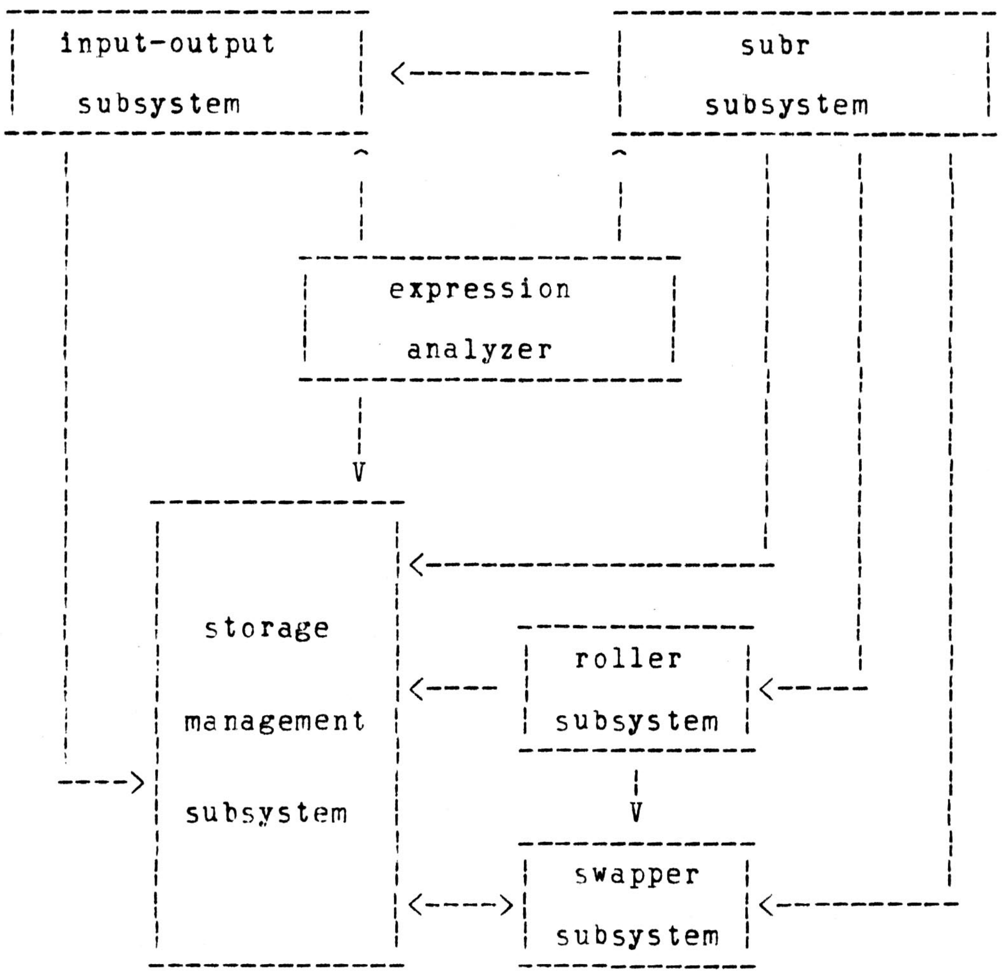


Figure 2.1-1
System Structure Overview

In the following section, a brief description of the different subsystems will be given in terms of their purpose and the interfaces available to the other susbsystems.

2.2 Subsystem Functions and Interfaces

Each subsystem offers a certain set of functions as described above. These functions are accessible through "FORTRAN-subroutines or -functions, or through setting global "flags" affecting the mode of operation of the subsystems. In this chapter, the term "user" refers to the subsystems "using" features of other subsystems.

Expression Analyzer

The expression analyzer plays the most important role in that it controls the general flow of execution. One of its tasks is to organize the recursion in s-expression analysis which is not available implicitly in the FORTRAN language. This is an implementational aspect however; the LISP related operations to be performed are

a) "parsing" s-expressions

b) determining the function type

c) evaluating the arguments

d) maintaining the association list

The expression analyzer consists of the main program, two initialization subroutines, and a "big" subroutine actually containing the expression parser. There are no ways to call the expression analyzer from other subsystems except by indirectly influencing the mode of operation.

Storage Management Subsystem

This subsystem manages the memory space available for storing the LISP data. LISP data are stored in arrays, and if some of them are exhausted then the subsystem tries to free space not actually in use by performing a decent garbage collection automatically.

To the outside world, the subsystem offers a set of callable functions for allocation and access of LISP data. These functions include

a) MATOM: create or identify an atom described through appropriate parameters.

b) MKNUM: create a floating point or integer number.

c) MKARRY: create an array with a specified size and initial content.

d) CONS: create a new list cell.

e) GETPN: fetch the printname of the specified literal atom or string.

f) GET: fetch the value of the specified property of an atom.

g) GETNUM: fetch a number value for a given floating point or integer.

h) GETEL: determine the type of some given item.

i) GETARG: resolve internal pointer-to-pointer references and deliver a valid data pointer and its type.

j) NEXT: determine starting address of the next top level element of a given list.

Input-Output Subsystem

The input part of this subsystem translates s-expressions from external ASCII representation into equivalent internal structures. Expressions are read on a character-by-character basis, where the characters are formed into "tokens", and more complex structures, the lists, conforming to a certain set of rules specified by a character type table.

Whenever a complete token has been recognized, it is handed over to the storage management subsystem for creating the internal representation. Input can also be influenced through several global flags like left and right margin, and actual read position.

The output part translates internal representations of data into readable form. As for input, print margins can be set by the "user", and additionally, flags are available to affect the printing format. The subsystem supports "fast printing" and "pretty printing"; also, printing of some special characters can be enabled or disabled.

Access to the input-output subsystem is through the following FORTRAN elements:

a) IREAD: read a s-expression.

b) RATOM: read an atom.

c) SHIFT: read next character from input stream.

d) PRIN1, IPRINT: print an expression (and flush buffer).

e) PRINAT: move an atom's printname to the printbuffer.

f) TERPRI: flush the print buffer.

subr Subsystem

The LISP system offers a set of built-in functions which serve as the basis to make available more complex functions. Whenever the expression analyzer has found a call to such a function, the subr subsystem is called with appropriate parameters to execute the desired function. Sometimes, a subr execution can be finished only after being supplied with more expression analyzer results; in these cases, it passes the information over to the expression analyzer for evaluation and "waits" for the results.

Subr-execution covers all types of LISP data; as a consequence, it refers to functions of all other subsystems. On the other hand, access to it is only possible from the expression analyzer. The "user" interface therefore consists of appropriate calls to the FORTRAN subroutines containing the subr code, supplied with an identifying function number and argument pointers held in certain global variables, such as the stack, for instance, and the function execution result is also returned through a global variable.

Swapper Subsystem

One of the data types available in this LISP system, the array, tends to occupy considerable memory space when used heavily. Sometimes, however, it is affordable to maintain arrays on secondary storage, and keep arrays in main memory only for access. In this case, only a buffer needs to be set aside permanently, into which arrays are mapped, when neccessary.

This method of managing array space is implemented in the swapper subsystem. The functions available to the other subsystems are:

a) MKSWPA: take away an array from access responsibility of the storage management subsystem and put it under control of the swapper.

b) UNSWPA: reverse of MKSWPA.

c) SWPIN: swap in an array from disk into the swap buffer.

d) SWPOUT: swap out an array currently in the swap buffer.

There are no functions available to access an array element in the storage management subsystem. The same is true for the swapper. The reason for this is that array element access is only done within the subr code for the corresponding LISP functions.

Roller Subsystem

The function of this subsystem is to copy binary system status images from and to disk. This feature is used as a quick initialization method for system start-up. Also, it can be accessed by any LISP user, who wants to stop the actual terminal session, and continue later starting with the results achieved so far beeing available without big effort.

Two access primitives are available:

a) ROLLOUT: copy system status to disk.

b) ROLLIN: read back system status.

3 Functional Description of the LISP System

3.1 Input-Output Subsystem

The purpose of the I/O subsystem is to

- read LISP expressions and data from the input channel
- convert the input to internal representation
- convert internal data into readable form
- print those data on the selected output channel.

Input/Output is driven by two important internal tables, one of which defines the semantics of single characters, and the other carries information on the I/O organisation.

Character Semantics:

Each character of the LISP character set is assigned a type which is stored as a number in the table CHTAB. The table is addressed using the ASCII character code of the character to be analyzed. The standard character type is 10, indicating that the associated charcacters do not have a special meaning and hence can be used directly in the names of literal atoms and strings.

Table 3.1-1 contains the list of legal characters and their types.

character	type	meaning
	1	space
(	2	left parenthesis
)	3	right parenthesis
<	4	left super bracket
>	5	rigth super bracket
'	6	quote character
"	7	string delimiter
#	8	user break
.	9	dot
others	10	all other ASCII characters
+	11	plus sign
-	12	minus sign
0..9	13..22	digit
{ or ^	23	escape character (1)
	24	rescue character

Table 3.1-1
LISP characters and their types

---

(1) Note that the escape character must be selected depending on the terminal type available.

The semantics of type 8, 23 and 24 are as follows:

User Break

The user break character enables the user to interrupt the input.

Example: A#B is treated as the 3 atoms A , # and B

escape character

The escape character changes the type of the following character to 10, invalidating the meaning of special characters.

Example: (SETQQ X {") defines an atom with printname ".

rescue character

The rescue character sets the interpreter into break mode.

The valid LISP characters are contained in the first record of the file ATOMS which is read during system initialization.

The character types can be changed by the user using the LISP function CHTAB.

Example: exchange the meaning of the characters "(" and "<"

```
(CHTAB '{( (CHTAB '{< (CHTAB '{(>
```

I-O Organisation

The table IOTAB contains 10 global parameters used during I-O operation as shown in table 3.1-2. For both input and output, the table keeps the actual channel to be used, the actual buffer pointer, and left and right margins independently.

Additionally, the output routines can be directed to print only to a certain expression nesting level; also, the number of top level elements to be printed can be restricted.

1	LUNIN	logical input channel
2	RDPOS	current read position
3	LMARGR	left read margin
4	MARGR	right read margin
5	LUNUT	logical output channel
6	PRTPOS	current print position
7	LMARG	left print margin
8	MARG	right print margin
9	LEVELL	# of top level elements
10	LEVELP	nesting level

Table 3.1-2
I-O organisation parameters

The LISP function IOTAB can be used to change the contents of the table, or to fetch the actual values.

```
Example: (IOTAB 1)    delivers the actual input channel number
         (IOTAB 1 22) sets the input to channel 22
```

The table can also be accessed by some LISP-expr´s which actually use the subr IOTAB, e.g.,

```
(INUNIT 22)   is equivalent to
(IOTAB 1 22)
```

For the last two elements of the table IOTAB, two expr´s are available referring to the subr IOTAB:

```
(PRINTLENGTH n)  <=>  (IOTAB 9 n)
(PRINTLEVEL  n)  <=>  (IOTAB 10 n)
```

Examples:

```
(SETQ X ´(A (B C (D (E F) G) H) K))

(PRINTLENGTH 2) X
will print as: (A (B C ...)...)


(PRINTLEVEL 2) X
will print as: (A (B C --- H) K)


(PRINTLENGTH 3) (PRINTLEVEL 3) X
will print as: (A (B C (D --- G) ...) K)
```

It may be useful to use the subr IOTAB instead of the expr´s defined in the system, since IOTAB is faster and less space consuming.

3.1.1 Input Handling

The input system always reads characters from the selected input channel, until a complete s-expression has been recognized. The input is split into "tokens" using the separators and break characters defined in the table CHTAB. The s-expressions are stored in internal presentation.

Input is always performed in a "stream"-mode, i.e., the input is treated as sequence of characters with no respect to line limits, blank lines and tabulations. Therefore, s-expressions can be entered in any format desired.

The input system involves three levels of action which will be described in the following sections.

3.1.1.1 List Handling

Input of s-expressions is controlled by the function IREAD. S-expressions may be lists, atoms or strings. A list is expected, whenever the first input token is an un-escaped left paranthesis.

By updating a parenthesis count, IREAD analyzes the input character sequence, splits it up into tokens (parentheses and atoms), and stores the atoms using the storage management functions.

Also, IREAD sets up an internal-format list (in the array LIST), which contains the pointers returned from the storage management functions for each atom. LIST access is done through the function CONS (see: storage management).

3.1.1.2 Atom- and String-Handling

For reading the next token from the input string, the function RATOM is used. If the actual character is a parenthesis or super bracket, it will be returned to IREAD directly. Space characters are treated as separators; they are not significant unless beeing escaped or occuring in strings.

If the first character of a token indicates an atom, all it´s characters will be fetched and stored using the appropriate storage management functions (MKATOM or MKNUM). The resulting pointer is returned to IREAD.

For strings, the function SHIFT will be called until the matching double-quote is read. Note that strings may contain any character except double-quotes, unless escaped. Since strings may be of any length, SHIFT takes care of storage allocation by using MKATOM.

3.1.1.3 Character Handling

The "stream"-input is performed by the function SHIFT. It converts input records into a sequence of characters using the array RDBUFF with respect to the left and right margins defined in IOTAB.

Besides delivering a character through a global variable (CHR), SHIFT determines the character's type using CHTAB. It is returned in the global variable CHT.

When identifying the escape character, SHIFT does not return it to the caller; instead the next character is read and returned with CHT set to 10.

A global flag is used to direct SHIFT to read from PRBUFF instead of RDBUFF. PRBUFF actually is the print buffer. This feature is used for internal analysis of print images. In this mode, SHIFT treats all characters as beeing of type 10.

3.1.2 Output Handling

Additionally to the features described above, the output format can be specified by setting several global flags contained in the array DREG. These are:

Flag	Value	Meaning
DREG(2)	NIL	fast print (no special formatting)
	T	pretty print
DREG(5)	NIL	don't print escape and duoble-quote
	T	print escape and duoble-quote
DREG(7)	NIL	continue on actual line if list fits
	T	start a new line with every new list

Table 3.1.2-1
Output Format Flags

If fast printing is specified, DREG(7) will not be examined. This is the default print format. Since it is much faster then pretty printing, fast printing is recommended except for printing complex list structures.

The print flags can be accessed through the LISP-subr SYSFLAG.

Output is done through the print buffer PRBUFF. An output line is actually printed, if

- the length of a print item causes PRBUFF overflow
- the subroutine TERPRI is called explicitly.

Four output subroutines are available which will be described in the following sections.

3.1.2.1 Expression Printing - PRIN1

The subroutine PRIN1 is used for writing expressions into the print buffer in external format. If the item to be printed is an atom, PRINAT is called. For lists, PRIN1 examines the print flags described above to determine print positions and line feeds. Also, the type of parenthesis to be used (normal or super bracket) is determined.

Whenever the print buffer is filled to the right margin and more has to be printed, TERPRI is called to flush the buffer. When leaving PRIN1, the print buffer may contain more data to be printed later.

3.1.2.2 Expression Printing - IPRINT

The subroutine IPRINT can be functionally compared to the LISP-expr PRINT. It calls PRIN1 for filling the print buffer and, after that, it flushes the buffer using TERPRI.

On return from IPRINT, the buffer is always empty.

3.1.2.3 Atom Printing - PRINAT

The subroutine PRINAT uses storage management functions to decode the pointer handed over by the caller. The printname is fetched, and it´s length is tested, whether it fits into the buffer. If not, the buffer is flushed before the printname is transferred into it.

PRINAT examines the print flag DREG(5): if it is set to T, the special characters (escape and double quote) have to be stored, where neccessary.

3.1.2.4 Buffer Flushing - TERPRI

TERPRI is the subroutine which actually writes the contents of the print buffer to the actual output channel. When output is done, TERPRI resets the buffer to spaces and the print pointer to the left margin.

3.2 Storage Management

This chapter is divided into two sections, reflecting the general functions provided by the storage management subsystem.

In section 3.2.1, the internal data structures used to implement the data types available in this LISP implementation are explained. Also, the access functions available to the other subsystems are described.

During a LISP run, storage for data of different types is allocated dynamically. Since memory space is limited, especially on small installations, and, a lot of data may become unaccessible during program execution, all unused memory should be made available to the user. The garbage collection methods implemented in this system are described in section 3.2.2.

3.2.1 Storage Allocation

3.2.1.1 Literal Atoms

Literal atoms are internally represented by records containing the following items:

- a sequence of bytes in the array PNAME for the atom´s printname
- a pointer to the first byte of the printname
- an integer number specifying the printname length
- a 32 bit word of memory for keeping the atom´s value binding
- a 32 bit memory word for keeping the atom´s property list
- an element in the hash array, containing a pointer to the value / property cell.

To establish a link between the internal and external representation of an atom, it´s name is used to compute the address of the hash table element containing the pointer to the value / property cell. This pointer also allows to access the printname pointer and length.

All input tokens which cannot be interpreted as numbers, strings or special characters, will be stored as literal atoms. Characters preceeded by the escape character will be treated as type 10 characters and will be stored directly without the escape character.

Printnames of literal atoms are stored in the same way as printnames of strings; they are packed to four characters per 32 bit memory word in PNAME, and are aligned on a byte boundary, if posssible. Since numbers are stored in the same part of PNAME, sometimes the printnames have to be aligned on the next full-word boundary following a number.

The data structures, used to represent literal atoms, and their interrelationship are shown in figure 3.2.1.1-1. They are implemented using a hash array HTAB, the array PNAME and three "parallel" arrays CAR, CDR (both 32 bit per element), and PNP (two consecutive 32 bit words per atom).

The CAR cell of an atom contains it´s value pointer which will be initialized to point to the atom NOBIND. The CDR cell (property cell) initially contains a pointer to the atom NIL. In PNP, printname pointer and length are stored.

Access to an atom´s constituents is done either directly or through access functions. To fetch or modify an atom´s value binding, the corresponding CAR cell is used directly; the same is true for fetching the pointer to the property list.

There are more complex operators, however. If access to the printname is neccessary, then the function GETPN can be used. It fetches the printname pointer and length, and also a flag is returned to indicate that the item is an ordinary literal atom.

If not the entire property list, but only the value of some indicator is required, the function GET, supplied with the atom pointer and indicator name, retrieves the indicator´s value, if defined, otherwise NIL. Adding properties (indicators and values) is done by the FORTRAN code for the subr PUT.

Literal atoms are created by a call to the function MKATOM either in resonse to user input or as part of certain functions.

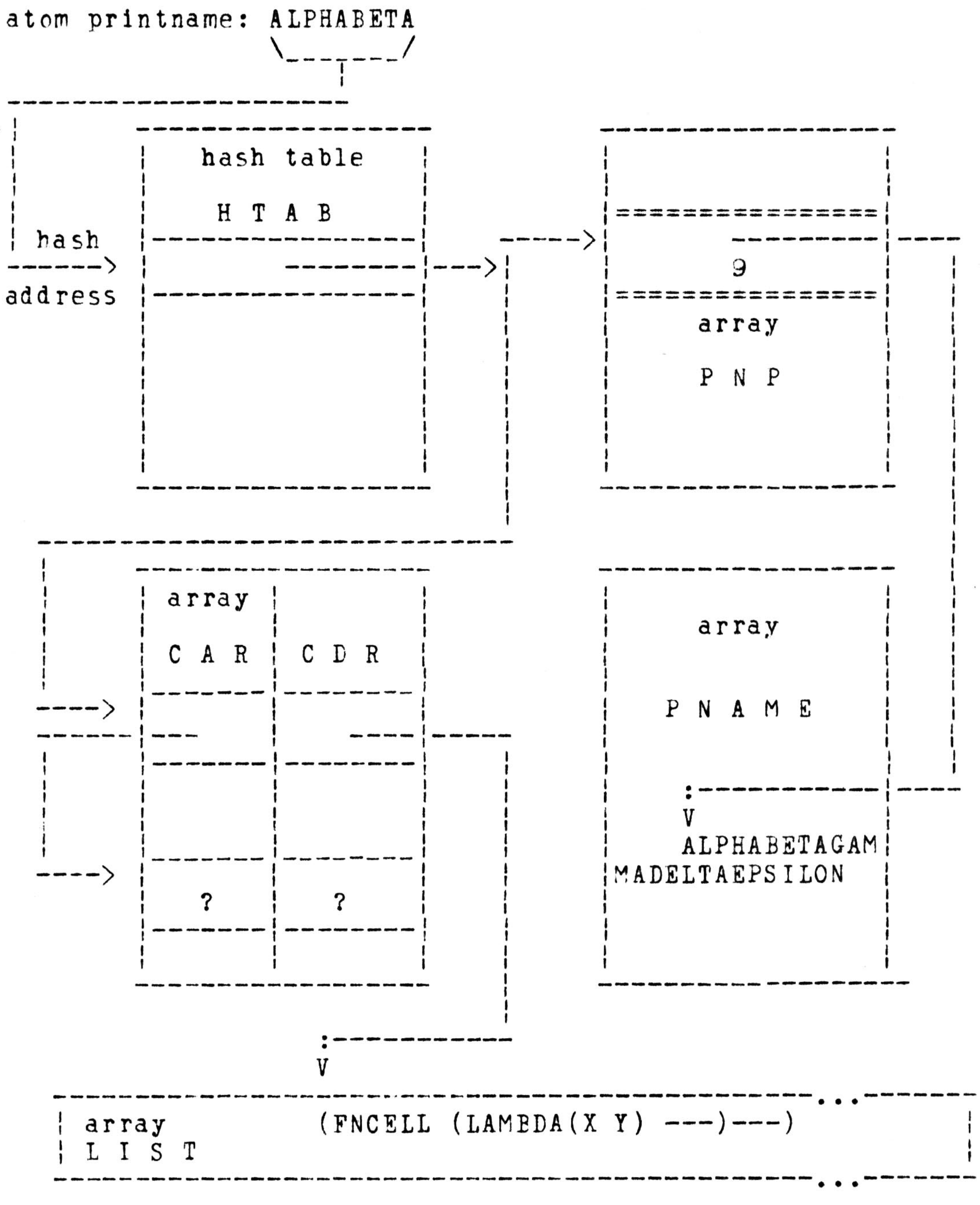


Figure 3.2.1.1-1
Literal Atom Data Structures

In the above figure, the atom ALPHABETA has as value a pointer to some other atom which may be UNBOUND, for example, and the cdr cell points to a property list containing a function definition.

3.2.1.2 Strings

Strings are internally represented as records containing the following items:

- a printname in PNAME
- a pointer to the first byte of the printname
- a number specifying the printname length
- a value cell in CAR
- a pointer cell in CDR
- for substrings, a list of a special format.

On input, strings are identified by enclosing double quotes. Inside strings, all characters except the double quote are treated as type 10 characters. To inhibit the special meaning of the duoble quote within strings, it must be preceeded by the escape character.

Strings are stored in nearly the same way as literal atoms, except that they do not use a hash table entry, and that substrings use a list. Figure 3.2.1.2-1 shows the data structures involved in storing strings.

Also, the value cell is not used to store values like in atoms, but instead, a pointer to the atom STRING or SUBSTRING is stored permanently in the corresponding CAR cell. The CDR cell contains NIL for strings, and a pointer to the list mentioned above for substrings.

This list has the following structure:

(main start . length)

where <main> is a pointer to the string which the substring is part of, and <start> and <length> are coded numbers specifying the offset and number of bytes of the substring.

Access to the CAR cell is done directly, as to the CDR cell. To obtain the printname, the function GETPN can be used as for literal atoms.

Strings are created by a call to the function MKATOM. Substrings are normally created within the subr code of the corresponding LISP function.

```
(SETQ X "ABCDEFG")
(SETQ Y (SUBSTRING X 4 5 ))
```

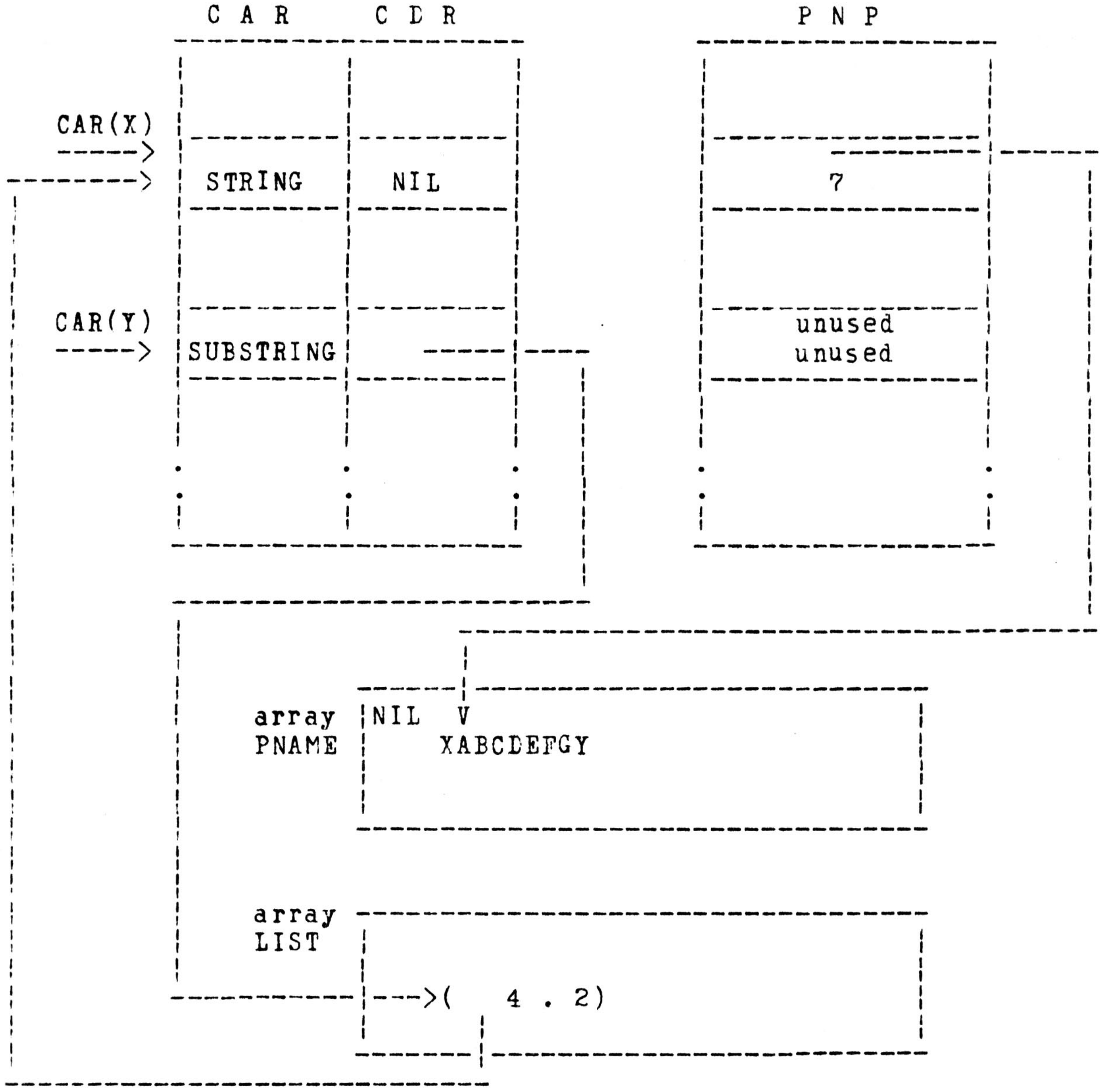


Figure 3.2.1.2-1
String Data Structures

The figure is somewhat simplified, as it does not show all data involved in the given example; for a complete overview, the atoms X, Y and UNBOUN also had to be shown. They do not give useful information on string and substring data structures, and therefore, they have been ommitted.

The CAR and CDR cells in the above figure actually contain pointers to the atoms STRING, SUBSTRING and NIL instead of the names. These pointers are addresses to the parallel arrays CAR, CDR and PNP.

3.2.1.3 Numbers

Two numerical data types are available in this LISP system:

- integers (32 bit)
- floating point numbers (32 bit)

For reasons of storage efficiency, integer numbers are stored either as "small integers" or as "large integers". The "small integers" are encoded using a function explained later. No PNAME space is reserved, but the encoded number is stored in the desired atom value cell, list cell or array element directly. If stored as a "large integer", a 32 bit word in PNAME is allocated into which the number is stored in binary format, and a pointer is encoded which then is put into the desired atom, list or array cell.

All numbers evaluate to themselves, when presented to the expression analyzer, therefore, their printnames never need to be quoted.

Encoding numbers is done in the following way:

Small Integers

If an integer number is in the range of [-NSMIN..NSMIN], then it is handled as small integer. Let x be the value of the integer, then it is encoded by

```
x +   NPNP         size of array PNP
  +   NLIST * 2    size of array LIST, twice
  +   NPNAME       size of array PNAME (without swap buffer)
  +   NSMIN        small integer limit
```

Large Integers

If the integer is not in the range [-NSMIN..NSMIN], then it is treated as large integer. In this case, a PNAME cell is allocated on a word boundary, the binary value is put into that cell, and the number pointer is calculated as follows:

Let j be the address of the cell allocated in PNAME to carry the integer's value, then the pointer is:

```
j +   NPNP       size of array PNP
  +   NLIST      size of array LIST
```

Note that large integers are allocated in the same part of PNAME as strings and literal atoms; therefore, two different allocation pointers JBP and NUMBP are used, and the printnames of strings / atoms and binary numbers are packed as much as possible to reduce the amount unused memory.

Floating Point Numbers

Floating point numbers are stored in the same way as large integers. Pointers to these numbers are set up in a slightly different way, however. Let j be the address of the PNAME cell containing the real number in binary format, then the pointer is calculated as:

```
j +   NPNP       size of array PNP
  +   NLIST      size of array LIST
  +   #8000000   which sets the first bit to one.
```

All numbers are created by the function MKNUM which is supplied with the binary value of the number and a flag indicating a real or integer number to be created. Access to a number is done using the function GETNUM, wich is supplied with a number pointer or encoded small integer, and it returns the binary value and a number type indicator.

LISP provides three types of numeric functions:

- integer functions

- real functions

- functions with value depending on the argument's types.

For integer functions, all arguments will be converted to integer, if not already, and the result returned is of integer type. For real functions, all arguments are converted to real, if neccessary, and the type of the result is real. For the third function type, the result is integer, if all arguments are integer, otherwise the result is real.

The type of a given numeric function can be derived from it's name: if the first character is an "I", then it is an integer function. If it is "F", then it is real, otherwise it is a function of the third type.

3.2.1.4 Arrays

Arrays are contiguous regions of PNAME space. They consist of a 4-word header, a number of cells reserved for "unboxed" numbers, and a number of cells containing any LISP pointer.

The arrays are stored in the upper part of PNAME; allocation of arrays is controlled by the global variable NARRYP. The header contains the total array size, the number of "unboxed number" cells, and two words normally set to Ø. These cells are used by the swapper.

The unboxed number section may be of length Ø, as may be the pointer section. This is specified by appropriate arguments to the subr ARRAY.

The purpose of the unboxed number region is to store integers in binary format, where it is up to the user to interpret these data in the desired way. Each pointer cell in an array can have a <car> and a <cdr>, therefore, the total array size computes to

4	size of header
+ p	number of unboxed number cells
+ 2 * j	where j is the number of pointer cells.

Arrays are created through the function MKARRY which allocates the neccessary PNAME space and initializes the header with the values mentioned above. Also, the unboxed number region is initialized to zero, and the pointer region (both <car> and <cdr>) is initialized to the value specified in the subr ARRAY arguments (which may be NIL).

After setting up the array, it's pointer is calculated in the following way: Let j be the address of the first header cell of the array in PNAME; then the pointer is

j + #4ØØØØØØ

Arrays also have printnames; these are formed by converting their pointer to ASCII hexadecimal presentation. Arrays cannot be read in using the interpreter's input functions, they can only be created by the function ARRAY.

Access to array elements can be done with the corresponding subr's ELT, ELTD, SETA and SETD.

3.2.1.5 Lists

In the LISP literature, the representation of lists is normally explained using the "box notation", where each list element consists of to cells, each of which carries a valid LISP pointer.

These double cells are implemented using two parallel arrays, CAR and CDR, where the CAR cell of a list element contains a pointer to its value, and the CDR cell points to the next list element.

In this implementation, however, each list element is represented by only one memory cell (in the array LIST) containing the value pointer. Lists are constructed through the subroutine CONS which allocates a cell of LIST space and enters the value to be CONS'ed.

The successor pointer is available through incrementing the LIST array address by one. There are several special constructs however; therefore, the internal representation of lists and the basic list operators are explained in detail in the following sections.

Since the box notation functionally remains valid also for this implementation, all constructs are explained both in box-notation and in a grafic representation equivalent to the internal structure.

I. List Structures

Sequences of List Elements

a) In box-notation, each list cell contains two pointers, one of which points to the value of the cell (which may be an atom, string, sublist or array), the other points to the next list element constituting the start of the tail of the list.

```
---------------      ---------------      ---------------
| car1 | cdr1 |----->| car2 | cdr2 |----->| car3 |  NIL |
---------------      ---------------      ---------------
  |                    |                    |
  V                    V                    V
  value 1              value 2              value 3
```

b) In this LISP implementation, there is only one memory cell for each list element, containing the car. The successor is defined in one of two ways: by either the next LIST cell or by the contents of the next cell which is called "continuation marker".

In the grafic representation, these markers are shown as cells containing an asterisk: "*".

Successor determined by next LIST address:

```
-------------------------------------------------
| (      | car1   | car2   | car3   | )      |
-------------------------------------------------
   n       n+1      n+2      n+3      n+4
```

Successor determined by continuation marker:

```
----------------------------------------
| (      | car1   | car2   |    *    |
----------------------------------------
  n                             |
                  ---------------
                  |        ------------------
                  ----> | car3  | )       |
                        ------------------
                          m       m+1
```

Note also that beginning and end of any list is represented by a left or right parenthesis, respectively. This is true also for sublists, as shown in the next figure.

Sublists
----

a) In box-notation, a sublist is determined by a car-pointer of some list element pointing to another list element:

```
----------------       ----------------       ----------------
| car1 | cdr1 |----->| car2 | cdr2 |----->| car3 | NIL |
----------------       ----------------       ----------------
  |
  |
  V
------------------       -----------------
| car11 | cdr11 |----->| car12 | NIL |
------------------       -----------------
  |                        |
  |                        |
  V                        V
```

b) In this system, the sublists are identified in two ways. Since the start and end of a list is always marked by appropriate parentheses, every left parenthesis within a list indicates the occurance of a sublist - which is the same in the external representation.

In the second case, a normal pointer to some LIST element indicates the start of the sublist, if this cell contains a left parenthesis.

Embedded sublist:

```
-----------------------------------------------------------------
| (   | (   | car11 | car12 | )   | car2 | car3 | )   |
-----------------------------------------------------------------
  n    n+1   n+2     n+3     n+4   n+5    n+6    n+7
```

Sublist pointer:

```
----------------------------------------
| (    | car1 | car2 | car3 | )    |
----------------------------------------
  n       |
          |     ---------------------------------
          |
          --->| (    | car11 | car12 | )    |
              ---------------------------------
                m      m+1     m+2      m+3
```

Note that in the first case, the sublist may be pointed to by a continuation marker, demonstrating that embedded sublists do not need to be physically embedded in the surrounding list elements.

Dotted Pairs

a) In box-notation, a dotted pair consists of a double cell, where the right cell does not point to a list. Lists can be thought of as constructed from dotted pairs, and hence, an equivalent has to be available in this system.

b) Here, a dotted pair is implemented as a list with a dot specified as character code directly, the dotted pair (A . B) will be stored as:

```
--------------------------------------------------
| (    | car1 |   .   | car2 | )   |
--------------------------------------------------
          |               |
          |               |
          V               V
        atom A          atom B
```

Note that the list (A . (B)) is equivalent to (A . B) - since parentheses are stored, it is up to the expression analyzer to skip superfluous parentheses.

II. List Access Functions

List access functions are neccessary in this system, since for instance a search in LIST may be neccessary to find the next top level element in a list - which is simply done by fetching the cdr of a list element in box-notation systems. The access functions available here are:

Find next top level element

The function NEXT, supplied with a pointer to the current list element, returns a pointer to the next list element. It resolves all embedded-sublist-skipping and continuation-marker evaluation.

Determine value of actual list element

To obtain the value of a list element in box-notation systems, only the car of the element has to be fetched. Here, a car-value may point to a LIST cell containing another pointer (to a pointer ...). The function GETEL just returns the contents of the specified cell after determining its type. The pointer returned may be a continuation marker.

The function GETARG resolves all pointer-to-pointer references including continuation markers and returns the real list element value and its type.

Accessing the first element of a list

Whenever the next top level element of a list has been identified as beeing a sublist (after finding it through NEXT), and this sublist has to be analyzed, access to the first element is done by incrementing the LIST address by one and using GETEL or GETARG, depending on what has to be done.

III. List-manipulating functions

When reading lists from the input channel, they can be stored sequentially in LIST, and no continuation markers are neccessary. There are several LISP functions, however which manipulate and construct lists internally. For these functions, the resulting structures are explained in this section.

The function CDR

a) In box-notation systems, the result of the CDR function applied to a list is just the contents of the CDR cell of the first element.

b) Here, the address of the next top level element of the list is fetched using NEXT, and then a list is constructed CONSing a left parenthesis and a continuation marker to this address. The result of the function CDR then is a pointer to the new list:

```
(SETQ X '(A B (C D)))
(SETQ Y (CDR X))

         ----------------------------------------------------
X --->  | (  |  A  |  B  |  (  |  C  |  D  |  )  |  )  |
         ----------------------------------------------------
                      ^
                      |
         -------------|
Y --->  | (  |  *  |---
         -------------
```

The function CONS

a) A double cell is allocated, and its CAR cell is filled with a pointer to the first argument, and the CDR cell receives the second argument.

b) A new list has to be created defining a dotted pair, if the second argument is not a list. If it is a list, then the result of CONS is a list which has argument 1 as the first top level element, and the top level elements of the second argument are also top level elements of the new list. The result of CONS in this case is:

```
(SETQ X (CONS 'A '(B C)))

          --------------------
X ---> |  (  |  A  |  *  |
          --------------------
                        |
                        V
                 ---------------------------
                 |  (  |  B  |  C  |  )  |
                 ---------------------------
```

Note that the above examples, the list elements contain pointers to the atoms A, B and C. For reasons of simplicity, these are represented by the atom names.

Note also that in the CONS example, no dot has to be specified, since a dot and a following parenthesis pair eliminate each other.

The function APPEND

a) in box-notation systems, a copy is made of all top level elements of the first argument. The CDR-cell of the last top level element is then filled with a pointer to the second argument.

```
          (SETQQ X (A (B C)))
          (SEYQQ Y (U (V) W))
          (SETQ  Z (APPEND X Y))

               -------------     -------------
      X ---> | A |   ---|----> |     | NIL |
               -------------     ---|---------
                                    |
                                    V
                                 -------------     -------------
                          ----> | B |   ---|----> | C | NIL |
                          |      -------------     -------------
                          |
                          ----------
               -------------     ---|---------
      Z ---> | A |   ---|----> |     |     |
               -------------     ---------|---
                ------------------------------
                |
                V
               -------------     -------------     -------------
      Y ---> | U |   ---|----> |     |  ---|----> | W | NIL |
               -------------     ---|---------     -------------
                                    |
                                    V
                                 -------------
                                | V | NIL |
                                 -------------
```

b) In this system, a new list is made from the top level elements of sublist, a pointer to the original sublist will be put in the corresponding position in the new list. Following the last top level element, a dot and a pointer to the second argument and a right parenthesis are stored. For the above example, the following structure results:

```
               -------------------------------------------
      X ---> | ( | A | ( | B | C | ) | ) |
               -------------------------------------------
                                ^
                                |
               ----------------|--------------------
      Z ---> | ( | A |       | . |       | ) |
               ---------------------------|---------
                 -------------------------
                |
                V
               -------------------------------------------
      Y ---> | ( | U | ( | V | ) | W | ) |
               -------------------------------------------
```

The function NCONC
----

a) In box-notation, the CDR cell of the first argument's last top level element receives a pointer to the second argument.

```
    (SETQQ X (A B C))
    (SETQQ Y (U V))

               -------------     -------------     -------------
       X ---> | A |  ---|----> | B |  ---|----> | C | NIL |
               -------------     -------------     -------------

               -------------     -------------
       Y ---> | U |  ---|----> | V | NIL |
               -------------     -------------

    (SETQ Z (NCONC X Y))

               -------------     -------------     -------------
     X,Z ---> | A |  ---|----> | B |  ---|----> | C |     |
               -------------     -------------     ----------|---
                 ---------------------------------------------
                 |
                 V
               -------------     -------------
       Y ---> | U |  ---|----> | V | NIL |
               -------------     -------------
```

b) Here, the end of the first argument is searched. Let j be the address of the list element preceeding the right parenthesis, then LIST(j) is CONS'ed into a new cell, and a continuation marker to this cell is put into LIST(j). Then, a pointer to the second argument, a dot and a continuation marker, pointing to the last right parenthesis of the first argument, are CONS'ed. For the above example, we have:

```
               ------------------------------
       X ---> | ( | A | B | C | ) |
               ------------------------------

               -------------------------
       Y ---> | ( | U | V | ) |
               -------------------------

               ------------------------------
     X,Z ---> | ( | A | B | * | ) |
               ------------------------------
                                 |     ^
               -------------------     |
               |                       |
               |       ----------------|---
               ----> | C | . |    | * |
                       ----------|---------
                                 |
                                 V
                           ----------------------------
                   Y ---> | ( | U | V | ) |
                           ----------------------------
```

The function NCONC1
----

a) In box-notation, a new cell is allocated, and a pointer to this cell replaces the contents of the CDR cell of the first argument's last element. The CAR of the new cell receives a pointer to the second argument, and the CDR receives NIL.

```
(SETQQ X (A B C))
(SETQQ Y (U V))
```

See the NCONC example for box representation.

```
(SETQ Z (NCONC1 X Y))
```

```
             -------------        -------------        -------------
X,Z --->  |  A  |  ---|----> |  B  |  ---|----> |  C  |     |
             -------------        -------------        ---------|---
                                    -------------------------
                                    |
                                    V
                                 -------------
                                 |     | NIL |
                                 ---|---------
                   -----------------
                   |
                   V
             -------------        -------------
  Y --->  |  U  |  ---|----> |  V  | NIL |
             -------------        -------------
```

b) Here, the same is done as in NCONC, except that no dot is CONS'ed.

For the above example, we have:

```
          ---------------------------------
X,Z ---> | ( | A | B | * | ) |
          ---------------------------------
                                |     ^
          ----------------------      |
          |                           |
          |     --------------------  |
          ----> | C |     | * |---
                ----------|----------
                          |
                          V
                     --------------------------
          Y ---> | ( | U | V | ) |
                     --------------------------
```

The function RPLACA
----

a) In box-notation, the contents of the CAR part of the element pointed to by the first argument is replaced by the pointer to the second argument.

Example 1:
(SETQQ X (A B C))
(SETQQ Y (U V))

See the NCONC example for box representation.

(RPLACA X Y)

```
           --------------     -------------     -------------
 X ---> |       |  ---|----> |  B  |  ---|----> |  C  | NIL |
           ---|----------     -------------     -------------
              |
              |
              V
           --------------     -------------
 Y ---> |  U  |  ---|----> |  V  | NIL |
           --------------     -------------
```

Example 2:
(SETQQ X ((A B) C))
(SETQQ Y (U V))

```
           --------------     -------------
 X ---> |       |  ---|----> |  C  | NIL |
           ---|----------     -------------
              |
              |         --------------     -------------
              ---> |  A  |  ---|----> |  B  | NIL |
                        --------------     -------------

           --------------     -------------
 Y ---> |  U  |  ---|----> |  V  | NIL |
           --------------     -------------
```

(RPLACA X Y)

```
           --------------     -------------
 X ---> |       |  ---|----> |  C  | NIL |
           ---|----------     -------------
              |
              |         --------------     -------------
              |         |  A  |  ---|----> |  B  | NIL |
              |         --------------     -------------
              |
              |
              V
           --------------     -------------
 Y ---> |  U  |  ---|----> |  V  | NIL |
           --------------     -------------
```

b) In this implementation, action depends of the type of the first top level element of the first argument. If it is a pointer to a number, string/substring, array or atom, this pointer is replaced by the second argument. This applies also for sublist pointers.

Example 1:
(SETQQ X (A B C))
(SETQQ Y (U V))

See the NCONC example for internal representation.

(RPLACA X Y)

```
            -------------------------------
  X ---> |  (  |      |  B  |  C  |  )  |
         ---------|---------------------
                  |
                  V
                  ------------------------
        Y  ---> |  (  |  U  |  V  |  )  |
                  ------------------------
```

For embedded sublists, a different method is used. The first left parenthesis of the list pointed to by the first argument is replaced by a continuation marker pointing to a new CONS'ed left parenthesis. Then, the pointer to the second argument and a continuation marker to the second top level element of the first argument are CONS'ed.

Example 2:
(SETQQ X ((A B) C))
(SETQQ Y (U V))

```
           ---------------------------------------------
  X ---> |  (  |  (  |  A  |  B  |  )  |  C  |  )   |
           ---------------------------------------------

           ------------------------
  Y ---> |  (  |  U  |  V  |  )  |
           ------------------------
```

(RPLACA X Y)

```
           ---------------------------------------------
  X ---> |  *  |  (  |  A  |  B  |  )  |  C  |  )   |
           ---------------------------------------------
              |                                  ^
              |         -------------------     |
              ------->|  (  |      |  *  |-----
                        ---------|---------
                                 |
                                 V
                          ------------------------
               Y ---> |  (  |  U  |  V  |  )  |
                          ------------------------
```

The function RPLACD
----

a) In box-notation, the contents of the CDR cell of the element pointed to by the first argument is replaced by the pointer to the second argument.

Example 1:
(SETQQ X (A B C))
(SETQQ Y (U V))

See the NCONC example for box representation.

(RPLACD X Y)

```
              -------------     -------------       -------------
      X  ---> |  A  |     |     |  B  |  ---|---->  |  C  | NIL |
              ---------|---     -------------       -------------
                       |
                       |
                       V
                    -------------      -------------
            Y  ---> |  U  |  ---|----> |  V  | NIL |
                    -------------      -------------
```

Example 2:
(SETQQ X ((A B) C))
(SETQQ Y (U V))

See example 2, function RPLACA for box representation.

(RPLACD X Y)

```
              -------------      -------------
      X  ---> |     |     |      |  C  | NIL |
              ---|-----|---      -------------
                 |     |
                 |     |
                 V     ------------------------------
              -------------      -------------      |
              |  A  |  ---|----> |  B  | NIL |      |
              -------------      -------------      |
                 ------------------------------------
                 |
                 V
              -------------      -------------
      Y  ---> |  U  |  ---|----> |  V  | NIL |
              -------------      -------------
```

b) Here, the last LIST cell belonging to the first top level element of the first argument has to be searched. Its contents is CONS´ed and replaced by a continuation marker pointing to the new CONS´ed cell. Then, the pointer to the second argument and a continuation marker pointing to the last right parenthesis of the first argument are CONS´ed.

Example 1:
(SETQQ X (A B C))
(SETQQ Y (U V))

See the NCONC example for internal representation.

(RPLACD X Y)

```
        -------------------------------
 X ---> | ( | * | B | C | ) |
        -------------------------------
              |                 ^
              V                 |
        -------------------     |
        | A |     | * |--
        ---------|---------
                 |
                 V
             -------------------------
   Y ---> | ( | U | V | ) |
             -------------------------
```

Example 2:
(SETQQ X ((A B) C))
(SETQQ Y (U V))

See example 2, function RPLACA for internal representation.

(RPLACD X Y)

```
        --------------------------------------------
 X ---> | ( | ( | A | B | * | C | ) |
        --------------------------------------------
                                |     ^
           ----------------------     |
           |                  -------------------
           ---------------->| ) |     | * |
                              ---------|---------
                                       |
            ---------------------------
            |
            V
        -------------------------
 Y ---> | ( | U | V | ) |
        -------------------------
```

3.2.2 Garbage Collection

Consider the sequence

```
(SETQ X 4000)
(SETQ X 5000)
```

In execution of the first expression, storage will be allocated for the number 4000, and a pointer to that cell will be returned and bound to the atom X. The PNAME cell used for the number will be used only as long as the atom X remains bound to it; the execution of the second expression will lead to the allocation of another PNAME cell receiving the value of 5000.

Then, the first cell will no longer be reachable, therefore, this cell should be freed for later re-allocation. Such a situation may arise for any data type, and hence, for each FORTRAN data structures involved in LISP storage management, a decent garbage collection method has to be available.

The arrays used for representating LISP data can be grouped for garbage collecting purposes as follows:

- CAR, CDR, PNP
- PNAME
- LIST

Whenever space in one of these groups is exhausted, the garbage collector is called to free any unused space. Also, the user can explicitly activate garbage collection. The garbage collection methods are described in the following sections.

3.2.2.1 CAR, CDR and PNP Garbage Collection

All literal atoms and strings currently in use are marked by negating their CDR-cell. Then, the active cells are compacted by shifting to the top as much as possible.

This is accomplished by searching for positive CDR cells and testing whether the corresponding CAR cell contains a pointer to NOBIND, STRING or SUBSTRING. If an unused cell is found, the next used cell is searched from the end, and its contents is filled into the unused cell. Let j be the address of some used cell, then HTAB(j) will receive the new or old address.

When all active cells are compacted, all data structures are searched for pointers to atoms, and the old pointers will be replaced by the values stored in HTAB. Finally, the hash table has to be refilled by a call to the subroutine REHASH.

3.2.2.2 PNAME Garbage Collection

The array PNAME contains four kinds of data:

- printnames of literal atoms
- printnames of strings
- numbers
- arrays

There is no need for marking printnames of strings and literal atoms, since only printnames are accessible which have a PNP entry. For marking numbers and arrays, an additional array MARK is used as a bit-map, where each bit represents one PNAME word. Each number occupies one PNAME word, and the corresponding MARK bit is set to one, if the number cell is active. For arrays, the bit corresponding to the first header cell is set to one, and also, this header word will be marked by setting bit 4, if the array is active.

For literal atom, string and number garbage collection, the following compression algorithm is used: starting with the printname following that of the atom T, string and atom printnames will be compacted by packing as much as possible through overwriting unused space.

Wenever an active number is found between printnames, it will be stored in the swap buffer which has been emptied on entry to garbage collection. Normally, the swap buffer will be large enough to receive all active numbers, but, if not, printname compaction will be interrupted, and the numbers collected in SWPBUF will be copied to the next free PNAME cells.

For each block of numbers, an entry is made in high HTAB which consists of the old address of the first number of the block and the length of the block. For printnames, the new address is written into the corresponding PNP cell immediately.

After emptying the SWPBUF, printname compaction and number collection can proceed in the way described above until all printnames and numbers are moved.

Since the printname pointers have been updated during compaction, only number pointers have to be changed in all data structures. For a given number pointer, this is done by calculating the block address and offset of the new number cell using the old address, the MARK bit-map and the entries made in HTAB for each block of numbers.

Arrays will be compacted by shifting their contents to the end of PNAME as much as possible. Active arrays are recognized by the MARK bit corresponding to the first header word being set - the size of a given array can be determined from its header.

when updating the array pointers, the new address for a given array is calculated using the array free space pointer NARRYP, the arraysize information, contained in each array header, and the MARK array.

If an array specified by some old address, x is determined as being the j-th array in sequence, then the new array address can be calculated from NARRYP and the length of the arrays preceeding it, since after compaction, all cells are active in the PNAME array space from NARRYP to NPNAME.

Both number and array pointer updating requires repeated sequential search in the MARK array which may cause considerable garbage collection execution time, especially for number-crunching applications. Suggestions for enhancements are made in chapter 5.

3.2.2.3 LIST Garbage Collection

For lists, all active elements are marked by setting bit 4 of the corresponding LIST cells. LIST compression is performed in a way similar to printname compaction: starting at the top, free space is searched and filled with contents of active cells which are searched starting at the end of LIST.

For each unused block of length, j>3, (j-1) active cells are searched from the bottom, their contents is replaced by a pointer to an unused cell, where the old value is stored. The j-th cell then is filled with a continuation marker which points to the block of active cells just copied.

After that, the next free and active blocks are searched and handled in the same way, until the array is compacted completely.

List pointers then can be updated through replacing them by the contents of the LIST cell they are pointing to.

3.3 Expression Analysis

The expression analyzer is the main part of the system. It controls all actions and calls the other subsystems, whenever needed. This chapter explains the functional structure of the expression analyzer. More details on its operation can be found in appendix 2, where flow of control and data is shown using an example LISP program.

Set system to initial state

Before starting a program run, or after occurance of fatal errors, several global variables and the stack have to be reset to bring the system into a clean status.

Read next item

After initialization or printing the result of the last expression evaluated, the interpreter requests the next expression to be input by calling the I-O handling subsystem which reads a complete expression, stores it in internal format and returns a pointer which enables the expression analyzer to access that structure.

Analyze item

Using storage management functions, the type of the item will be identified, and the expression analyzer acts depending on that type.

For numbers, strings or array pointers, nothing has to be done to evaluate them, because they evaluate just to themselves. Control is transferred to the function "recursion: popping".

For literal atoms, the actual binding has to be checked which may be either local on the actual association list (ALIST), or global through the atom's value cell, if it does not occur on the ALIST. If there is no value specified for the atom on the ALIST, and the value cell contains a pointer to UNBOUN, then control is transferred to the error handling section.

If a value is defined, then control is transferred to the function "recursion: popping" as above.

For lists, the global variables ARG1 and ARG2 are set to point to the first and second top level element, respectively. The first element always has to be a function name - if it is not, control is transferred to error handling.

A function can be a subr or an expr (one of their variants). If it is a subr, a new recursion level is initialized by setting up the stack properly, and then control is transferred to the subr-treatment.

If it is an expr, the atom has a property list, and the function definition is fetched from the property FNCELL. Each function definition is either LAMBDA or NLAMBDA; in the first case, the arguments have to be evaluated, therefore, control is transferred to the function "recursion: pushing", otherwise, to the function "LAMBDA/NLAMBDA treatment".

Recursion: pushing

The system uses two stacks, both implemented in an array STACK:

- the function stack
- the argument stack

Whenever a function has to be executed and the arguments have to be evaluated before execution, a function code is stored in the function stack. In the argument stack, all argument pointers belonging to the function code are stored.

In some cases it cannot be predicted, how many arguments will be stored for a certain function code. In this case, a pointer to the argument stack cell containing the first argument will be pushed into the function stack.

After pushing the function code, the expression analyzer has to evaluate the next item: it branches to the function "analyze item".

Recursion: popping

A function code has to be popped from the top of the stack. Depending on this function code, control is transferred to:

- LAMBDA/NLAMBDA treatment
- argument treatment
- subr-treatment
- error handling
- result printing

LAMBDA-NLAMBDA treatment

Recursion will continue until all bindings are established. The variable names (actually: pointers) and the associated values are stored in ALIST. Then, the function definition will be fetched, and a function code will be pushed, indicating the fact that multiple expressions may have to be evaluated due to the nature of LAMBDA and NLAMBDA bodies. Control is transferred to the function "recursion: pushing".

Argument treatment

For each argument in the expression actually evaluated, control is transferred to the function "recursion: pushing" which leads to evaluation of that argument, and a pointer to the result will be stored in the argument stack. After all arguments have been evaluated, control is transferred to the subr-treatment.

subr-treatment

If the function is a fsubr, the subr-subsystem will be called to perform all neccessary actions. For subr's, the argument treatment has to be activated to analyze the expression's top level elements.

After argument evaluation, the result pointers are stored in the global variables ARG1..ARG3 or in the stack, and the subr-subsystem is called to execute the function.

Error treatment

There are two kinds of errors:

- hard errors, the system has to be reset to initial state
- soft errors, a message is printed, then "read next item"

Print result

The input has been completely analyzed - the result is handed over to the I-O handling subsystem which performs the output. Control is transferred to "read next item".

Note that the subr subsystem decides itself, where to return to, since execution of subr code may request evaluation of expressions before completing, and, on return from a subr, error handling or result printing may be neccessary, for example.

3.4 The subr Subsystem

The subr subsystem consists of data structures and the FORTRAN code implementing a set of LISP functions which can either be used directly or as a basis for implementing more complex functions. These functions are sometimes referred to by the term "built-in functions" in analogy to predefined functions in other language processors.

In LISP systems, normally the term "subr" is used to indicate that a certain function is defined internally, as opposed to "expr" for functions defined as s-expressions. The term "subr" is sometimes preceeded by "f", indicating that the function is NLAMBDA which means, it's arguments are not evaluated before execution, and sometimes the character "*" is attached to the term "subr", indicating that the function allows for a variable number of arguments. These additional attributes are also posssible for "expr"-functions.

3.3.1 Internal Representation of subr's

For each subr, not only the FORTRAN code implementing the function has to be supplied, but also this code must be identifiable by the name of the function. Therefore, several lists of subr-names are provided in the file ATOMS which is read during system initialization.

Each of the 7 lists contains the names of functions of a certain type, which is defined by the number of arguments and other attributes. When reading the ATOMS initialization file, for each type, numbers are assigned to function names and saved in a global variable dedicated to that type. The names of these variables and their meaning are listed in table 3.3.1-1.

The function numbers are actually the numbers assigned internally to the atoms, whose names were read from the ATOMS file during initialization. When analyzing a LISP expression, the function name is used to determine the function number which is then matched to one of the above types. The interpreter can then identify the subroutine which contains the function code and transfer control to it.

The subr code is contained in the 6 subroutines listed in table 3.3.1-2. Also, for each subr this table shows the LISP function type of the functions contained in the corresponding subroutine.

variable	meaning
SUBRØ	number of functions with Ø arguments
SUBR11	highest number of numeric functions with one argument (add to SUBRØ)
SUBR1	highest number of functions with one argument (add to SUBR11)
SUBR2	highest number of functions with two arguments (add to SUBR1)
SUBR3	highest number of functions with three arguments (add to SUBR2)
SUBR	highest number of functions with arbitrary many arguments (add to SUBR3)
FSUB	highest number of NLAMBDA functions (add to SUBR)

Table 3.3.1-1
Identification of functions by numbers

subroutine	LISP function type
ISUBRØ	subr , Ø arguments
ISUBR1	subr , 1 argument
ISUBR2	subr , 2 arguments
ISUBR3	subr , 3 arguments
ISUBRN	subr*
IFSUBR	fsubr, fsubr*

Table 3.3.1-2
FORTRAN subroutines containing subr code

3.3.2 Execution of subr´s

Whenever the expression analyzer has found an executable list containing a subr´s function name, the function number is computed as described above, and the arguments are evaluated. Their values then are put into the corresponding global variables (ARG1 .. ARG3 for subr or argument stack for subr*), and, using the function number, control is transferred to the appropriate subroutine.

Since the argument values are pointers to the actual values, the latter are fetched using the storage-management function corresponding to the argument type. Then, the function number is used to branch to the requested function code via a table of FORTRAN statement labels.

In most cases, the function code covers all actions neccessary to perform the function, sometimes however, it is neccessary to use features implemented in the expression analyzer. In this case, the stack is set up properly to allow for a decent return to the subr code as well as the expression analyzer after the function execution is completed.

For a normal return, a pointer to the value resulting from the function execution is returned via the global variable ARG1. For error conditions, return is via the error handling section defined in each subroutine.

3.3.3 Execution of fsubr´s

For the fsubr´s, function execution is similar to the subr handling. However, the arguments are not evaluated before the function is activated. The fsubr code contained in the subroutine IFSUBR handles arguments in one of two different ways:

a) the arguments are not evaluated by the expression analyzer, but they are scanned directly within the function code. This is true for the functions QUOTE and SELECTQ, for example.

b) The arguments are evaluated under control of the function code by using the expression analyzer. This is the case for functions like PROG, where the arguments have a special meaning, as for instance the local variable bindings which have to be evaluated before execution of the sequence of s-expressions defining the PROG body.

3.5 The Roller subsystem

The roller subsystem allows for quick initialization of the LISP system. Once the user environment is set up by defining the desired set of LISP functions and data structures, a binary image of the system data can be stored on an external file for later use by calling the subr ROLLOUT.

This image can then be read back in during system start-up, or by calling the subr ROLLIN.

3.5.1 Creating a Binary Image

The subr ROLLOUT first calls the garbage collector (see: storage management) to free any unused space, thus minimizing the amount of data to be written to disk. Then, all global variables defining the actual system status are written in binary format to a logical channel specified by the user:

```
COMA    -- dynamic pointers to different FORTRAN arrays
COMB    -- COMMON area /B/ up to the system flags (DREG(7))
COMCH   -- COMMON area containing the character variables
CHTAB   -- character type table
HTAB    -- hash table
CAR,
CDR,
PNP     -- values defining atoms and strings
IMESS   -- message array
STACK   -- function- and argument stack
LIST    -- lists
PNAME   -- atom printnames, numbers and arrays
```

The output is actually performed by the subroutine DMPOUT.

3.5.2 Reading a Binary Image

A binary image created by the function ROLLOUT can be read back by the function ROLLIN. It first reads the dynamic array pointers (COMA) and checks them on consistency to the actual interpreter structure.

If the actual system lay-out allows to read in the data from the roller file (all dynamic pointers less than array bounds), the image is transferred to main memory, otherwise a message is issued.

Input is performed by the subroutine DMPIN.

3.6 The Swapper Subsystem

The swapper subsystem provides all neccessary features to extend the memory space available for one of the LISP data types, namely the arrays.

These are allocated in PNAME, as described in chapter 3.2 (storage management). Since PNAME space may be too restricted on a certain implementation, arrays can be maintained on a disk file instead of in main memory, transparent to the user. Once an array has been created using the appropriate LISP function, it can be made "swappable" through the function MKSWAP. Also, swappable arrays can be made resident in main memory.

Access to swappable arrays (i.e., retrieving the contents of array elements, or changing) is done by the normal array access functions. Additionally, the subr subsystem contains predicates to test array attributes.

3.6.1 Internal Presentation

Arrays are represented by pointers to their headers. For normal arrays, the header is followed physically in PNAME by the array body. For swappable arrays, there is also a header (of the same format) which just carries some additional information. The array body, however, is not allocated in PNAME, but on the swap file.

Whenever body of a swappable array has to be accessed (and, if it is not already swapped in), it is fetched from disk and transferred to the swap buffer (actually part of PNAME). The swapper data structures are shown in figure 3.6.1-1.

The swap table SWPTBL

The swap table is an array containing an installation dependent number of four-word entries. Each entry, when in use, contains information on a swappable array actually swapped in (body in the swap buffer).

Each entry consists of:

- a pointer to the array header in PNAME
- a pointer to the first swap buffer block allocated for the body
- the number of blocks allocated
- an additional cell, currently unused

Entries in SWPTBL are allocated dynamically, whenever an array has to be swapped in, and are freed, when swapping out the corresponding array.

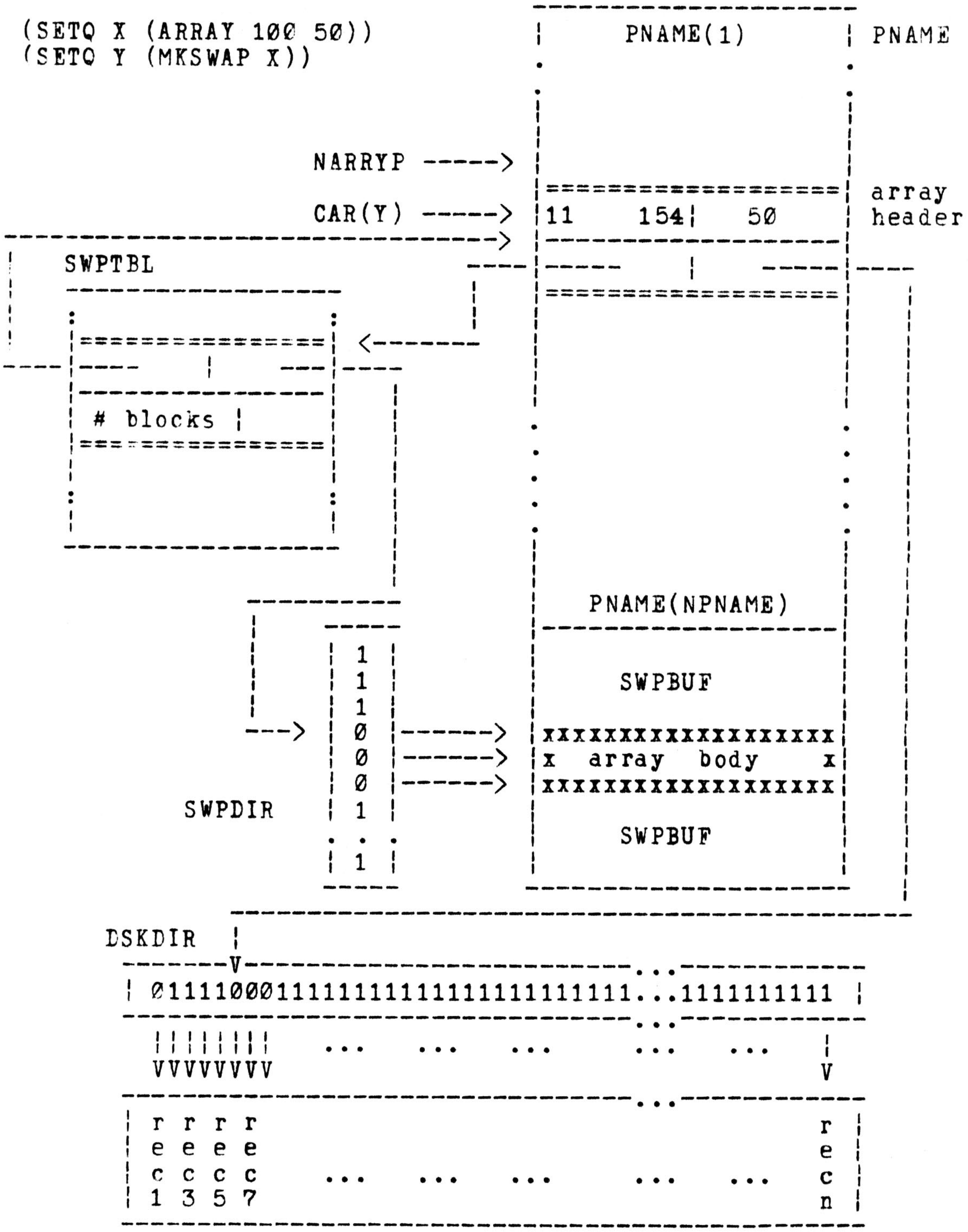


Figure 3.6.1-1
Swapper Data Structures

The array header in PNAME

Each array header consists of four words in PNAME. For normal arrays, only the first two words are used to store the arraysize and the size of the unboxed number region.

For swappable arrays, the first header word contains two additional flags, one of them indicating the array being swappable, the other marking the array as swapped in or swapped out.

The third header cell contains a pointer to the swap table entry allocated for that array, when it is swapped in. The fourth cell contains a pointer to the first record allocated on the swap file for that array.

The swap buffer

The swap buffer is organized as a sequence of 256-byte blocks in PNAME. Whenever an array is swapped in, the swap buffer contains a complete copy of the array contents, including the original header.

The number of blocks allocated for an array depends on it's size (first word of header). Only arrays smaller than the swap buffer can be made swappable. On the other hand, more than one array can be in the swap buffer at the same time, thus reducing swapper overhead.

The swap buffer directory

The swap directory SWPDIR is used to keep track of used/free blocks in the swap buffer. It is actually a bit-map: for an allocated block, the corresponding bit is off (Ø), and free blocks are marked by a 1-bit.

The swap file

For each swappable array, a number of 256-byte records is allocated on the swapper's disk file. The space allocated for an array is kept reserved for it, until it eventually is made resident again, or the swap file is initialized.

The swapper disk file directory

The array DSKDIR is used as a bit-map to indicate used/free blocks in the same way as SWPDIR for the swap buffer. In this implementation, the binary image of DSKDIR is written to the first record of the swap file, whenever it has been updated. This allows for keeping the swap file contents between different system runs.

3.6.3 Swapper Operation

The swapper subsystem contains four major functions resembling the possible status transitions of arrays:

```
      MKSWAP            ------------
       ------------| resident |
       | ---------->| array    |
       | |           ------------
       | |
       | | MKUNSWAP
       V |
   --------------    SWAPOUT       --------------
   |   array     |  ------------->  |   array     |
   | swapped-in  |  <-------------  | swapped-out |
   --------------    SWAPIN        --------------
```

Figure 3.6.3-1
Array Status Transitions

The swapper functions are called either during execution of the corresponding LISP functions or as a consequence of certain interpreter internal events, as for instance garbage collection. Therefore, the most swapper FORTRAN code is separated from the ISUBR´s and put into a set of FORTRAN subroutines and functions. These functions are described in the following sections.

3.6.3.1 Function MKSWAP

The FORTRAN function MKSWAP serves to

- allocate space for the array on the swap file
- copy the contents of the array from PNAME into SWPBUF.

Allocation of disk space is performed by calling the function FDSPAC which tries to find a proper number of consecutive blocks (records) in the disk file by inspecting the array DSKDIR.

If enough space is available, the records are marked as allocated, and the disk image of DSKDIR is updated. If not enough space is available, an error message is returned to the user.

Next, the function FBSPAC is used to provide space in the swap buffer. Eventually, other swapped-in arrays have to be swapped out before the array can be copied. Also, FBSPAC allocates a swap table entry and links it to the array header. Shortage of swap table entries will also force some swapped-in array to be swapped out.

MKSWPA then sets up more array header and swap table information, and then the array is copied from PNAME to SWPBUF using the function MVARRY. The array header remains valid: all pointers to it will now point to a swappable array. The former array body however, will be lost during garbage collection.

3.6.3.2 Function SWPOUT

The function SWPOUT is called in the following cases:

a) swap buffer space needed during MKSWAP or swap-in
b) swap table entry needed for MKSWAP or swap-in
c) the subr ROLLOUT has been called
d) the subr EXIT has been called
e) garbage collection is called

The function SWPOUT fetches all neccessary information on the array to be swapped out from it´s header and swap table entry. Starting at the SWPBUF address (fetched from SWPTBL), the appropriate number of blocks (also fetched from SWPTBL) is written to the swap file starting at the first record allocated for the array (fetched from the header).

Next, the swap buffer directory SWPDIR is updated, the swap table entry is freed and the array is marked as swapped out in it´s header.

3.6.3.3 Function SWPIN

This function is called during execution of the LISP functions ELT, FLTD, SETA, SETD and MKUNSWAP. It first checks, whether the array is already swapped in. If not, space is allocated in SWPBUF through a call to FBSPAC which may cause swap-out of other arrays, as described above.

The swap table entry and the array header are set up, and the array contents is read into the swap buffer from disk. SWPIN returns a pointer to the first word of the array header in SWPBUF which allows the caller to treat the array as a normal resident one in the following array operations.

3.6.3.4 Function UNSWPA

This function is used to make a swappable array resident. It first swaps the array in using SWPIN, then PNAME space is allocated through MKARRY (see: storage management) and the array contents are moved to the space allocated by a call to MVARRY. The swap table entry and the space allocated on disk and in the swap buffer are freed by updating SWPDIR and DSKDIR. The LISP function MKUNSWAP returns a pointer to the new array; the old array header is still available, however, if there are any bindings to it occuring in variables, lists or arrays.

4 Porting and Installing the System

The LISP system is implemented in FORTRAN-IV as its predecessor LISPF3 to provide a high degree of portability. Therefore, to install the system on a given computer, only a few modifications of the source code have to be made, mostly where functions provided by the operating system are referenced.

The LISP system is implemented on two different computers currently, namely the MODCOMP CLASSIC (ATM 78) and the ATM 80-60. Operating system dependent features are therefore clearly identified in the source code already by providing a file for each computer containing the critical functions and subroutines.

To install the system on another computer, a file of subroutines and functions has to be provided which interface between the LISP and the operating system. Also, for other FORTRAN-IV versions, it may be neccessary to modify the system source elsewhere, as for instance to change statement ordering, or character translation and storage.

Once the source code is accepted by the FORTRAN compiler of the computer system in question, the software has to be "configured" to define the sizes of the internal data structures. Since FORTRAN allows only static data structures, all arrays have to be dimensioned properly, and a number of variables have to be preset, which enable the system to know about the software configuration.

All programs involved in system initialization are collected in one file, namely the file INIT. In most cases it will therefore only be neccessary to modify the contents of this file to tailor the system. Only, when changing the system structure (e.g., to enhance the subr set), other subroutines might be affected. This especially applies when changing the system "data base", the COMMON structure which on the ATM 78 system is held in a separate file, but is available in each subroutine directly for the ATM 80-60 system due to the missing "include" facility.

In the following sections, the steps involved in system installation are described.

4.1 Data Structure Configuration

The subroutine INIT1, contained in the INIT file, presets all variables related to data structure lay-out and variables relating to machine dependent features. Here, the data structure lay-out will be described. For a specification of machine dependent variables, refer to appendix 1.

Note that for changing any array declarations, they have to be changed, wherever they occur, - in file VAR, if "include" is available, and in each subroutine or function, if not.

Array LIST

Since in this array, all lists created during system execution are stored, it should be dimensioned as large as possible. A minimum size of 10000 elements is recommended for systems providing the complete expr package to the user. Set the variable NLIST to the value desired, and change the declaration of LIST appropriately.

In this version, no data structures are equivalenced to LIST, and therefore nothing else has to be done.

Arrays CAR, CDR, PNP, HTAB

Since these arrays are all used to store literal atoms, their sizes depend on each other. The array HTAB is used as hash table, and must therefore provide more elements than the expected number of atoms used in a system run. Let n be this number, then CAR and CDR each have to be configured to n elements. The size of HTAB then should be n*1.5 or more.

Also the size of the array PNP depends on the size of CAR and CDR: it must be configured to 2*n+1. Besides declaring the arrays appropriately, the variable NHTAB has to be set to the size of HTAB, and the variable NPNP has to be set to 2*n.

Array STACK

This array is used to maintain the two stacks (function- and argument stack). It should be declared to not less than 500 elements. The variable NSTACK has to be set to the size of the array. There is no relation to other data structures.

Array PNAME

This array is used to store atom and string printnames, numbers and arrays, and therefore should also be configured as large as possible. Also it contains the swap buffer which is equivalenced to its upper part. The variable NPNAME specifies the upper data storage limit, where NSWPBW gives the size of the swap buffer in PNAME elements. PNAME must be declared to the desired size, and the variables NPNAME and NSWPBW have to be set appropriately (see also swapper data structures).

Note also that real values will be stored in PNAME and to inhibit data conversion by FORTRAN, a REAL array RPNAME of the same size as PNAME is equivalenced to the latter. The RPNAME declaration has to be changed.

Swapper Data Structures

The swap buffer SWPBUF mentioned above is equivalenced to PNAME as follows:

```
SWPBUF(1)       = PNAME(NPNAME+1)
                = PNAME(NSWPBB)

SWPBUF(NSWPBW) = PNAME(NSWPBE)

and therefore,
NSWPBW          = NSWPBE - NSWPBB + 1
```

The variables NSWPBW, NSWPBE and NSWPBB have to be set conformingly.

The swap buffer and the disk file are accessed on a record basis for data transfer and storage management purposes. The number of records or "blocks" in the swap buffer is then:

```
NSWBLK = NSWPBW / (NBPREC / BYTES)
```

where NBPREC and BYTES are machine dependent values.

The array SWPDIR must be declared to provide one bit for each block in the swap buffer, and its size has to be specified in the variable NSWPDI as:

```
NSWPDI * 32  =,>  NSWBLK
```

For each swpped-in array, a four-element entry is used in the array SWPTBL, so that its length, specified in the variable NSWPTB, must be

```
NSWPTB = 4 * NSWPTE
```

where NSWPTE then specifies the number of swap table entries available.

Finally, the array DSKDIR has to provide one bit for each record in the swapper's disk file. It is reasonable to dimension the DSKDIR to a multiple of the record size, since it is maintained on the first records in the swap file for re-initialization. Let NDIRSC be the number of disk records used for the disk directory, then DSKDIR must be declared to

```
NDSKDI = NDIRSC * (NBPREC / BYTES)
```

elements.

Buffers

The variable IOBUFF has to be set to the size of the arrays ABUFF, RDBUFF and PRBUFF. These arrays are statically initialized in the block data segment.

Array IMESS

This array keeps all system messages, and it has to be declared to

```
MAXMESS * (NBMESS / BYTES)
```

where MAXMESS is the number of messages available, and NBMESS is the maximum message length.

Array CHSET

This array is equivalenced with the variables keeping characters of special meaning. Besides proper CHSET declaration, the variable NCHTYP has to be set to the size of CHSET.

4.2 Operating System and non-standard FORTRAN Calls

For several purposes, bit manipulation is used internally. This is done by four functions:

```
SETBT    SETBF    TESTB    ISHFT
```

Conforming to bit ordering and true/false value assignment, appropriate FORTRAN functions have to be provided.

For direct disk access (swapper), two routines RDREC and WRREC have to be supplied. The system time has to be delivered through the routine TIMDT4.

Finally, character fetching and stuffing is expected to be done by GETCH and PUTCH. Different versions are supplied with the system. The ATM-80-60 version can be used for systems supporting LOGICAL*1 data type.

4.3 System Initialization Files

The file ATOMS contains character definitions, the subr names, some variable names and the system messages. It is read during initial system start-up, when no ROLLIN file is available. Of special interest are the character definitions which might have to be changed on a given system.

In this case, also the file SYSPACK might have to be inspected, and possibly characters may have to be changed there also.

The SYSPACK file contains all expr definitions available as described in appendix 3. More function definitions may be added, or functions may be deleted or modified without affecting the system structure.

The functions available are packaged in the following way:

BASIC1	BASIC2	IO1	FUNC1
DEBUG1	DEBUG2	EDIT	MAKEFILE
I&F-PACKAGE			

Package functions may refer to functions defined in other packages, so care must be taken, when deleting any function definition.

4.4 System Environment

Several files and devices have to be provided for proper system operation. These are:

standard input device	(LUNIN)
standard output device	(LUNUT)
ATOMS file	(LUNSYS)
SYSPACK file	
roller file	(LUNROL)
swapper direct access file	(LUNSWP)

Additionally, there may be

alternate input / output files
alternate roller files
MAKEFILE files

When starting a new run, the user is asked for the type of the system desired which may be either "clean" (only "subr's defined), or a predefined binary image provided by the "standard" roller file. When using the first type of system, a direct call to ROLLIN may be used with the appropriate logical channel number to read in a user specific binary image.

The initialization method may be changed, however, by adding or deleting code in the "main program" INI.

5 Conclusion

5.1 Portability Problems

The system is implemented in 'pure' FORTRAN IV, avoiding the use of the extensions in the ATM-78 FORTRAN, and all critical operating system functions have been collected in system dependent files, I experienced some surprise, when installing the software on the ATM-8060.

Nearly no one of the 20 or more source files was compiled correctly on the first attempt. Most of the problems came from the fact that the ATM-8060 FORTRAN performs some type-checking for subroutine and function parameters, as long as these are collected in one file and compiled together - a nice feature for those, who can afford to use it.

There are more serious problems, however; as for instance the rejection of statement functions within functions, or problems related to FORTRAN internal storage lay-out, documented nowhere - at least not in accessible documentation.

One of these storage problems on the AEG-8060 is the limitation on array sizes, where no FORTRAN array may be larger than 64 k bytes. This restriction is not applicable, when running LISP-SP on other machines.

Originally, I implemented the swapper in a way that the swap buffer was part of the LIST array as well as PNAME - thereby providing the basis for later implementation of swappable function definitions. This was achived on the ATM-78 by overlaying LIST and PNAME partially, but as a side effect of the size restriction for arrays, it was not possible to implement on the ATM-8060, and therefore is not available on the latter.

This is no real serious problem, however, since very likely, a better way to implement swappable functions is to provide a separate swapper for LIST, involving a different swap buffer management algorithm reflecting the need to keep a function definition in memory until it is completely executed.

5.2 Testing the Interpreter

LISP is a very friendly system, seen from the tester's point of view, since system operation can be verified beginning at a very early development stage. Once the expression analyzer allows for calling subr's, each subr can be implemented and tested immediately. Also, if the subr code is known to perform correctly, it can be used to test the expression analyzer which can be stabilized with reasonable effort only by additionally using some tracing mechanism.

Besides throughput aspects, the availability of a symbolic tracer on the ATM-78 was one of the reasons for originally implementing the system on this machine.

Once the system seemed mature enough, more complex tests were derived from example programs in [EP79] and [WH81], the biggest one being the animal identification problem. Some changes to the program had to be made, however, to convert MACLISP into INTERLISP.

More test packages then where drawn from the LISPF3 expr package, e.g., the editor, and a number of problems were identified and fixed.

Since no fully symbolic debugger was available, two additional features have been implemented to aid in testing. One is the WSTACK routine which can be called anytime to print the contents of the argument and function-stack, and which can be activated at the most important places in the program by just setting the sysflag 3 (DREG[3]) to T.

Another useful assistant is the 'debugger' which, when activated through a backslash in column 1 in the input, accepts command lines specifying global variable names (system status) and prints their contents in any format desired. This feature is always available, and the debugger's symbol table is installed as a normal LISP array, and can as such be made swappable, or deleted, of course.

There is one function in the system which is more difficult to test, namely the garbage collector. Especially for testing it, the 'debugger' was of great importance. Once the garbage collector was stabilized enough to do so, the ackermann function was used to create heavy storage management activity, and then a more complex program was written to test especially list management.

This program allowed for relaxing a bit in the most difficult implementation phase: it required to sit in front of a character driven color grafics terminal, generating circles of growing sizes and different colors. Whenever the sequence of colors was out of the expected order, a problem had been found in garbaged list structures.

5.3 Proposals for Improvement

Software never is perfect - this applies to LISP-SP as to any other program. The problem just is that time is limited, and new ideas cannot be implemented if work is ever planned to be ended. If someone wants to pick up LISP-SP and make it better - here are some hints about what can be done.

First of all, swappable lists would probably add most to LISP-SP's attractiveness, and, based on the FORTRAN elements available for array swapping, it should not be too hard to do.

Then, it may be reasonable to change the storage management for numbers: in the current version, due to the limited hardware environment which was available, numbers are stored interleaving with atom and string printnames. This leads to a garbage collection algorithm involving sequential searches for number positions, and in 'heavy number crunching' applications can imply high overhead.

If LISP-SP is ever going to be used for that purpose, it might be worth to set up a separate array for storing those numbers, change MKNUM, GETNUM and the other related functions appropriately, and then - the most complicated part of the job - change also garbage collection.

Also LIST garbage collection might be changed: pointer-to-pointer references might be resolved to compact lists by trying to store all elements of lists sequentially without use of continuation markers. This will require a highly sophisticated algorithm, if it is possible at all. There is no doubt, however, that list garbage collection will take more time in this case, and no estimates can be made on the amount of space freed additionally.

Another, and again, far more interesting enhancement might be to change the ALIST and property list management to avoid sequential searches for variables and properties.

Also, the user may wish to add subr's to the system - this is only as complicated as the functions themselves are. Embedding new subr code into the system can easily been done by using the FORTRAN elements and documentation available.

Finally, implementation of some sort of 'compiler' would be an enhancement of major importance - certainly a project requiring quite some amount of time.

5.4 Differences to INTERLISP

LISP-SP implements only a subset of INTERLISP for use on today's 'mini-computers'. All functions of major importance are available, and can be used to implement more complex features. This can be seen by inspecting the expr package, containing an editor, debugger, makefile and others.

Some of INTERLISP's features require additional FORTRAN coding, however: hash-functions and -arrays, and CLISP features, as the record data type, for example.

The INTERLISP functions not implemented in LISP-SP are marked in the comparative function list, appendix 4.

Appendix 1 : System Global Variables

COMMON SWP

The variables belonging to this COMMON-block carry information about the swapper.

NSWBLK	--	number of the swap-blocks
NSWPBB	--	number of the swap-buffer-begin
NSWPBE	--	number of the swap-buffer-end
NSWPBW	--	number of the swap-buffer-words
NSWPDI	--	number of the swap-directory-words
NSWPTB	--	number of the swap-table-words
NSWPTE	--	number of the swap-table-entries
NBPREC	--	number of bytes per record
NDIRSC	--	number of directory-sectors
NDSKDI	--	number of disk-directory-words
DSKDIR	--	disk-directory
SWPDIR	--	swap-directory
SWPTBL	--	swap-table

COMMON CHARS

The variables belonging to this COMMON-block carry the single 24 character which the interpreter used.

SPACE	--	blank	--	" "
LPAR	--	left parenthesis	--	"("
RPAR	--	right parenthesis	--	")"
ILBCHR	--	left bracket character	--	"<"
IRBCHR	--	right bracket character	--	">"
STRCHR	--	string character	--	"""
IQCHR	--	quote character	--	","
UBR	--	user break	--	"#"
DOT	--	dot character	--	"."
ITCHR	--	letter character	--	"T"
IPLUS	--	plus-sign	--	"-"
IMINUS	--	minus-sign	--	"+"
IFIG	--	array for the digits	--	"0" .. "9"
ATEND	--	escape character	--	"{"
SOFTBR	--	input-break character	--	"}"
CHTAB	--	character-table		

COMMON A

BYTES	--	number of BYTES per word
LUNROL	--	logical channel number for the rollin/rollout
LUNSWP	--	logical channel number for the swapper
MAXINT	--	limit of same do-loops
CHDIV	--	using for shift a byte (2**24)

In ROLLIN and ROLLOU the following 6 variables are equivalenced with the array COMA. They contain all the dynamic pointers.

NATOMP	--	dynamic pointer of PNP
NLISTP	--	dynamic pointer of LIST
NARRYP	--	dynamic pointer of PNAME (array-part)
JBP	--	byte-pointer of PNAME (litatom-part)
NUMBP	--	dynamic pointer of PNAME (number-part)
VNAMSP	--	pointer of PNAME (variable-names)

NSMIN	--	upper limit for small integers
NPNP	--	size of PNP
NLIST	--	size of LIST
NPNAME	--	size of PNAME
NSTACK	--	size of STACK
NHTAB	--	size of HTAB
NATHSH	--	using for compute the hashaddress
NBYTES	--	size of a physical record
MAXREC	--	size of a logical record
NCHTYP	--	number of different character-types
NRMESS	--	number of messages
MAXMES	--	maximal number of messages

For each interpreter-run the number of the different garbage-types are counted. The follwing 5 variables store the number of garbage-calls for type:

GARBS	--	5
LAGARBS	--	1
PNGARB	--	2
AAGARB	--	3
LIGARB	--	4

COMMON B

The next 10 variables contain pointers actually worked on that means all information for the garbage collector which are needed for not destroying some still used items.

```
ARG1      --  first  element of the actual expression
ARG2      --  second element of the actual expression
ARG3      --  third  element of the actual expression
ALIST     --  local variable-list
FORM      --  start of the actual list
TEMP1     --  temporal variable for storing
TEMP2     --  temporal variable for storing
TEMP3     --  temporal variable for storing
I1CONS    --  start of a new created list
I2CONS    --  next item to be put into the array LIST
NARGS     --  number of the above variables
```

The following variables contain lisp-litatoms which are needed by the interpreter.

```
NIL       --  NIL
ERROR     --  SYSERROR
PROG      --  PROG
LAMBDA    --  LAMBDA
FUNARG    --  FUNARG
EXPR      --  SUBR
FEXPR     --  FSUBR
T         --  T
GENNUM    --  GENNUM
UNUSED    --  NOBIND
QUOTE     --  QUOTE
A000      --  A
LISPX     --  LISPX
EVAL      --  EVAL
APPLY     --  APPLY
RSTATE    --  RANDSTATE
UNBOUN    --  NOBIND
STRING    --  STRING
FNCELL    --  FNCELL
BTRACE    --  *BACKTRACEFLG
BLIST     --  *BACKTRACE
NLAMBD    --  NLAMBDA
SUBTSTR   --  SUBSTR
```

The following variables contains the number of built-in functions.

```
SUBR0     --  with 0 argument
SUBR11    --  with 0 and 1 argument (only for numerical functions)
```

```
SUBR1       --  with 0 and 1 argument
SUBR2       --  with 0 and 1 and 2 arguments
SUBR3       --  with 0 and 1 and 2 and 3 arguments
SUBR        --  with 0 and 1 and 2 and 3 and n arguments
FSUBR       --  number of all built-in functions

IP          --  dynamical pointer of the function-part of STACK
JP          --  dynamical pointer of the argument-part of STACK
IPP         --  dynamical pointer to the function-code of PROG-values
JPP         --  dynamical pointer to the argument-part of PROG-values
MIDDL       --  allocation of the array STACK
ABUP1       --  dynamical pointer of ABUFF
CHT         --  character-type
CHR         --  character
ASA         --  carriage control character
IUNUTS      --  storing logical channel number for input
LUNINS      --  storing logical channel number for output
```

The following 10 variables are equivalenced with the array containing the i/o-values which are changeable by the user.

```
LUNIN       --  logical channel number for input
RDPOS       --  dynamical pointer in RDBUFF, reader position
LMARGR      --  left margin for input
MARGR       --  right margin for input
LUNUT       --  logical channel number for output
PRTPOS      --  dynamical pointer in PRBUFF, printer position
LMARG       --  left margin for output
MARG        --  right margin for output
LEVELL      --  maximal number of top level elements to be printed
LEVELP      --  maximal bracket-depth for printing

LUNSYS      --  logical channel number for
NSYM        --  number of symbols
BRFLG       --  bracket flag
BRLEV       --  bracket level
IFLAG       --  flag for printing
IBREAK      --  input brak
MAXLUN      --  maximal number of logical channel number
IOBUFF      --  size of PRBUFF, RDBUFF and ABUFF
ERRTYP      --  actual error-code
BFLG        --  backtrace-flag
DREG        --  buffer for system-flags
                DREG(1)  --  garbage collector messages printing
                DREG(2)  --  pretty printing?
                DREG(3)  --  print stack-contents?
                DREG(4)  --  unused
                DREG(5)  --  escape- and string-character printing?
                DREG(6)  --  unused
                DREG(7)  --  start a new line at each occurence of
                             a left parentheses
```

The following variables are the "big arrays":

ABUFF	--	storing i/o-data
PRBUFF	--	printer-buffer
RDBUFF	--	reader-buffer
IMESS	--	message-buffer
CAR	--	array containing the car of a litatom
CDR	--	array containing the cdr of a litatom
PNP	--	array containing pointer to litatom-printnames
HTAB	--	hashtable
STACK	--	stack
PNAME	--	array containing printnames
RPNAME	--	equivalenced with PNAME
SWPBUF	--	swap-buffer
LIST	--	array for storing lists
VNAMS	--	array containing the global variables

Appendix 2: FORTRAN Elements

1 File INIT

Contents:

PROGRAM INI

SUBROUTINE INIT1

SUBROUTINE INIT2

BLOCK DATA

1.1 PROGRAM INI

The main program is very short. It just calls the initialization subroutines, the interpreter LISPSP and the exit routine LSPEX. The LISP system can be initialized in two different ways:

a) After calling INIT1, the main program invokes INIT2 which creates a 'fresh' system by reading in the atoms-file. Only the FORTRAN defined functions will be available in such a system. This kind of initialization has to be performed at least once to create a (minimum) environment for the second initialization method.

b) After calling INIT1, the interpreter calls the function ROLLIN, which normally provides a more complete environment including FORTRAN and LISP defined functions (expr, fexpr) by reading in a binary image of previously defined functions. If selecting this kind of start, a 'rollout' file must be available, from which the interpreter can read the image (via logical channel LUNROL). Rollout-files are created by

- system initialization type a)
- (optionally) defining more functions (expr/fexpr)
- executing the rollout function (ROLLOUT 'LUNROL')

Before generating the Lisp system from source code, the main program should be modified appropriately to provide the desired environment.

1.2 SUBROUTINE INIT1

This subroutine performs all mandatory variable initializations. Variables affected include:

a) array limits
b) logical channel numbers
c) system dependant variables
d) variables used by the function ROLLIN
e) variables not affected by ROLLIN

1.3 SUBROUTINE INIT2

This initialization-routine provides the subr/fsubr-environment neccessary to create more 'intelligent' LISP systems. This is accomplished by:

a) initializing all dynamic pointers to internal structures
b) reading in the set of predifined symbols and atoms
c) initializing the swapper data structures
d) reading in the subr/fsubr-names and assigning function numbers
e) assigning some lisp-names, e.g. LAMBDA, FNCELL and T

1.4 BLOCK DATA

The BLOCK DATA segment contains some compile-time array initializations.

2 File LISPSP

contents:

SUBROUTINE LISPSP

2.1 SUBROUTINE LISPSP

Parameters are:

IREE -- interpreter entry code

IREE = 1 Start a new run and print some messages, e.g., the user space available for different data types.

IREE > 1 Interpreter restart - no messages printed

This subroutine is the interpreter which analyzes and executes the user input. The steps involved in interpreting LISP programs are explained in the following sections. First, a list is created internally which would read as:

```
(LISPX)
```

The interpreter then executes the LISPX subr which functionally resembles the LISP program

```
(PROG NIL
     LOOP (PRINT (EVAL (READ)))
          (GO LOOP)).
```

This LISP form describes the whole principle of the interpreter. The overall evaluation algorithm will now be illustrated by an example. Initially, the interpreter sets the following variables:

```
FORM   = (LISPX)
ARG1   = LISPX
L      = LISPX
ARG2   = NIL
EVALSW = NIL
```

It then jumps to the function-analyzing part.

Here it is determined, whether it's a built-in (subr/fsubr) or a user defined (expr/fexpr) function. This is done by using the function GET with the actual parameters LISPX and FNCELL (FunctioN CELL). If GET returns NIL, then there is no indicator FNCELL in the property list of the atom LISPX and hence, LISPX is a FORTRAN defined function.

Next, it is determined, whether this function is a subr or a fsubr. In this case it is a subr.

The evaluation-flag is NIL which means the arguments have to be evaluated first. In this case there are no arguments (ARG2 is NIL) - evaluation can proceed.

The next step is to determine the FORTRAN subroutine (ISUBRx), which contains the definition of LISPX. LISPX has no arguments and therefore, the subroutine ISUBRØ will be called. Within ISUBRØ the code for LISPX is executed, and ISUBRØ returns with exit code 3 - this means execution of the read-section (call the input-routine) is requested.

Now the input-section reads the next lisp-expression and returns the starting address of the expression.

Let the input be:

```
(MINUS 3)
```

The important variable-values then are:

```
ARG1  = (MINUS 3)
FORM  = (LISPX)
```

When starting the evaluation of the new expression, the old expression will be saved. This is done in the following way:

a) Push 'end-of-evaluation' indicator on function stack.

b) Push contents of FORM (last expression) onto argument stack.

Then, the following variables are set:

```
FORM = (MINUS 3)
ARG1 = (MINUS 3)
```

Now ARG1 is analyzed:

a) ARG1 is a litatom. Fetch the binding of the atom. If there is an ALIST (association list), search for this atom. If it is present, fetch its binding, otherwise fetch the binding from the cell of the array CAR corresponding to the litatom. If the value of that cell is equal to UNBOUN, an error has occured - unbound variable. The interpreter will jump to the error handling section.

After successfully retrieving a binding, a value is popped from the function stack indicating the next function to be executed.

b) ARG1 is neither a litatom nor a list. A value is popped from the function stack indicating the next function to be executed.

c) ARG1 is a list. This is true in the above example - the interpreter's action will be explained in more detail.

The following variables are set by 'parsing' the list:

```
ARG1 = MINUS
ARG2 = 3
```

Next, the function type (FORTRAN or LISP) is determined: the function GET returns an address that represents a user defined function. In this case it hands back the definition for the function MINUS:

```
(LAMBDA (X)
        (DIFFERENCE Ø X))
```

The interpreter now acts, as if the input has been

```
((LAMBDA (X)
         (DIFFERENCE Ø X))
  3)
```

The next step is the binding of the LAMBDA-variable X to the value 3, therefore control is transferred to the lambda-section of the interpreter.

There, the beginning of the list of lambda-variables is saved on the argument stack and their bindings are fetched. While nlambda-bindings are passed directly to the calling function, lambda-bindings have to be evaluated before executing the function. The example involves lambda-bindings, therefore all important values are stacked, and the lambda-binding is treated as new input.

For numbers, it is easy to find out, what the interpreter does. Finally, the function-code is popped which brings the interpreter back to the LAMBDA-part. There, the result will be stored in a new list, and it is tested, if there are more bindings.

In the example, all bindings are evaluated. Next, the type of parameter-argument-association is determined:

```
a) new ALIST = ((X . 1)(Y . 2)       ..    . old ALIST)
   (spread)

b) new ALIST = ((X . 1) .. (Y . (2 3 4)  . old ALIST)
   (half-spread)

c) new ALIST = (X . (1 2 3)              . old ALIST)
   (no-spread)
```

The example will produce the simple case:

```
new ALIST = ((X . 3) . old ALIST)
```

After setting up the ALIST, the interpreter must prepare for more than one expression in the body of the lambda-expression. This is done by pushing an appropriate code on the function stack.

Now, FORM and ARG1 are set:

FORM = (DIFFERENCE Ø X)
ARG1 = (DIFFERENCE Ø X)

The interpreter starts eavluation of the lambda body. First, the function DIFFERENCE will be analyzed - the function GET returns NIL - DIFFERENCE is a FORTRAN defined function. The evaluation-flag is set, so the interpreter jumps to the section testing for arguments.

The current pointer of the argument-stack will be stored into the function-stack, because all following items in the argument-part belong to the actual function. Next, an end-of-evaluation indicator is pushed to the function stack.

Now, a pointer to the list cell containing the next argument (X) and a pointer to the start of this function will be pushed on the argument stack.

Since ARG1 is Ø, no evaluation has to be done - it's a small integer. The last function code stored on the stack is popped.

The next argument is popped from the stack and the value just computed is pushed into the stack. The argument is not the end of a list but a real number (X), so it is also pushed on the stack.

ARG1 - pointing to the LIST-cell containing X - will now be 3. After fetching the value for X out of the ALIST, the function-code just stored is popped.

The next argument will be fetched from the stack, and the value just computed will be saved. In this case there are no more arguments and the interpreter fetches the first argument-address from the function-part of the stack.

FORM still contains the pointer to the beginning of the function. Using this fact, the number of the function to be executed is determined, defining also the subroutine which has to be called, ISUBR2 in this case. After executing the code for DIFFERENCE, ISUBR2 returns the value -3 in ARG1 (small integer). The interpreter now pops the next function code from the stack.

This number drives the interpreter to look, whether there is more than one statement in the LAMBDA-expression. This example involves only one, and therefore the ALIST has to be changed into its old value.

The next function-code makes the interpreter jump to the output-routines. At this time FORM points to the old LIST-cell, containing (LISPX).

Now the next input is requested. The interpreter analyzes the input. It searches for the first pair of parentheses (expression) which can be interpreted. The next function code is popped indicating the action to be taken.

Also the embedding lisp-statement is fetched from the argument stack. The result of the statement just computed now replaces the original function.

A number of statements in LISPSP doesn´t belong to the interpreter directly. Instead, they are - functionally - part of the ISUBR´s. Sometimes however, it is impossible to execute a lisp-function in some ISUBR internally, but execution involves multiple calls to the subroutine with interleaving interpreter action. This is true for the MAP functions, as an example.

The sections of the interpreter dedicated to execute those functions can easily be identified in LISPSP. They are reached only by executing some ISUBR function and are treated, as if they were part of some ISUBR.

To obtain a general understanding of the interpreter´s operation, it is not neccessary to investigate these sections.

3 File ISUBRØ

contents:

SUBROUTINE ISUBRØ

3.1 SUBROUTINE ISUBRØ

Parameters are:

L	--	number of function
JUMP	--	exit-code
JUMP = 1		start at the beginning, reset the system
JUMP = 2		execution of the function has been finished, continue with execution of the next or embedding function.
JUMP = 3		read next item
JUMP = 4		error has occured, branch to error section.

ISUBRØ contains the FORTRAN code for the LISP functions with no parameters. The function number is used to select the appropriate section via a computed GOTO. Return is normally directly from the selected section, except, if an error occured.

The actions involved in function execution are best understood by reading the source code directly. ARG1 returns the result of the function.

4 File ISUBR1

contents:

SUBROUTINE ISUBR1

4.1 SUBROUTINE ISUBR1

Parameters are:

L	--	number of function
JUMP	--	exit-code
JUMP = 1		start at the beginning, reset the system
JUMP = 2		execution of the function has been finished, continue with execution of the next or embedding function.
JUMP = 3		evaluation of the argument requested
JUMP = 4		evaluation of the argument-list requested
JUMP = 5		error has occured, branch to error section

ISUBR1 contains the FORTRAN code for the LISP functions with one parameter which is passed in ARG1. The function number is used to select the appropriate section via a computed GOTO. Return is normally directly from the selected section, except, if an error occured.

The actions involved in function execution are best understood by reading the source code directly. ARG1 returns the result of the function.

5 File ISUBR2

contents:

SUBROUTINE ISUBR2

5.1 SUBROUTINE ISUBR2

Parameters are:

L	--	number of function
JUMP	--	exit-code
JUMP = 1		execution of the function has been finished, continue with execution of the next or embedding function.
JUMP = 2		make a dotted pair of ARG1 and ARG2 which will be interpreted as a function
JUMP = 3		evaluation of the argument requested
JUMP = 4		ALIST has been destroyed
JUMP = 5		error has occured, branch to error section

ISUBR2 contains the FORTRAN code for the LISP functions with two parameters, passed in ARG1 and ARG2. The function number is used to select the appropriate section via a computed GOTO. Return is normally directly from the selected section, except, if an error occured.

The actions involved in function execution are best understood by reading the source code directly. ARG1 returns the result of the function.

6 File ISUBR3

contents:

SUBROUTINE ISUBR3

6.1 SUBROUTINE ISUBR3

Parameters are:

L	--	number of function
JUMP	--	exit-code
JUMP = 1		start at the beginning, reset the system
JUMP = 2		execution of the function has been finished, continue with execution of the next or embedding function.
JUMP = 3		make a dotted pair of ARG1 and ARG2
JUMP = 4		error has occured, branch to error section

ISUBR3 contains the FORTRAN code for the LISP functions with three parameters, passed in ARG1, ARG2 and ARG3. The function number is used to select the appropriate section via a computed GOTO. Return is normally directly from the selected section, except, if an error occured.

The actions involved in function execution are best understood by reading the source code directly. ARG1 returns the result of the function.

7 File ISUBRN

contents:

SUBROUTINE ISUBRN

7.1 SUBROUTINE ISUBRN

Parameters are:

I	--	number of function
JUMP	--	exit-code
JUMP = 1		execution of the function has been finished, continue with execution of the next or embedding function.
JUMP = 2		error has occured, branch to error section
EJP	--	stack pointer to the first argument
IARGS	--	number of arguments

ISUBRN contains the FORTRAN code for the LISP functions with a variable number of parameters, passed in the stack. The function number is used to select the appropriate section via a computed GOTO. Return is normally directly from the selected section, except, if an error occured.

The actions involved in function execution are best understood by reading the source code directly. ARG1 returns the result of the function.

8 File IFSUBR

contents:

SUBROUTINE IFSUBR

8.1 SUBROUTINE IFSUBR

Parameters are:

I	--	function number or return code for repeated function call
JUMP	--	exit-code
JUMP = 1		start at the beginning, reset the system
JUMP = 2		execution of the function has been finished, continue with execution of the next or embedding function.
JUMP = 3		save the function-code and evaluate the argument
JUMP = 4		evaluate the argument
JUMP = 5		jump to the condition-part in LISPSP
JUMP = 6		fetch the next function
JUMP = 7		ALIST has been destroyed
JUMP = 8		error has occured, branch to error section

IFSUBR contains the FORTRAN code for the LISP functions with non-evaluated parameters, ARG2 pointing to the beginning of the argument list. The function number is used to select the appropriate section via a computed GOTO. Return is normally directly from the selected section, except, if an error occured.

The actions involved in function execution are best understood by reading the source code directly. ARG1 returns the result of the function.

9 File INFN

contents:

INTEGER FUNCTION NEXT

SUBROUTINE CONS

INTEGER FUNCTION EQUAL

INTEGER FUNCTION NCHARS

INTEGER FUNCTION LENGTH

INTEGER FUNCTION COMPPN

9.1 INTEGER FUNCTION NEXT

Parameters are:

IARG -- pointer to some LISP object

NEXT returns the address of the next top level element of the list pointed to by IARG.

NEXT determines the type of the structure pointed to by IARG by calling GETEL. If GETEL returns a value in the range [1,2,3,4,5,7], then IARG is not a pointer to a list, and NEXT returns with value NIL.

If GETEL returns 6, then IARG contains a pointer to a list. Pointer-to-pointer references are resolved by GETARG, then the first element of the referenced list is examined.

If it is a left parenthesis, then the corresponding right parenthesis is searched, and NEXT returns the pointer to this LIST cell, incremented by 1.

If it does not contain a left parenthesis, then NEXT returns the pointer to this cell, incremented by 1.

If the type of IARG is determined to be 8 (by GETEL), then IARG contains a 'continuation-marker' - the list actually continues on some other LIST cell. This reference is resolved resulting in a pointer of type 6, so processing continues, as if IARG contained a pointer of type 6.

9.2 SUBROUTINE CONS

Parameters are:

I1 -- pointer to some LISP structure

CONS puts the contents of parameter I1 into the LIST cell pointed to by NLISTP after incrementing. If the LIST space is exhausted, then garbage collection is initiated.

9.3 INTEGER FUNCTION EQUAL

Parameters are:

I -- pointer to some LISP structure

J -- pointer to some LISP structure

EQUAL compares the two expressions pointed to by I and J on structural equality. It happens by analyzing each used LIST cell of I and J. This involves several steps:

1) First the LIST cells will be fetched which have to be examine next. Then the number of top level elements of the expressions pointed to by I and J are calculated (call to function LENGTH). If they are not the same, goto step 4.

2) Analyze the expression-types and, in case of equality, jump to the section corresponding to that type, described in 3). If they are different, see whether one or both contains a dot followed by a left parenthesis {note: (A . (B)) = (A B)}, in this case continue at 1), otherwise goto 4).

3) Type dependant actions

3.1) LITATOM
They are compared by using the integer function COMPPN. If the result is zero goto 1), otherwise goto 4).

3.2) NUMBER
Fetch the numbers. If they are equal goto 1), otherwise goto 4).

3.3) LEFT PARENTHESIS
There are sublists which also have to be analyzed. Save their addresses and start the comparison with their first top level element: goto 1).

3.4) RIGHT PARENTHESES
Fetch the address of the sublist just analyzed and get the next top level element of the embedding list with the help of the integer function NEXT, then goto 1).

3.5) ARRAY-POINTER
If they are equal, goto 1), otherwise goto 4).

4. The expressions pointed to by I and J aren't equal, therefore EQUAL returns with value NIL.

At the end of each type dependant section it is checked, if more has to be compared. If there is, then continue with 1), otherwise EQUAL returns with value T.

9.4 INTEGER FUNCTION NCHARS

Parameters are:

INPUT	-- pointer to some LISP structure
IFLG	-- a flag

NCHARS returns the number of characters in the PRIN1-printname of the expression pointed to by INPUT.

At first the type of the expression pointed to by input has to be analyzed by using the integer function GETARG. NCHARS is initially set to zero, and for each item in the (possibly recursive) evaluation it's length is added to NCHARS. Depending on the type of the expression, the following action is taken:

TYPE 1

The item is a literal atom, string or substring. The integer function GETPN returns the number of characters of this item in the variable IPL. If it is a string or substring, GETPN returns 1, and, if the flag isn't equal to NIL, the length of this item is larger than 2. Return NCHARS + IPL [+2].

TYPE 2

Item is a dot. Return NCHARS + 1.

TYPE 3

Item is a number. Depending on the type of the number (integer or real), return NCHARS +

- for the integer number, the number of significant characters

- for a real number,

```
  [ 1       for sign ]
    + number of digits of mantissa
    + 1     for radix point
    + 7     for digits of fraction
    + 2     for exponent
  [ + 1     for a larger exponent ]
  [ + 1     for a negative exponent ]
```

TYPE 4

Item is a left parenthesis. Return NCHARS +1.

TYPE 5

Item is right parenthesis. If necessary, fetch the address of the next item to be analyzed. Return NCHARS + 1.

TYPE 6

Item is a list-pointer. The pointer is saved in the stack, and the list-elements are analyzed one-by-one. For each type, the actions described above and below are taken. The lengths of the elements are added. Note that the elements are separated by blanks in their printnames, so for each element, 1 is added.

TYPE 7

Item is an array-pointer. The printnames of arrays are always of length 9, therefore return NCHARS + 9.

9.5 INTEGER FUNCTION LENGTH

Parameters are:

I -- pointer to a list

LENGTH returns as value the number of top level elements of the list pointed to by I. Note that lists of the form

(A . (B C))

are the same as

(A B C)

and for both types of lists, LENGTH will return 3.

9.6 INTEGER FUNCTION COMPPN

Parameters are:

I -- pointer to a printname

J -- pointer to a printname

COMPPN compares the printnames of I and J, on alphabetic order. It returns the following values:

```
 0   ===>   I = J
-1   ===>   I < J
 1   ===>   I > J
-2   ===>   I is illegal
 2   ===>   J is illegal
```

Using GETPN the length and the byte address in PNAME will be fetched. Then the lengths are tested. If both printnames are of length 0, then COMPPN returns 0. If one printname is of length 0 and the other is >0, then the latter is larger than the first, and the appropriate code is returned.

If both printnames are of length >0, then they are compared bytewise using the normal string comparison algorithm, and COMPPN returns the appropriate result.

10 File GTFN

contents:

INTEGER FUNCTION GETPN

SUBROUTINE GETNUM

INTEGER FUNCTION GETEL

INTEGER FUNCTION GETARG

INTEGER FUNCTION GET

10.1 INTEGER FUNCTION GETPN

Parameters are:

I	--	pointer to a litatom/string/substring
MAIN	--	returns the address of the main-string
JB	--	returns the byte-address of string I
IPL	--	returns the length of I

GETPN returns the printname of a string or substring in the parameters described above.

First, GETPN fetches the byte address and length of the printname from PNP. Then, CAR(I) is tested. If it does not contain the value SUBSTRING, then GETPN returns.

Substrings are stored in the following way: CDR(I) points a list with the following structure:

(MAIN JB1 . IPL)

where JB1 is the address of the first substring-byte of the main string. If CAR(I) does contain SUBSTRING, then the list pointed to by I is checked for the above structure. After verifying it, the byte address and byte count are passed back in JB and IPL.

GETPN returns:

-1	I does not point to a litatom/string/substring
0	I points to a litatom
1	I points to a string or a substring

10.2 SUBROUTINE GETNUM

Parameters are:

- I -- pointer to a number, on exit contains the integer
- R -- returns a real number
- L -- indicates the type of the number returned

GETNUM returns the number pointed to by I in I or R, depending on the type of the number. The parameter L indicates the type: it is set to 'true' for a real, and 'false' for an integer number.

Numbers are decoded in the following way:

Integer Numbers

If the first bit of I is off (zero), then I points to an integer. In this case,

I - NLIST - NPNP

is checked on less than NPNAME. If it is, then it is taken as an address to PNAME, and the value of the integer is fetched from the corresponding PNAME cell. If it is larger then NPNAME, then it is a small integer, defined as integer in the range of [-2000 .. 2000]. The number in this case is calculated as

I - NPNAME - NLIST - 2000

In both cases, the result is returned in I, and L is set to false.

Real Numbers

If the first bit in I is on (one), then the number is a real. The value I - NLIST - NPNP is used as an index to PNAME to fetch the real number which is returned in R. L is set to 'true' to indicate the return of a real number.

10.3 INTEGER FUNCTION GETEL

Parameters are:

I -- pointer to some LISP structure

GETEL determines the type of the structure pointed to by I. Each type is assigned a subrange of the set of integers in the interval [0 .. 2^32-1]. Since numbers with the first bit set to one are normally treated as negative numbers by FORTRAN systems, this fact is used to recognize real numbers. The following tables show the distribution of data types on the above interval.

```
***********************************************************************
*                                                                     *
*                                                                     *
*   a-----b-----c-----d-----e-----f-----g-----h-------------->        *
*   |     |     |     |     |     |     |     --> real numbers        *
*   |     |     |     |     |     |     --> array pointer             *
*   |     |     |     |     |     --> special markers: [ (,),. ]      *
*   |     |     |     |     --> small integers                        *
*   |     |     |     --> continuation markers                        *
*   |     |     --> integer numbers                                   *
*   |     --> pointers to LIST                                        *
*   --> strings / litatoms                                            *
*                                                                     *
*                                                                     *
*           a       0                                                 *
*           b       NPNP                                              *
*           c       NPNP + NLIST                                      *
*           d       NPNP + NLIST + NPNAME                             *
*           e       NPNP + NLIST + NPNAME + NLIST                     *
*           f       #20000000   [ '(   ',')   ','.   ' ]              *
*           g       #40000000                                         *
*           h       #80000000                                         *
*                                                                     *
***********************************************************************
```

Table 10.3-1 LISP Data Types
Distribution on the interval [0 .. 2^32-1]

Each data type is assigned a number by GETEL as shown in table 10.3-2.

1	litatom or string
2	dot
3	number
4	left parenthesis
5	right parenthesis
6	list pointer
7	array pointer
8	continuation marker

Table 10.3-2 Data Type Identification

10.4 INTEGER FUNCTION GETARG

Parameters are:

I -- pointer to be analyzed

TYP -- returns GETEL(I)

GETARG is the function used to resolve pointer-to-pointer references within lists, whenever the interpreter needs to determine the type of a substructure in the LIST array.

If a continuation marker has been found in following the pointer chain, I will be changed to point to the LIST cell pointed to by the continuation marker.

If GETARG finally determines that I points to a sublist (a structure starting with a left parenthesis), then the parameter TYP is set to 6, and GETARG returns the direct pointer to the left parenthesis.

If I does not finally point to a sublist, then TYP is set to the type of the structure found, and GETARG returns that item.

10.5 INTEGER FUNCTION GET

Parameters are:

HEAD -- pointer to a litatom

TAIL -- pointer to a litatom

GET returns the value of the property TAIL on the property list of HEAD. If the litatom HEAD does not have a property-list, GET returns NIL.

Each property-list has the following form:

(indicator value indicator value ..)

Note that property list are normal lists and may contain pointer-to-pointer references and continuation markers.

11 File DEBU

contents:

SUBROUTINE DEBUG

INTEGER FUNCTION POS

INTEGER FUNCTION GETSYM

11.1 SUBROUTINE DEBUG

DEBUG is a debugging tool giving symbolic access to the systems global variables collected in the various COMMON areas.

When the symbol "\" has been found in column 1 of an input record, DEBUG is called. It then reads command lines from logical channel LUNIN, and output is written to channel LUNUT. The comman line syntax is as follows:

```
global-variable   [format]   [first array-cell]
                             [last  array-cell]
```

Each of these input data will be analyzed one after the other by using GETSYM. The integer function POS returns the number of the variable in the lisp-array VNAME which is stored in PNAME.

By the aid of this number the debugger jumps to a statement, which calls the subroutine OUTPUT with the name of this variable.

Exit from DEBUG is done by entering "\" in column 1 of an input line.

11.2 INTEGER FUNCTION POS

Parameters are:

SYMBOL -- global variable name

TABADR -- table address

The result of this function is the position of the global variable in the array VNAME.

11.3 INTEGER FUNCTION GETSYM

Parameters are:

LINE	--	buffer with the input data
C	--	current position of LINE
SYMBOL	--	variable for storing the symbol
NUMBER	--	variable for storing the number
L	--	length of the string

GETSYM scans the input line for the next symbol. If it is a number, then it is stored into NUMBER, and GETSYM returns 2.

If it is a string, then it is stored into the variable SYMBOL and GETSYM returns 1.

12 File DOUT

contents:

SUBROUTINE OUTPUT

INTEGER FUNCTION GETTYP

12.1 SUBROUTINE OUTPUT

Parameters are:

FIELD	--	array which elements have to be printed
FIRST	--	first to print element of FIELD
LAST	--	last to print element of FIELD
TYPE	--	format for printing

OUTPUT prints the value of the variable specified in the DEBUG command line in the specified format. Depending on the format the maximal number of printing elements is:

R8	--	60
R4	--	100
Z8	--	160
Z4	--	320
I4	--	100
I2	--	160
A4	--	160
A2	--	320
A1	--	160

If the desired number of values is greater than the maximal number, the debugger will print only the format depending number.

This routine does not check variable types and bounds. It just treats the variable as an array and prints all elements from FIRST to LAST.

12.2 INTEGER FUNCTION GETTYP

Parameters are:

SYMBOL -- input token describing the printing format

This function gives each format type a number. This number is returned to the calling program. If the format is illegal, zero will be returned.

13 File STCK

contents:

SUBROUTINE FPUSH

SUBROUTINE APUSH

SUBROUTINE APOP

13.1 SUBROUTINE FPUSH

Parameters are:

I -- item to be pushed on function stack

FPUSH is used to push one item on the function stack. The function stack is the upper part of the array STACK. If the stack is full, the code 21 is pushed instead of I.

13.2 SUBROUTINE APUSH

Parameters are:

I1 -- value which has to be pushed into stack

I2 -- value which can be pushed into stack

I3 -- value which can be pushed into stack

I -- number of values which must be pushed

This subroutine pushes values into the lower part of the array STACK. This part contains the arguments actual function. 1 - 3 values will be pushed into the array depending on I.

If the stack is filled up, the code 21 is pushed into the function stack.

13.3 SUBROUTINE APOP

Parameters are:

I1	--	first value from stack
I2	--	second value from stack
I3	--	third value from stack
I	--	number of items to be popped

The subroutine is the opposite of the subroutine APUSH. It works also on the lower part of the STACK and pops the values which have been stored using APUSH. 1 - 3 values will be popped depending on I.

If the upper part is empty, the code 22 will be stored in the function part of STACK.

14 File L7845

contents:

SUBROUTINE RDREC

SUBROUTINE WRREC

SUBROUTINE GETCH

SUBROUTINE PUTCH

14.1 SUBROUTINE RDREC

Parameters are:

LLUN	--	logical cannel number for the input
RECNO	--	record-number of the reading record
BUFADR	--	buffer for writing

This routine reads a record, whose record-number is given by RECNO, into the buffer BUFADR via logical channel LLUN. The read is handled by the operating system subroutine READ4.

14.2 SUBROUTINE WRREC

Parameters are:

LLUN	--	logical channel number for the output
RECNO	--	record-number of the record to be written
BUFADR	--	buffer fokr reading

This routine writes buffer BUFADR to the record, whose number is given by RECNO, via logical channel LLUN. The writing is handled by the operating system subroutine WRITE4.

14.3 SUBROUTINE GETCH

Parameters are:

LTEXT	--	array with the desired character
ICH	--	returns the required character
I	--	byte-number of the desired character in LTEXT

This routine fetches the character at the byte-number I from the buffer LTEXT into ICH. The last 3 bytes are filled up with blanks.

14.4 SUBROUTINE PUTCH

Parameters are:

LTEXT	--	array for storing the character
ICH	--	character to be stored (leftbound)
I	--	byte position

This routine stores the first character from ICH into the array ITEXT at the byte-number I.

15 File L8060

contents:

SUBROUTINE TIMDT4

LOGICAL FUNCTION TESTB

SUBROUTINE SETBT

SUBROUTINE SETBF

INTEGER FUNCTION ISHFT

SUBROUTINE RDREC

SUBROUTINE WRREC

SUBROUTINE GETCH

SUBROUTINE PUTCH

15.1 SUBROUTINE TIMDT4

Parameters are:

```
KCLOCK        --  this array returns:
                  KLOCK(1)  --  hours
                  KLOCK(2)  --  minutes
                  KLOCK(3)  --  seconds
                  KLOCK(4)  --  seconds / 200
                  KLOCK(5)  --  day
                  KLOCK(6)  --  month
                  KLOCK(7)  --  year
                  KLOCK(8)  --  200

KCC           --  dummy parameter for the AEG 8060
```

This routine converts the MAX/IV TIMDT4 built-in subroutine into MARTOS TIME and DATE calls.

15.2 LOGICAL FUNCTION TESTB

Parameters are:

I -- word which has to be tested

J -- bit-number, 1 .. 32

The MAX/IV bit test returns 'true', if the J-th bit of the word I is zero. The MARTOS starts at the other end of the word, therefore the bit 32-J has to be tested, and it returns 'false', if the bit is zero. These two differences have to be corrected.

15.3 SUBROUTINE SETBT

Parameters are:

I -- word to be affected

J -- bit to be affected

SETBT is a MAX/IV setting a bit in a word to logical 'true' which is implemented as Ø. Therefore, when running the system on the 8Ø-6Ø, this call has to be replaced by a call to the MARTOS function BCLR. Also, the bit number has to be reversed.

15.4 SUBROUTINE SETBF

Parameters are:

I -- word to be affected

J -- bit to be affected

SETBF is the MAX/IV subroutine to set a bit in a word to 'false' which is implemented as 1. When running the system on the 8Ø-6Ø, this call has to be converted into a call to the MARTOS function BSET. Also, the bit number has to be reversed.

15.5 INTEGER FUNCTION ISHFT

Parameters are:

I	--	word whose bits have to be shifted
J	--	number of bit-shifts: J > Ø ===> left shift J < Ø ===> right shift

The MAX/IV call has to be replaced by the MARTOS call ISHL.

15.6 SUBROUTINE RDREC

Parameters are:

ILUN	--	logical channel number for input
IREC	--	record-number for the record to be read
IBUF	--	buffer to store the input

This subroutine calls only the MARTOS built-in routine RDRW for reading a 256 byte record.

15.7 SUBROUTINE WRREC

Parameters are:

ILUN	--	logical channel number for the output
IRECC	-	record-number of the record to be written
IBUF	--	buffer for reading

This subroutine calls only the MARTOS built-in routine WRTRW for writing a 256 byte record.

15.8 SUBROUTINE GETCH

Parameters are:

LTEXT	--	array with the desired character
ICH	--	returns the required character
I	--	byte-number of the desired character in LTEXT

This routine fetches the character at the byte-number I from the buffer LTEXT into ICH. The last 3 bytes are filled up with blanks.

15.9 SUBROUTINE PUTCH

Parameters are:

LTEXT	--	array for storing the character
ICH	--	contains character to be stored (leftbound)
I	--	byte position

This routine stores the first character from ICH into the array LTEXT at the byte-number I.

16 File SWP1

contents:

INTEGER FUNCTION SWPIN

INTEGER FUNCTION SWPOUT

INTEGER FUNCTION MKSWAP

INTEGER FUNCTION UNSWAP

SUBROUTINE MVARRY

16.1 INTEGER FUNCTION SWPIN

Parameters are:

ARRAY -- pointer to header of array to be swapped in

If the array is not a swappable, then SWPIN returns with no other actions and value of SWPIN is ARRAY. If the array is swappable, then it is read into the swap buffer (if neccessary), and the value of SWPIN is the pointer to the first word of the array-header now in the swap buffer.

Since the swap buffer is part of PNAME, the array functions can access swappable arrays in the same way as non-swappable arrays after making sure by a call to SWPIN that the array is resident in memory.

16.2 INTEGER FUNCTION SWPOUT

Parameters are:

SWPARR -- pointer to first word of header of the array.

This routine will be called, whenever it is neccessary to remove a swappable array from the swap buffer. The value of SWPOUT is the number of 256-Byte blocks freed in the swap buffer.

If SWPOUT is called with SWPARR negative, then SWPARR contains the (negative) starting position of a sequence of free swap buffer blocks. In this case, SWPOUT returns the number of free blocks starting with block -SWPARR.

16.3 SUBROUTINE MKSWPA

Parameters are:

ARRAY -- pointer to first word of the array header

This function converts a non-swappable array into a swappable array. First, the size of the array is checked against the total size of the swap buffer to make sure,that it fits into the latter - if not, an error is generated, and the array remains unswappable.

Next, the function allocates disk space on the swap file - if not enough space is available, an error is generated, and the array remains unswappable.

Finally, the function allocates space in the swap buffer (other arrays may be swapped out), moves the array into the swap buffer, markes the array header and returns as value the original array pointer.

16.4 SUBROUTINE UNSWPA

Parameters are:

ARRAY -- pointer to first word of array header

After swapping in the array, space is allocated in PNAME to construct a non-swappable array (that may cause garbage collection!). The array contents are moved to the new array, including the header information. Then, the disk and swap buffer areas are cleared. UNSWPA returns as value the pointer to the new header.

16.5 SUBROUTINE MVARRY

Parameters are:

SWPTBP -- pointer to a swap table entry

ARRAY -- pointer to an array header

ATOB -- indicates direction ov move

Depending on the value of ATOB (true or false), the array is moved to/from swap buffer from/to array space in PNAME.

17 File SWP2

contents:

SUBROUTINE INISWP

INTEGER FUNCTION FBSPAC

INEGER FUNCTION FDSPAC

INTEGER FUNCTION FSWPTE

INTEGER FUNCTION ARRINX

SUBROUTINE SWPALL

INTEGER FUNCTION IRAND

17.1 SUBROUTINE INISWP

Parameters are:

RDBACK -- read back swap file directory or initialize

INISWP is called during system start up. Depending on the value of RDBACK, the swapper's disk file directory is initialized or read back from disk.

17.2 INTEGER FUNCTION FBSPAC

Parameters are:

IBLKS -- number of blocks to be allocated

This function allocates space in the swap buffer. If neccessary, it swaps out other arrays. Only as many arrays will be swapped out as needed to provide enough space. The position of the first array to be swapped out is determined by a random number.

The function returns as value the pointer to the swap table entry (allocated by a call to FSWPTE) which will be used to reference the swappable array, while it is swapped in.

17.4 INTEGER FUNCTION FDSPAC

Parameters are:

IBLKS -- number of 256 Byte blocks to be allocated

This function searches the swap file directory for array space. In contrast to swap buffer space allocation, only an error can be reported, if no more space is available. The function returns as value the number of the first record allocated on the disk.

The swap file must be of fixed size, since the first record(s) on disk contain the image of the swap file directory (which is actually a bit-map of free/allocated disk records).

17.5 INTEGER FUNCTION FSWPTE

Parameters are:

IDUMMY -- guess, what it does.

For all swappable arrays currently swapped in, the system needs an entry in the swap table (consisting of 4 words). If no entry is available, then the function selects one of the allocated entries by using a random number to determine a swap-out candidate. By swapping out that candidate, the entry is freed and can then be reused. the value of this function is the pointer to the swap table entry.

17.7 SUBROUTINE ARRINX

Parameters are:

IBLKNO -- index to swap buffer

This function determines the pointer to the swap table entry describing the swappable array which is currently in the swap buffer starting with block number IBLKNO. If block IBLKNO is currently not allocated, then the function returns -IBLKNO.

17.8 SUBROUTINE SWPALL

This subroutine searches the swap table for arrays currently swapped in and swaps them out. The swap table entries are freed. Finally, the swap file directory is written to disk to allow for proper re-initialization. This subroutine will be called during system shut-down and garbage collection.

17.9 INTEGER FUNCTION IRAND

Parameters are:

IX -- left interval margin

IY -- right interval margin

IRAND is used to determine random numbers for swapper internal use.

18 File INPT

contents:

INTEGER FUNCTION IREAD

INTEGER FUNCTION RATOM

SUBROUTINE SHIFT

INTEGER FUNCTION MATOM

INTEGER FUNCTION MKNUM

INTEGER FUNCTION MKARRY

INTEGER FUNCTION COPARR

18.1 INTEGER FUNCTION IREAD

Parameters are:

IDUMMY -- guess, what it does

Before starting a new input, a function-code is saved in the function stack.

IREAD reads a LISP expression from the actual input channel.

The function calls the routine RATOM which returns two different items:

a) separators (parentheses or blank)
b) atoms

If IREAD gets an atom, then this atom is stored, and it´s address is passed back to the calling program.

If the first non-blank input character is a left parenthesis or a left super bracket, the input is expected to be a list, and the list will be stored in the array LIST.

The process of creating a list is as follows:

a) actual character : LEFT PARENTHESIS

RATOM returns 1 and CHT the charactertype, CHT = 2. Using CONS a left parenthesIs will be put into the array LIST. The bracket level BRLEV is increased by 1.

b) actual character: RIGHT PARENTHESIS

RATOM returns 1 and CHT the charactertype, CHT = 3. Using CONS a right parenthesis will be put into the array LIST. The bracket level BRLEV is decreased by 1.

c) actual character: LEFT SUPER BRACKET

RATOM returns 1 and CHT the charactertype, CHT = 4. The bracket level and a code for the super bracket will be pushed on the stack, and bracket level BRLEV will be set to zero. Then IREAD acts, as if the item has been a left parenthesis.

d) actual character: RIGHT SUPER BRACKET

RATOM returns 1 and CHT the charactertype, CHT = 5. A flag (BRFLG) will be set indicating that all open parentheses have to be closed. This drives IREAD to call CONS repeatedly with parameter RPAR and decrement BRLEV by one until it finally becomes zero. Then, the last item pushed on the function stack is popped. Depending on it´s value, the following action occurs:

1 -- return with value address of expression read

2 -- the super bracket is already executed

3 -- take care of the function QUOTE:
'A is the same as (QUOTE A).
A right parenthesis has to be put into LIST.

4 -- take care of the construction " . (".
There is a right parenthesis in the input buffer which must not be CONS'ED into LIST.

e) atom

RATOM returns 2 and the address of the atom. Using CONS the address will be put into the array LIST. If the atom is quoted, then the LISP function QUOTE has to be closed with a right parenthesis.

18.2 INTEGER FUNCTION RATOM

Parameters are:

X	--	returns the address of the atom
IOP	--	a flag to indicate, who called
IOP = Ø		call from the LISP function RATOM
IOP = 1		internal call

RATOM calls the routine SHIFT to get the next character from the input buffer. Blanks are separators, they are treated as non-significant characters, except, if they occur within strings or are preceded by the escape character. Therefore the subroutine SHIFT will be called until there is a character, which is not equal to blank, CHT > 1.

If the character returned by SHIFT is a separator other than blank, RATOM returns it back to the calling program. In all other cases, it is a litatom, number or string. The following actions occur, when finding the corresponding characters:

a) actual character: "

The next item in the buffer is a string. The whole string will be read. If the string has more than 8Ø characters, it is greater than one input line, SHIFT returns the charactertype Ø. In this case SHIFT will be called with actual parameter 3 to drive it to store the 8Ø characters in PNAME before continuing reading. When finally the matching "-character is found, SHIFT will be called to complete the string. The address of the string will be returned to the caller.

b) actual character: ´

The next item in the buffer is a quoted item. Ratom has to create a list containing the function QUOTE.

```
Remember: ´A     = (QUOTE A)
          ´(A B) = (QUOTE (A B))
```

Using CONS a left parenthesis and the coded QUOTE will be put into LIST. The bracket level BRLEV and a function code will be pushed into the stack, and the bracket level will be set to 1. Then next item will be read.

c) actual character: user break

MATOM is called for making an atom of the user break character, and the address of the atom is returned to the calling program.

d) actual character: .

A decision is made, whether this character is a radix point, a litatom or the dot in a dotted pair. This can be done by analyzing the next character of the input buffer.

d.1) radix point
The next character has to be a digit; the radix-flag RFLG is set to T, and a jump to the atom section in RATOM is performed.

d.2) litatom
The next character is neither a digit nor a blank, set the number-flag NUMFLG to NIL and jump to the atom part of RATOM.

d.3) dot in a dotted pair
The next character is a blank. Take care of the construction:

```
(A . (B C)) = (A B C)
(A . <B (C D>) = (A B (C D))
```

The next non-blank character will be read. If it is a right parenthesis or a right super bracket, the dot doesn't belong to a dotted pair. The dot is returned as an atom to the caller.

If this character isn't a left parenthesis nor a left super bracket, the reader position RDPOS is decreased by one, and the dot is returned as an atom to the caller.

If it is a left super bracket, the bracket level BRLEV and a function code is pushed into the stack; BRLEV is set to zero, and RATOM acts, as if it was a left parenthesis: BRLEV and a function code is pushed on the stack, BRLEV is set to -1, and the next item is read.

e) actual character: + or -

A decision has to made, whether the character is part of an atom or a sign. The next character is fetched, and, if it is a digit, the number flag is set to T. Then a jump to the atom part is performed.

f) actual character: Ø..9

A number is created using the digit, and a jump to the atom part with number-flag NUMFLG equal to T is performed.

g) atom part

The next character is fetched, and the old character is saved in ABUFF.

g.1) If the character-type is less than 9, the whole expression is read.
For litatoms, MATOM is called for creating a literal atom, and its address is returned to the calling program.
For numbers, MKNUM is used to create the number, and it's address is returned to the calling program.

g.2) If the atom is a litatom (NUMFLG = NIL), the next character is fetched, and the old is saved in ABUFF, until CHT < 9.

g.3) If the character-type is equal to 9, the character "." has been read. If it is the first dot in this expression, and, if there hasn't been an exponent, the radix-flag RFLG is set 'true'. Otherwise, NUMFLG is set to NIL, and the input is taken as a litatom.

g.4) If the character "E" has been read, and, if the exponent-flag EFLG is .true., NUMFLG is set to NIL, and the input is taken as a litatom. Otherwise, the exponent- and the radix-flag are set to .true., and the digit is saved into R1SU. RSU is set zero, and the exponent is expected next.

g.5) If the EFLG is .true., the number of digits is incremented. If there are more than two digits, the number will be treated as a litatom, because it is out of range (now: NUMFLG = NIL).

g.6) If the next character is a digit, all digits are shifted by 1 place to the left, (multiply with 10), and the new digit is added. Then, the next character is read.

g.7) If none of the above cases occurs, then the expression is a litatom: NUMFLG is set to NIL, and the next character is read.

18.3 SUBROUTINE SHIFT

Parameters are:

I -- SHIFT control value

SHIFT supplies the caller with the next character and it's type from the input channel.

If the control parameter value is 1, then the old character is stored in ABUFF, and SHIFT continues, as if the value was 2.

If the control value is 2, then depending on IFLG2 the next actions are:

IFLG2 = T: SHIFT has to read from the printerbuffer PRBUFF, this is neccessary, e.g. in case the user has called the LISP function PACK. SHIFT sets CHR (the character itself) and the CHT (the character-type). The last character read is in PRBUFF(ARG2).

IFLG2 = NIL: The normal input buffer is used, the next character is fetched from the readerbuffer RDBUFF.

If I = 3, then some special action has to be performed. First, all characters in ABUFF are put into PNAME, and, in case the characters are part of a string, the string length is updated.

The normal input buffer is used (RDBUFF).

If RDPOS <= MARGR, the next character is fetched from RDBUFF and analyzed. If it is of type 1..22, CHT and CHR are set. If it is of type 23, the next character is fetched, and the type 10 is assigned to it. If it is of type 24, an input break is requested, so all neccessary variables are set.

If RDPOS > MARGR, RDA is called which fills up the readerbuffer RDBUFF with a new input line. The readerposition is set to the beginning and the charatertype to zero, so that the calling program knows about the new input. This is important for creating strings, because there the characters from the previous input line have to be stored into PNAME.

13.4 INTEGER FUNCTION MATOM

Parameters are:

LP -- MATOM control value

MATOM is the function to create atoms. If LP is negative, MATOM has to create a string, otherwise a litatom.

If LP is positive, the atom is searched in PNAME. First, the hashaddress of the litatom is determined. If the computed address of the hashtable HTAB contains UNBOUND, the atom isn't in PNAME.

If it is, the atoms are compared. If they are equal, the address is returned, otherwise the following atoms are compared until either the atom is found or the hashtable entry is equal to UNBOUND.

If the atom is not yet known to the system, it is examined, if there is enough space for the new entry. If this test is negative, the garbage collector is called.

After finally space is provided for the atom, the different array-cells belonging to this input are set. Also, the printname-pointers which contain the byte-address and the length of the token, are set. The CAR-cell is set to STRING or UNBOUN (marker for string or litatom), and the CDR-cell is set to NIL. Then, the characters are transferred from ABUFF to PNAME.

18.5 INTEGER FUNCTION MKNUM

Parameters are:

N	--	number to be stored (binary)
M	--	type of the number
M = 1		number is integer
M = 2		number is real
M = 3		number is integer, use spare PNAME space

MKNUM is the function to store 'make' numbers, i.e., encoding them and storing them into the appropriate space.

MKNUM allocates the next free PNAME-element, and, if necessary, calls the garbage collector.

If N contains a small integer, (range currently [-2000.. 2000]), the number is encoded by adding the length of array PNP, array LIST, array PNAME, array LIST and 2000 to the number itself.

Otherwise, the number is saved in PNAME and coded by adding NUMBP, NPNP and NLIST. If it is a real, bit 1 is set in the coded number.

This encoded number will be returned to the calling program.

18.5 INTEGER FUNCTION MKARRY

Parameters are:

S	--	size of the new array
P	--	size of the unboxed number part of S
V	--	value for initialization
INIT	--	initialization desired?

MKARRY is the function to create an array which may be requested explicitly by the LISP user (MKARRAY) or internally by the system (part of MKUNSWAP, no initialization).

First, using the arraysize parameter, it is checked, if there is enough space for the array. If there is not, then garbage collection is invoked.

If initialization of the array is desired, MKARRY builds up the header. The first element contains the arraysize, the second the size of the unboxed number part and the following two are set to zero. MKARRY then initializes each arrayelement in the unboxed number part with Ø and all other elements are set to V which may be any valid LISP pointer.

MKARRY returns the address of the first word of the array header, encoded by setting bit 2 of the address to 1.

18.6 INTEGER FUNCTION COPARR

Parameters are:

IARG	--	array pointer

COPARR is used to create an array as a copy of an existing one. This function is used whithin the LISP function COPYARRAY.

First, the header of array IARG is fetched and space is allocated in PNAME (which may cause garbage collection).

Then, a new header is created with the same contents as the header of IARG, and all elements are copied.

COPARR returns the address of the new array, encoded by setting the bit 2 of the real address to 1.

19 File OUTP

contents:

SUBROUTINE PRIN1

SUBROUTINE LSPEX

SUBROUTINE IPRINT

SUBROUTINE TERPRI

SUBROUTINE PRINAT

19.1 SUBROUTINE PRIN1

Parameters are:

S -- item to be printed

PRIN1 prints the LISP structure pointed to by S.

If S is not a list, PRIN1 calls only PRINAT and, in case of pretty printing (DREG(2) <,> NIL), also the routine TERPRI.

In case S is a list, PRIN1 looks at the LIST element physically following the item just printed. This must have one of the 8 types, returned by GETEL.

Before the type-depending actions are explained, it is usefull to know about the type independant actions which therefore will be explained first:

a) If the item to be printed does not fit into the actual print line, then TERPRI is called to print a line,contents of PRBUFF.

b) It is tested, if the number of top level elements of a list (LTOP) is greater than the desired number (LEVELL).

When a top level element is put into the printerbuffer, LTOP is increased by 1. If the next top level element is a left parenthesis, LTOP and a function code will be saved, and LTOP will be set to Ø.

If LTOP is greater than LEVELL, PRIN1 calls PRINAT for storing the characters "..." into PRBUFF and searches for the end of the list, the corresponding right parenthesis.

Then the right parenthesis will be stored into PRBUFF, and PRIN1 starts with the analyzing of the next element of the list.

c) If LDEPTH (the number of open parentheses) is greater than LEVELP (the desired „parentheses nesting level during print), the characters "---" are put into the printerbuffer, and analysis of the next top level element is started.

d) Pretty printing.

d.1) If DREG(2) is not set to NIL, pretty printing is requested. If a left parenthesis is the next element of the list, the number of open left parentheses will be fetched and saved into IREE. If IREE is greater than 4, super brackets will be printed instead of normal parenthesis.

If DREG(7) is equal to T, the number of characters to be printed is fetched. If this whole list can't be written into the printerbuffer, or, if DREG(7) isn't equal to T, some tests have to be done:

Has the left margin (LMARG) been changed? If so, then reset the left margin.

Has something been written into the printerbuffer? If so, TERPRI is called for printing.

Does the whole list fit into the actual line? If not, a function code, meaning the list, has to be pretty printed is stored.

Is the left margin greater than half an output line? If so, reset the margin as necessary.

Now all pretty printing information for this list is defined and pushed into the argument stack (3 cells). JP points to the last used cell:

```
STACK(JP)   = Ø   ===>   normal parenthesis
            = 1   ===>   super bracket

STACK(JP+1) = Ø   ===>   list fits on one line
            = 1   ===>   list doesn't fit on line

STACK(JP+2) = left margin
```

Now, the list containing this list is tested, if it has to be split up. If necessary, the left margin is reset.

d.2) There is one important variable by the pretty printing: II. At all times this variable contains the number of atoms stored into the printerbuffer. If II is greater than 1 and the actual list has to be split up, PRIN1 calls TERPRI. If II is equal to 1 and a new item is put into the printerbuffer, the left margin will be updated.

The different types will be handled in the following way:

a) TYPE 1

If the litatom is QUOTE, the last character in the printerbuffer PRBUFF is tested. If it is a left parenthesis, PRTPOS and LDEPTH are decreased by one. Then the number of top level elements is updated and a function code and the number of open parentheses are saved in the stack. Then, the charachter "´" is put into PRBUFF.

If it is a normal atom, PRINAT is called for printing the atom, and the next element of the list is analyzed.

b) TYPE 2

The symbol "." is put into the printerbuffer, and the next item of the list is fetched.

c.) TYPE 3

Numbers are treated in the same way as normal literal atoms.

d) TYPE 4

The list-element is a left parenthesei, so it is tested, whether the last character in PRBUFF is a DOT of a dotted pair. In this case, the printer position and the number of top level elements are decreased by 1, and the global level of parentheses GLLEV and a function code are saved into the stack. GLLEV is set to 1. Then the next element of the list is analyzed.

If the last character in PRBUFF is not a dot, the number of top level elements and a function code are saved into the stack, and LTOP is set to zero. Then it is tested, whether pretty printing is desired Depending on IRET, a left parenthesis or a left super bracket are put into PRBUFF.

e) TYPE 5

It is tested, whether this right parenthesis shouldn´t be stored into PRBUFF: (QUOTE A) --> ´A and (A . (B C)) --> (A B C). After pretty printing has been tested, either the parenthesis or the bracket are put into PRBUFF.

If GLLEV is equal to zero, a function code is popped out of the stack. In any case, printing is started with the first element of the list.

f) TYPE 6

The next element of the array LIST is a list-pointer. The number of open parentheses (GLLEV) , the actual address of the array LIST and a function code are saved. Then GLLEV is set to zero. The next LIST cell to be examined is pointed to by the list-pointer. When this part is put into PRBUFF, GLLEV will be equal to zero, and then the list containing this list-pointer will be analyzed.

g) TYPE 7

The next element is an array pointer which is treated as a normal literal atom.

h) TYPE 8

Using GETARG the next list-element will be fetched. Then this and the following LIST element is analyzed.

19.2 SUBROUTINE LSPEX

LSPEX is the normal interpreter exit routine. It calls TERPRI for printing the contents of the printerbuff PRBUFF and writes some information about the garbage collection and then stops the interpreter.

19.3 SUBROUTINE IPRINT

Parameters are:

I -- item to be printed

IPRINT calls PRIN1 for storing I into PRBUFF and TERPRI for printing the printerbuffer.

19.4 SUBROUTINE TERPRI

This subroutine writes the contents of the printerbuffer PRBUFF via actual output channel LUNUT.

19.5 SUBROUTINE PRINAT

Parameters are:

X -- item to be printed

GILEV -- number of open parentheses

PRINAT is the subroutine used to print an atom. First it is tested, whether the number of the top level elements or open parentheses has been overstepped. In this case, either the character "." or "-" is printed. Then, for the different types of atoms, the following occurs:

a) Literal atom or string

Using GETPN, the byte-address and the length of the atom is fetched. Besides this GETPN returns, whether the item is a literal atom or a string. If the normal printing is desired which means DREG(5) is equal to NIL, a string will be printed as a literal atom. If the atom does not fit into the actual line, TERPRI will be called. If the atom still doesn't fit in the line and the left margin LMARG has been changed during this printing, LMARG will be reset to the saved left margins until LMARG is equal to 1 or the item will fit in the line.

All characters and, if necessary, the double quote (") is put into the printerbuffer PRBUFF. If the printerposition PRTPOS is greater than the right margin MARGR, the printerbuffer will be printed, and the next characters will be put into PRBUFF.

b) number

Using GETNUM PRINAT will get the decoded number. If it is the number 0.0, these 3 characters are put into the printerbuffer, and PRINAT returns. It the number is negative, the variable SIGN is set, and the number is made positive. If the number is a real, the integer part of the number is rounded and saved.

In both cases the integer will be put into the printerbuffer first. The digits will saved in reversed order into the buffer ABUFF. It is tested, whether the number will fit in the actual line. If necessary, the sign and then the digits from ABUFF are put into PRBUFF. For a real, a radix point is put into PRBUFF, and the fraction is multiplied by 10**5 and rounded. The first 5 digits are saved into an integer, and the same mechanism is applied for storing these 5 digits into ABUFF as for the integer part of the number. If an exponent has to be written, it is put into the printerbuffer.

c) an array-pointer

If the printerbuffer is filled, TERPRI is called. Then, the hexadecimal representation of the array pointer is put into the printerbuffer.

20 File IOFN

contents:

SUBROUTINE RDA

SUBROUTINE WSTACK

SUBROUTINE MESS

20.1 SUBROUTINE RDA

Parameters are:

LUN	--	logical channel number for input
CARD	--	buffer to be write on
I1	--	first element of CARD which has to be write on
I2	--	last element of CARD which has to be write on
IEOF	--	exit-code

RDA reads a line into the array CARD via logical channel LUN. If the contents of the first cell of CARD is the symbol "\", the debugger is called.

20.2 SUBROUTINE WSTACK

WSTACK prints the contents of the function and argument stack. Following a header, on the left the function stack contents is printed [STACK(1..IP)], and on the right the argument stack [STACK(JP..NSTACK)].

20.3 SUBROUTINE MESS

Parameters are:

I -- MESS control value

I = 0	read the messages into the message buffer IMESS
I > 0	print IMESS(I)

MESS is called either to initialize the IMESS array during system startup, or to print a message.

21 File ROLL

contents:

INTEGER FUNCTION ROLLIN

SUBROUTINE ROLLOUT

SUBROUTINE DMPIN

SUBROUTINE DMPOU

21.1 INTEGER FUNCTION ROLLIN

Parameters are:

K -- logical channel number for input

ROLLIN reads a binary image of a previously defined interpreter status back into memory.

First, the array COMA will be read which contains the dynamic pointers. If their values do'nt fit in the corresponding arraysizes, ROLLIN returns -1, indicating unusable data on disk.

Using DMPIN the following arrays will be filled:

the first 83 variables of COMMON B	--	COMB
the 24 characters of the interpreter	--	COMCH
the character table	--	CHTAB
the hash table	--	HTAB
inforamtions about one atom or string	--	CAR
	--	CDR
	--	PNP
the interpreter messages	--	IMESS
the stack	--	STACK
the lisp-lists	--	LIST
the printnames, real numbers and arrays	--	PNAME

Then ROILIN rewinds the file and returns the logical channel number.

21.2 SUBROUTINE ROLLOU

Parameters are:

K	--	logical channel number for output

ROLLOU saves the current interpreter status on a disk file.

First, all swappable arrays will be swapped out, and then the garbage collector will be called, because it isn´t necessary to save unused space. Using DMPOU the contents of following arrays will be saved:

the first 83 variables of COMMON B	--	COMB
the 24 characters of the interpreter	--	COMCH
the character table	--	CHTAB
the hash table	--	HTAB
inforamtions about one atom or string	--	CAR
	--	CDR
	--	PNP
the interpreter messages	--	IMESS
the stack	--	STACK
the lisp-lists	--	LIST
the printnames, real numbers and arrays	--	PNAME

21.3 SUBROUTINE DMPIN

Parameters are:

LUN	--	logical channel number for input
AREA2	--	array which will be filled depending on I3
AREA4	--	array which will be filled depending on I3
I1	--	first cell which will be filled
I2	--	last cell which will be filled
I3	--	indicates buffer to be used
I3	= 1	AREA2 has to be filled
I3	= 2	AREA4 has to be filled

DMPIN reads from the logical channel I2-I1 words or halfwords into an array. The parameter I3 states, whether the array AREA2 has to be filled with halfwords (I3 = 1) or the array AREA4 has to be filled with words (I3 = 2).

21.4 SUBROUTINE DMPOU

Parameters are:

LUN	--	logical channel number for output
AREA2	--	array which contents will be saved depending on I3
AREA4	--	array which contents will be saved depending on I3
I1	--	first cell which will be saved
I2	--	last cell which will be saved
I3	--	indicates buffer to be used
I3 = 1		AREA2 has to be saved
I3 = 2		AREA4 has to be saved

DMPOU writes on the logical channel I2-I1 words or halfwords from an array. The parameter I3 states, whether the array AREA2 has to be saved (I3 = 1) or the array AREA4 has to be saved (I3 = 2).

22 File GBC

contents:

INTEGER FUNCTION GARB

SUBROUTINE REHASH

22.1 INTEGER FUNCTION GARB

Parameters are:

IGARB	--	garbage collection type
IGARB = 1		compress litatom and string pointer space
IGARB = 2		compress printname and array space
IGARB = 3		compress atom, string and array space
IGARB = 4		compress list space
IGARB = 5		compress all arrays

GARB is the garbage collector subroutine. It functionally contains 3 major sections:

a) All active cells will be marked which means all array-cells, which can be reached.

b) Depending on the garbage type the array will be compressed.

c) The pointers will be corrected and the array-cells unmarked.

These 3 sections will be explained in more detail below.

a) Marking all active cells

An active cell is an element of some FORTRAN array which can be reached by evaluation of some user input. There are various conditions for the different data types, which cause cells to be active:

```
litatom  --  bound variable
         --  function-name
         --  value bound to another atom
         --  element of an active LISP array
         --  element of an active LISP list

string   --  bound to a variable
         --  element of an active LISP array
         --  element of an active LISP list

number   --  bound to a variable
         --  element of an active LISP array
         --  element of an active LISP list

list     --  function definition
         --  bound to a variable
         --  element of an active LISP array
         --  element of an active LISP list
         --  list currently worked on

array    --  bound to a variable
         --  element of a LISP array
         --  element of a LISP list
```

For finding all active cells, the following arrays and variables have to be inspected:

Variables

```
ARG1    ARG2    ARG3    ALIST    FORM
TEMP1   TEMP2   TEMP3   I1CONS   I2CONS
```

These variables are equivalenced with the 10-element FORTRAN array ARGS.

Arrays

```
STACK(JP)       ..  STACK(NSTACK) (argument stack)
(ARGS(1)        ..  ARGS(10))
CAR(1)          ..  CAR(NATOMP/2+1)
CDR(1)          ..  CDR(NATOMP/2+1)
LIST(1)         ..  LIST(NLISTP) (every active cell)
PNAME(NARRYP)   ..  PNAME(NPNAME) (active elements)
PNAME swap buffer section for each swappable array
```

GARB scans the 4 arrays and marks the different data in the following way:

litatom or string	--	set CDR(litatom) negative
number	--	the bit-number of MARK is less or equal to the size of PNAME. If a number is active, with it's value stored PNAME cell number I, bit number I of MARK will be set to 1.
array	--	Bit number 4 of the first word of its header will be set to 1.
list	--	Each element of the list will be marked by setting bit 4 to 1.

If a LISP list or -array is reached, GARB has to examine each element and mark the different items. Passing through is performed in the following way:

LISP lists

It is tested, whether this list is already marked. If it is, return. If not, a function code specifying the array containing the pointer to this list-cell is saved on the stack. The cell is then marked and will be inspected.

Depending on the data type of the item, control is transferred to the corresponding marking section. If it is neither a litatom, string, number or array, GARB acts in the following way.

If it is a dot, the next list-element is fetched.

If it is a left parenthesis, GARB has to examine, whether the left parenthesis is reached by a list-pointer. If this is true, the address of the list-pointer is saved, otherwise a left parenthesis will be stored. Then the next element will be fetched.

If it is a right parenthesis, it is tested, whether the whole list is marked. If necessary, the next element is popped from the stack. If it is a parenthesis, the scan continues with the following LIST cell, otherwise with the address fetched, increased by 1.

If it is a list-pointer, its address is saved and the referenced cell is tested.

list-element are replaced by the new address.

If it is a cell which already is marked, the next unmarked cell is searched.

In all cases it must be tested, whether this list is the physically last one in LIST, which also is not neccessarily complete. This means that the list (currently under construction) does not have the same number of left parentheses as right ones.

LISP arrays

It is tested, whether this array is already marked. If it is , return.

Otherwise a function code is saved on the stack specifying the FORTRAN array containing the pointer to this array. The array is marked, and the high and low PNAME address of the array is fetched.

Now the array elements in the pointer region are tested. If such a cell contains a litatom, string, number or a list-pointer, control is transferred to the corresponding marking section.

If an element holds a pointer to another array, it´s address and the high address of te actual array will be saved.

If the whole array has been marked, it is tested by inspecting the stack, whether this array is pointed to by an element of another array. If it is, the scan continues with the element following the one, whose address was popped from the stack.

b) Compressing the FORTRAN arrays

Depending on the garbage type GARB has to act in 5 different ways:

b.1) TYPE 1

The FORTRAN arrays CAR, CDR und PNP have to be compressed. All cell contents belonging to the litatoms have to be moved to the top of the corresponding FORTRAN arrays. GARB passes through the arrays CAR and CDR and looks for addresses where the CAR-cell contents is equal to UNBOUND, STRING or SUBSTRING, and the CDR-cell is positive which indicates passive litatoms.

Then GARB looks for the next active litatom as a starting point for the move. For each active litatom, HTAB contains at the old address the new address, equal whether it had been moved or not. Starting at this address the same action will be taken until all active litatom-cells are moved.

b.2) TYPE 2

b.2.1) Compressing the litatom-part of PNAME.

Because the swap-buffer will be used, GARB swaps all swapable arrays. Then, all active numbers (with corresponding bit-numbers in MARK equal to 1) will be stored into SWPBUF until the buffer is filled up.

Starting at the byte in PNAME containing the first character of the first atom following the atom T, all characters belonging to active litatoms will be stored sequentially one after the other until the first byte of an active number not stored in SWPBUF is reached, or all active litatom-bytes are moved. The old byte-pointers in PNP have to be replaced by the new ones.

Now, the numbers in SWPBUF are restored to PNAME starting at the next word address. The size of the array HTAB is greater than the array CAR, so the leftover cells are used for saving the new number- addresses as two-byte entries: the first cell contains the word address and the second the number of words in the actual block.

If there are more active numbers, they are handled in the described way. If there are more active litatom-bytes, they are moved as described above.

b.2.2) Compressing the array-part of PNAME.

All active lisp-arrays are moved to the end of PNAME. This is done by searching active arrays and non-active cells starting at the end of PNAME and shifting the array contents to the end as much as possible.

b.3) TYPE 3

The arrays CAR, CDR, PNP and PNAME are compressed in the way described above. This is a combination of garbage collection type 1 and 2.

b.4 TYPE 4

Compressing the array LIST.

b.4.1) Starting from the top, a block of more than 3 contiguous passive cells is searched, and the length is saved in J.

b.4.2) Starting from the bottom, (J-1) active cells are searched.

b.4.3) The active cells are moved into the block of passive cells. Into the last passive cell, a continuation marker is put, pointing to the block last moved. The new addresses of the cells are stored into the corresponding old cells.

b.4.4) Continue with b.4.1 until all active cells are compacted.

Note that two addresses have to be handled in a special way:

1. The new address of the old last cell. Into this cell a continuation marker to the new next free LIST-cell has to be written.

2. The new address of the block last moved, because a continuation marker must be in the last used cell after the garbage collecter has finished.

b.5) TYPE 5

CAR, CDR, PNP, PNAME AND LIST have to be compressed in the way described above. This is a conbination of garbage collection of type 3 and 4.

c) Updating the pointers and unmarking the array-elements

GARB passes the 4 arrays as in part a) and updates the pointers in the following way:

Litatoms and strings

If the garbage type is equal to 1, 3 or 5, the new litatom address is fetched from the HTAB cell, whose number is equal to the old address.

Numbers

If the garbage type is equal to 2,3 or 5 and it is not a small integer, the new address is computed from the contents of the last part of HTAB.

First, the MARK bits equal to 1 are counted, until the bit corresponding to the number is reached. HTAB contains two-word entries, giving information about contiguous blocks of numbers. The new address is found by calculating the block number and the offset within the block, where the number has been stored.

Arrays

If garbage type is equal to 2, 3 or 5, the new address will be computed from the number of bits equal to 1 and the length of the arrays, whose addresses are higher than the one of the actual array. Starting at the bit, whose number is equal to NARRYP, the number of bits equal to 1 are counted until the bit is reached, whose number is equal to the desired array address.

Using NARRYP the length of the first array will be fetched. Adding the length to NARRYP the next array will be reached, and its length added to its address for getting the next array until the new address of the actual one is reached.

Lists

If garbage type is equal to 4 or 5, the new address of the LISP list has to be fetched from the old address, if it has changed.

If a LISP list or -array is reached, GARB has to examine every element and to update the pointers of the items. Passing through is done in the same way as described in part a).

When reaching an atom, array or a list, all active cells will be reset. The bits will be set to Ø, and the CDR-cells will be set positive.

After the 3 sections are executed, the hashaddresses will be reset. If desired, GARB prints some information and increments the type-dependant garbage collection count variable by one.

22.2 SUBROUTINE REHASH

REHASH resets the hash table by scanning through the FORTRAN structures describing the atoms.

First the hash table HTAB is initialized by setting all cells to UNUSED.

Then the array CAR will pass through for computing a new hashaddress of each literal atom. The length and the characters will be fetched, and the address will be computed as in the function MATOM.

FORTRAN Element Index

Appendix 3: LISP-SP reference guide

[p1,p2,..,pn] fsubr

An s-expression beginning with * is interpreted as a comment. Since * is interpreted in the same way as QUOTE (i.e., returns its first argument), comments should be placed, where they will not harm computation. If SYSFLAG[4] is set to T, then all comments will be discarded on input in order to save PNAME-space.

abs[x] expr

returns x, if x>0; otherwise -x.

add1[x] subr

returns x + 1

addlist[a,l] subr

if MEMB[a,l], returns l else returns CONS[a,l]

addprop[atm,prop,new,flg] expr

adds the value new to the list which is the value of property prop on property list of atm. If flg is T, new is CONSed to the front of value of prop, otherwise to the end. If atm does not have the property prop, or if the value of prop isn't a list, then prop will be added with the value LIST[new] to the property list of atm. Returns new.

advise[fn,when,where,what] expr

advises fn, when=BEFORE or AFTER, where specifies, where among the advises this new advise is put, can be specified as LAST (NIL) or FIRST or by editor commands, what specifies, what code to put in.

alist subr

returns the current value binding stack.

alphorder[a,b] subr

returns T, if arguments are in alphabetical order. Numbers come before literal atoms, and are ordered by magnitude. Literal atoms are ordered by comparing their pnames. If neither a or b are atoms or strings, the value of ALPHORDER is T.

and[x1,x2,..,xn] fsubr*

If all arguments are non-NIL, then AND returns its last argument. If some argument evaluates to NIL, then evaluation stops and NIL is returned.

antilog[x] subr

value is floating point number, whose logarithm is x. X can be integer or floating point.

append[x1,x2] subr

copies the top level of list x1 and appends to this x2. APPEND[x1] can be used to copy the top level of x1. For non-lists you get:

```
APPEND[A,(B C D)]            = (B C D)
APPEND[(A B C . D),(E F G)]  = (A B C E F G)
APPEND[(A B C),D]            = (A B C . D)
APPEND[(A B C . D)]          = (A B C . D)
```

apply[fn,args] subr

applies the function fn to the arguments collected in args. The arguments are not evaluated by apply; their evaluation depends only on fn. APPLY returns as ists value the value of fn applied to args. APPLY['CONS,'(A B)] = (A . B)

apply*[fn,arg1,arg2,..,argn] expr

equivalent to APPLY[fn,LIST[arg1,..,argn]]. Returns as value the value of fn applied to arg1..argn.

applya[fn,l,ass] subr

Variable bindings are stored in an association list, which simulates a push-down stack (see Interlisp). This list is passed to EVAL, APPLY and EVLIS implicitly. If however evaluation is to be performed in a special variable environment, then an association list can be passed explicitly. APPLYA[fn,l,ass] is equivalent to APPLY[fn,l], but uses ass as the push-down stack.

arccos[x,radiansflag] subr

returns arc cosine of x in degrees unless radiansflag=T. Range is 0..180, 0 to pi.

arcsin[x,radiansflag] subr

returns arc sine of x in degrees unless radiansflag=T. Range is -90..90, -pi/2..pi/2.

arctan[x,radiansflag] subr

returns arc tangent of x in degrees unless radiansflag=T. Range is 0..180, 0 to pi.

array[n,p,v] subr

allocates an array of n elements. If p is NIL, then all elements will contain pointers, initialized to v. Note that both car and cdr are available for storing information. If p>0, then the first p elements will be unboxed numbers, initialized to 0. The value of ARRAY is the so-called array-pointer which has as pname the hexadecimal presentation of the array's address, preceded by a "#". Note that the array pointer is not an atom.

arraybeg[x] subr

returns x, if x is an array; otherwise NIL.

arrayp[x] subr

returns x, if x is an array; otherwise NIL.

arraysize[x] subr

returns, the size of array x. Generates an error, if x is not an array.

arraytyp[x] subr

returns a value corresponding to the second argument to ARRAY, i.e., the number of unboxed number elements in x.

assoc[key,alst,fn] expr

alst is a list of dotted pairs. The value of ASSOC is the first sublist of alst, whose car is EQ to key, if fn is NIL. If fn is non-NIL, then the value of ASSOC is the first sublist y with fn[x,y]=T.

atom[x] subr

returns T, if x is an atom; otherwise NIL.

break[fn1,fn2,..,fnn] fexpr*

applies BREAKØ to each argument. For atomic arguments, BREAKØ[fni,T] is performed. For lists, APPLY['BREAKØ,fni]] is performed.

breakØ[fn,when,coms] expr

modifies the definition of fn by replacing its body to a BREAK1-call, where:

```
brkexp  = PROGN[fn-body]
brkwhen = when'
brkfn   = fn'
brkcoms = coms
```

The original function definition will be put as value to the property VIRGINFN of fn. If the value of property BROKEN is NIL, then (fn when coms) will be stored under BROKEN; otherwise the definition of fn will be rplaca'd to BROKEN. fn will be added to the front of BROKENFNS. If fn is not a function, BREAKØ returns NIL, otherwise fn.

break1[fn1,fn2,..,fnn] fexpr*

performs APPLYA['BREAK11 [fn1,..,fnn,CDR[ALIST]]]. Note that BREAK11 is an expr with 4 arguments. See BREAK11.

break11[brkexp,brkwhen,brkfn,brkcoms] expr

If brkwhen is non-NIL, and brkcoms does not contain break-commands, then a message of the form (brkfn BROKEN) is printed and commands are read from the terminal. If brkcoms contains break-commands, then these commands are executed one-by-one. Commands are:

!	: return to previous break, if any; otherwise reset.
GO	: print broken form and continue.
OK	: continue.
RETURN x	: return the value of x.
EVAL	: evaluate broken form and break afterwards. The value of the form is stored under VALUE.
!EVAL	: as EVAL etc., but the function
!GO	: is first unbroken
!OK	: then rebroken.
UB	: unbreaks the function
BR	: breaks the function
BT	: backtrace of function calls (only LAMBDA and NLAMBDA). This is only possible, if you have performed SETQ[*BACKTRACEFLG,T] before evaluation.
ALIST	: print current value-binding stack. (except for variables bound in BREAK1 and SYSERROR)

Any other input is evaluated and the value is printed.

car[x] subr

CAR gives the first element of a list x, or the left element of a dotted pair x. CAR[NIL] is NIL. For all other non-lists, CAR[x] is undefined. Note that successive CAR´s and CDR´s can be specified in short-hand up to CDDDR.

cdr[x] subr

CDR gives the rest of a list (all but the first element). This is also the right member of a dotted pair. CDR[NIL] is always NIL. For other non-lists, CDR is undefined. ** NOTE ** Successive CAR´s and CDR´s can be specified in short-hand up to CDDDR.

chtab[x,n] subr

returns the character type of the first character of the atom x, if n is NIL. Otherwise, the type of the first character of x is set to n (which changes the meaning of character). Default character types are:

1	space	6	"	11	+
2	(	7	´	12	-
3	)	8	break(.)	13-22	Ø-9
4	<	9	.	23	(1)
5	>	1Ø	alfanum	24	

(1) note that the type 34 must be selected depending on the terminal type available (ATM-8Ø6Ø: ∩ ,ATM 7845: {).

clock subr

returns time as a dotted pair (hours . minutes).

close[f] expr

close file f. f is the symbolic name of the file.

concat[x1,x2,..,xn] subr*

concatenates copies of the strings x1..xn. If the arguments aren't strings, they are converted. CONCAT[] is "" , the empty string.

cond[c1,c2,..,cn] fsubr*

The arguments of COND, c1..cn, are called clauses. Each clause ci is a list (pi,ei1,..,eim), where pi is the predicate, and ei1..eim are the consequents. COND evaluates pi, i=1..n until it finds some pi as non-NIL. Next, the following expressions ei1,ei2.. are evaluated, and the value of COND is the value of the last expression evaluated. If some pi evaluates to non-NIL and no expression follows in that clause, then the value of COND is pi. If no pi evaluates to non-NIL, then the value of COND is NIL.

cons[x,y] subr

constructs a dotted pair of x and y. If y is a list, x becomes the first element of that list.

copy[x] expr

makes a new list which is a copy of x and EQUAL to x but not EQ to x. All levels will be copied except strings and arrays.

copyarray[a] subr

creates a new array of same size and type as a, i.e., the same distribution of pointers and unboxed numbers, and with the same contents as a. Value is new array. If a is not an array, an error is generated.

cos[x,radiansflag] subr

x must be in degrees unless radiansflag=T. Returns cosine of x as a floating point number.

curfile[file] fexpr

defines file as current file. All subsequent function definitions belong to this file and are added to fileFNS. If CURFILE[file] is not evaluated, the name of the current file is CUR and the function names are saved on CURFNS. To define some functions and save them as MYFILE on logical unit 25 you write:

```
(OPEN 'MYFILE 'O 25)
(CURFILE MYFILE)
(DE...>
(DE...>  etc.
(MAKEFILE 'MYFILE T)
```

This pretty-prints a version of all definitions to logical unit 25.

de[fn,args,body] fexpr*

assigns a function definition of type LAMBDA to the atom fn. args is the list of parameters for spread or half-spread functions (type expr) or an atom for nospread functions (type expr*).

defineq[x1,x2,..,xn] fexpr*

defines functions as specified by x1..xn. Each xi is an expression of the form

```
(fn-name (LAMBDA ... ))   or
(fn-name (NLAMBDA ... ))
```

No message is given, if some fn-name has been used for a function definition before.

deflist[l,prop] expr

puts values under the same property name prop on the property lists of several atoms. l is a list of two-element lists, the first element of which is a literal atom, and the second is the value to put to property prop of that atom. The value of deflist is NIL.

df[fn,args,body] fexpr

assigns a function definition of type NLAMBDA to the atom fn. args is the list of parameters for spread or half-spread functions (type fexpr) or an atom for nospread functions (type fexpr*).

difference[x,y] subr

returns x - y.

dsort[l] expr

destructively sorts l.

editf[d] fexpr

edits a function. The argument d is a dotted pair (fn . edcom), where fn is the name of the function to be edited, and edcom is a list of edit commands. For edit commands see EDITS[].

edits[s,edcom] expr

edits any s-expression s with edit-commands in list edcom. For edit commands see next two pages.

Edit commands

P	:	print to level 2
PP	:	pretty-print to level 2
?	:	print to level 1000
??	:	pretty-print to level 1000
OK	:	leave editor.
UP	:	new cexpr (current expression) is expression with old cexpr as car.
F expr	:	if expr is an atom, top level of cexpr is searched for expr and cexpr is set to the expression with expr as car. If expr is not found on top level, then all levels are searched from the beginning. If expr is a list, then the new cexpr is the first expression which matches expr, regardless of its level.
F s	:	???
NX	:	sets current expression to next expression.
!	:	sets current expression to top level expression.
S x	:	set x to the current expression. Useful in combinations with US.
n	:	if positive, sets cexp to the n'th element of cexpr. If negative, search starts at the end. If n=0 ???

more edit commands

(n)	: n>=1 deletes the n´th expression of cexpr.
(n e1..em)	: n>=1 replaces the n´th expression by e1..em
(-n e1..em)	: n>=1 inserts e1..em before n´th element.
(N e1..em)	: adds e1..em to the end of cexpr
(R x y)	: replaces all occurances of x in cexpr by y.
(BI n m)	: both in. Inserts a left parenthesis before : the n´th element and a right parenthesis after : the m´th element.
(BI n)	: same as (BI n)
(BO n)	: both out. Removes both parentheses from : n´th element.
(LI n)	: left in. Inserts a left parenthesis before : the n´th element and a corresponding right : parenthesis at the end.
(LO n)	: left out. Removes the left parenthesis from : the n´th element. All elements after the : n´th element are deleted.
(RO n)	: right out. Removes right parenthesis from : the n´th element, moving it to the end of : the current expression. All elements following : the n´th element are moved inside the n´th : element.
(: e1..en)	: replaces cexpr by e1..en.
(US x coms)	: Use a copy of the saved value of x in commands.
(MARK x)	: save the current chain in x.
(\ x)	: reset the edit chain to x.

Note: the Interlisp print commands are not exactly like these.

eject subr

do a form-feed to current output channel

elt[a,n] subr

returns the n'th element of array a. If a is not an array, an error is generated. If n corresponds to the unboxed number region of a, then the value is returned as boxed integer. If n corresponds to the pointer region of a, then the value of ELT is the car half of the corresponding element.

eltd[a,n] subr

returns same as ELT for unboxed region of array a, but returns cdr half of the n'th element, if n corresponds to the pointer region of a.

eq[x,y] subr

The value of EQ is T, if x and y are pointers to the same structure in memory, and NIL otherwise. For equal numbers, EQ gives T only, if x and y are in the range -2000..2000.

eqp[x,y] subr

returns T, if x and y are pointers to the same structure in memory, or if x and y are numbers with the same value.

equal[x,y] subr

evaluates to T, if EQ[x,y] is T, or EQP[x,y] is T, or if STREQUAL[x,y], or if x and y are lists and EQUAL[CAR[x],CAR[y]] and EQUAL[CDR[x],CDR[y]]. Otherwise the value of EQUAL is NIL.

errorb expr*

is a programmed return from an error situation. If the error occurred under ERRORSET, then ERRORB returns with value NIL, otherwise a RESET is performed.

errormess[n] subr

prints error message, whose number is n. If n=Ø, then all messages are listed.

errorn expr*

returns the number for the last error occurred.

errorset[form,flg] expr

performs EVAL[form]. Note that ERRORSET is a LAMBDA-function and therefore its argument are evaluated, before it is entered. This means that EVAL is called with the value of form. If no error occurs in the evaluation of form, the value of ERRORSET is the value of EVAL[form]. If an error occurred, then the value of ERRORSET is NIL.

eval[x] subr

evaluates the expression x and returns this value,i.e, EVAL provides a way of calling the interpreter. Note that EVAL is itself a LAMBDA type version, so its argument is first evaluated.

evala[x,a] subr

evaluates x using a as an association list. Any variable, which appears free in x and also appears as car of an element of a, will be given the value of the cdr of that element.

evlis[x] subr

performs a MAPCAR[x,'EVAL],i.e., EVLIS evaluates the elements of x one-by-one and returns a list which has as top level elements the results of the corresponding evaluation.

exit subr

exits the LISP system and returns to whichever environment it was entered from.

expt[m,n] subr

value is m ** n. If m is an integer and n is a positive integer, then the value is an integer, otherwise the value is a floating point number. If m is negative and n is fractional, an error is generated. If n is floating and either too large or too small, an error is generated.

fdifference[x,y] expr

difference x - y of two floating point numbers.

fgreaterp[x,y] expr

returns T, if x > y, otherwise NIL.

fix[x] subr

converts x to an integer by truncating fractional bits.

float[x] subr

converts x to a floating point number.

floatp[x] subr

returns T, if x is a floating point number, NIL otherwise. It does not give an error, if x is no number.

fminus[x] expr

returns -x

fplus[x1,x2,..,xn] subr*

returns sum of floating point numbers x1..xn.

fquotient[x,y] expr

returns x / y.

ftimes[x1,x2,..,xn] subr*

returns product x1 * x2 * .. * xn.

function[fn,env] fsubr*

is an NLAMBDA function. If env=NIL, the value of FUNCTION is identical to QUOTE. If env is not NIL, it can be a list of variables which are presumably used freely by fn. In this case, the value of FUNCTION is an expression of the form (FUNARG fn pos), where pos contains the variable bindings for those variables which are not in the argument list of fn.

gcgag[message] expr

affects message printing by garbage collector. If message=T, "collecting" is printed, followed by the type of the collection. When garbage collection is completed, free space information is printed in a format depending on the garbage collection type. See RECLAIM.

gensym subr

generates a new atom of the form Annnn, where each of the n's is a digit. Thus, the first one generated is A000, the second one A001, etc. The value of the atom GENNUM determines the next GENSYM, e.g. if GENNUM is set to 10023, then GENSYM[] yields A0023.

getd[x] subr

gets the function definition of x. Value either its definition or NIL (if x is not a literal atom, or has no definition).

getint[s,f] subr

gets an integer value from a string. The string or substring to be used is s. The argument f is used to control the format of the string. If f is T, then the integer is taken from the string as binary value, otherwise as sequence of ASCII digits.

getp[atm,prop] subr

gets the property value for prop from the property list of atm. The value of GETP is NIL, if atm is not a literal atom, or prop is not found.

go[x] fsubr*

transfers control in a PROG. (GO L) will cause the program to continue at the label L. A GO can be used at any level in a PROG. If the label is not found, GO will search higher PROG's within the same function. If the label is not found, GO informs about the error.

go*[x] fsubr*

searches all current PROG's for the label x. If it is found, a jump is performed. If not, NIL is returned and no other action takes place. GO* can be used to jump to a label defined in some other currently active function.

greaterp[x,y] subr

returns T, if x > y; NIL otherwise.

idifference[x,y] expr

returns x - y for two integers x and y.

igreaterp[x,y] expr

returns T, if x > y for two integers x and y, otherwise NIL.

iminus[x] expr

returns - x for integer x.

inunit[n] expr

sets input channel to n. If n is NIL, then the current channel number is returned.

iotab[i,n] subr

sets element i in IOTAB. This is a 10 element structure with the following contents:

```
element  1 :  logical input channel
         2 :  current read position
         3 :  left mardin - input
         4 :  right margin - input
         5 :  logical output channel
         6 :  current print position
         7 :  left margin - output
         8 :  right margin - output
         9 :  print length
        10 :  print depth
```

If n is NIL, then the current value of the specified element is returned. If n is T, the element is set to it's default value. Otherwise, the value of n is put into the table.

iplus[x1,x2,..,xn] subr*

returns the sum x1 + .. + xn, x1..xn integers.

iquotient[x,y] expr

returns x / y truncated, x and y integers.

itimes[x1,x2,..,xn] subr*

returns the product x1 * .. * xn, x1..xn integers.

last[x] expr

returns a pointer to the last node in a list. Value is NIL, if x is not a list. Example: LAST[(A B C)] is (C), LAST[(A B . C)] is (B . C).

length[x] subr

returns the length of the list x defined as the number of CDR's required to reach a non-list.

lessp[x,y] subr

returns T, if x < y, NIL otherwise.

lispx subr

defined as: LOOP PRINT[EVAL[READ]] GO[LOOP]

list[x1,x2,..,xn] subr*

returns the list of the values of it´s arguments.

listp[x] subr

returns T, if x is a list; otherwise NIL.

litatom[x] subr

returns T, if x is a litatom; otherwise NIL.

load[f] expr

reads successive s-expressions from file f and evaluates each as it is read, until it reads STOP. Returns f.

log[x] subr

returns natural logarithm of x as a floating point number. x can be integer or floating point.

makefile[f,flg] expr

takes a number of variables, functions and properties and writes them out to the previously opened file f. These definitions can then be read back later by LOAD. MAKEFILE reads commands of the form

```
(P    ... )
(PROP ... )
(E    ... )
```

where (P e1,...,en) specifies expressions to be printed on the file, (PROP p atom ...) defines values on atoms under property p, and (E e1,...,en) specifies expressions which will be evaluated and their values written to the file. See [HA75] for more information on the MAKEFILE package. If flg=T, then pretty printing is used.

map[mapx,mapfn1,mapfn2] subr

If mapfn2 is NIL, then MAP applies mapfn1 to successive tails of the list mapx. That is, first it computes mapfn1[mapx], and then mapfn1[CDR[mapx]], until mapx is exhausted. If mapfn2 is provided, mapfn2[mapx] is used instead of CDR[mapx] for the next call for mapfn1, e.g. if mapfn2 were CDDR, alternate elements of the list would be skipped. The value of MAP is NIL.

mapc[mapx,mapfn1,mapfn2] subr

Identical to MAP, except that mapfn1[CAR[mapx]] is computed at each iteration instead of mapfn1[mapx], e.g. MAPC works on elements, MAP on tails. The value of MAPC is NIL.

mapcar[mapx,mapfn1,mapfn2] subr

computes the same values MAPC would compute, and returns a list consisting of those values.

maplist[mapx,mapfn1,mapfn2] subr

computes the same values MAP would compute, and returns a list consisting of those values.

memb[x,y] subr

determines, if x is a member of the list y, i.e., if there is an element of y EQ to x. If so, its value is the tail of the list y starting with that element. If not, it´s value is NIL.

member[x,y] subr

is identical to MEMB except that it uses EQUAL instead of EQ to check membership of x in y.

minus[x] expr

returns - x

minusp[x] expr

returns T, if x is negative; NIL otherwise. It works for both integers and floating point numbers.

mkatom[x] expr

creates an atom, whose pname is the same as that of the string x, or if x isn´t a string, the same as that of MKSTRING[x].

mkstring[x] expr

returns as value a string corresponding to the prin1-pname of x.

mkswap[a] subr

makes array a swappable, i.e., the array is put on disk. Only the array header remains in the array space in memory. If a is not an array, or the array is bigger than the swap buffer, an error is generated.

mkunswap[a] subr

makes a swappable array memory resident. An error is generated, if a is not an array.

nchars[x] subr

returns number of characters in pname of x.

nconc[x1,x2] subr

returns the same value as APPEND[x1,x2], but actually modifies the list structure of x1 and x2.

nconc1[lst,x] subr

performs NCONC[lst,LIST[x]]

neq[x,y] subr

returns T, if x is not EQ to y; NIL otherwise.

nil subr

returns NIL.

nlistp[x] subr

returns T, if x is not a list.

nth[x,n] expr

returns tail of x beginning with the n´th element.

null[x] subr

returns T, if x EQ to NIL; NIL otherwise.

numberp[x] subr

returns T, if x is a number; NIL otherwise.

oblist[x] subr

creates a new list of atoms, with the last atom created as the first member of the list, and the atomic argument x as the last one. As NIL is the very first atom created, (OBLIST) gives a list all atoms and as T is the last atom defined by a clean system, (OBLIST T) gives all atoms but SUBR´s and FSUBR´s.

open[f,io,n] expr

opens the file with symbolic name f, with io=I for input and io=O for output, where n is used as the logical channel number.

or[x1,x2,...,xn] fsubr*

returns as value that of the arguments, whose value is not NIL; otherwise NIL, if all arguments have the value NIL. Evaluation stops at the first argument, whose value is not NIL.

outunit[n] expr

sets logical channel for output to n.

pack[x] subr

If x is a list of atoms, the value is a single atom, whose pname is the concatenation of the pnames of the atoms in x. If the pname of the value is the same as that of a number, the value will be that number.

plus[x1,x2,..,xn] subr*

returns the sum x1 + .. + xn of the numbers x1..xn.

pp[x1,x2,..,xn] fexpr*

prettyprints x1..xn.

prin0[x,a,b] subr

prints x with no TERPRI before or after. If a=NIL, does not print escape- or string-character.If a=T, then these characters are printed, then reading back properly is possible. If b=NIL, then it´s an ordinary print. If b=T, then ?

prin1[x] expr

prints x.

prin2[x] expr

print x with escape- and string-character inserted, where required for it to read back in properly by READ.

print[x] expr

prints x using prin2 followed by carriage-return / line-feed.

printdef[x] expr

pretty print x with escape- and string-character.

printl[x1,x2,..,xn] expr*

print x1,..,xn using print1.

printlength[x] expr

returns the maximum number of top elements to be printed, if x is NIL. If x is not NIL, then the "printlength" is set x. Elements beyond the current printlength are printed as "---".

printlevel[x] expr

returns the maximum number of levels to be printed, if x is NIL. If x is not NIL, then the printlevel is set to x. Lists below the current printlevel are printed as "...".

printpos[x] expr

returns the current print-position, if x is NIL; otherwise it sets the current print-position to x.

prog[varlst,e1,e2,..,en] fsubr*

allows for writing programs consisting of expressions to be executed. The varlst is a list of local variables.

If no local variables are to be used, then NIL must be specified. Each atom in varlst is treated as the name of a local variable and is bound to NIL. Also, varlst can contain lists of the form (atom form). In this case, atom is the name of the variable and is bound to the value of form. Evaluations take place before any bindings are performed. An attempt to use anything other than a literal atom as a PROG variable will cause an error.

The rest of the PROG is a sequence of non-atomic statements and atomic symbols, used as labels for GO. The forms are evaluated sequentially, the labels serve only as markers. The two functions GO and RETURN alter this flow of control as described below. The value of PROG is usually specified by RETURN. If no RETURN is specified, PROG returns NIL.

prog1[e1,e2,..,en] fsubr*

evaluates it's arguments in order, and returns the value of it's first argument.

progn[e1,e2,..,en] fsubr*

evaluates it's arguments in order, and returns the value of it's last argument.

put[a,p,v] subr

puts v to property p on atom a. Returns v.

putd[fn,def] expr

puts the definition def into fn´s function cell. Value is def.

putint[s,x,f] subr

puts the integer x into string or substring s. If f is T, then x is put into s in binary format. If f is NIL, tnen the pname of x is put irto s left-justified. If f is anything else, then the pname of x is put into s right-justified.

putprops[atm,prop1,val1,..,propn,valn] expr

for i=1 to n puts propi,vali on property list of atm.

quote[x] fsubr*

prevents it´s argument from beeing evaluated. Returns x itself.

quotient[x,y] subr

returns x / y of two numbers x and y.

rand[lower,upper] subr

returns a pseudo random number between lower and upper inclusive, i.e. RAND can be used to generate a sequence of random numbers. If both limits are integers, the value is an integer, otherwise it is a floating point number.

The algorithm is completely deterministic, i.e., given the same initial state, RAND produces the same sequence of values. The internal state of RAND is initialized using the function RANDSET described below.

randset[x] subr

returns the internal state of RAND after RANDSET has finished operating. If x is NIL, value is the current state. If x is T, the variable RANDSTATE is initialized using the clock. Otherwise, x is interpreted as a previous internal state, i.e. a value of RANDSET, and is used to reset RANDSTATE.

ratom[x] subr

reads an atom. If x is not NIL, it is interpreted as the logical channel to read from.

read[x] subr

reads an s-expression. If x is not NIL, then it is interpreted as the logical channel to read from.

readc[x] subr

reads the next character. If x is not NIL, it is interpreted as the logical channel to read from.

readfile[x] expr

reads successive s-expressions from logical channel x until an error occurs, e.g. unbound variable. Each expression is evaluated as it is read. Returns x.

readpos[x] expr

returns the current read position. If x is not NIL, then the current read position is set to x.

readvise[x] fexpr*

restores a function to it's advised state. For each function on x, READVISE retrieves the advise information from the property ADVISED for that function and performs the corresponding advise operations. The value of READVISE is a list of the function names specified. If no advise information is available for some function, then NIL is returned instead of the function name.

rebreak[x] fexpr*

rebreaks each function on x by retrieving break information for the property BROKEN for that function and performing the corresponding operation. Value is a list of values corresponding to the values of BREAKØ. If no break information is found for a particular function, then NIL is returned for that function.

reclaim[x] subr

Initiates garbage collection of type x, where x is:

1 : atom pointer space compression

2 : pname space compression

3 : both atom and pname space compression

4 : list space compression

5 : compresses all structures

Value is number of words available for the corresponding type after the compression as a dotted pair (m . n), with number = m*1ØØØ + n. Type 3 returns atom pointer space, Type 5 returns list space. Note that pname space compression affects the pnames of atoms and strings, as well as numbers and arrays.

remainder[x,y] subr

returns modulus of x and y. Value is integer, if both arguments are integers, otherwise real.

remove[x,l,l1] expr

removes all occurences of x from list l, giving a copy of l with all elements EQUAL to x removed. This list will be bound to l1.

remprop[atm,prop] expr

removes the property prop (and it's value) from the property list of atm. Value is prop,if it was found, otherwise NIL.

reset subr

returns to top level immediately.

return[x] subr

is the normal exit from a PROG. It's argument is evaluated and is the value of the PROG, in which it appears.

reverse[l] subr

reverses and copies the top level of a list. If l is not a list, l is returned.

rewind[x] subr

rewinds logical channel x.

rollin[x] subr

restores a saved LISP system status from logical channel x. It is possible to perform ROLLIN even, if the size of the LISP system has been changed since the last ROLLOUT. If no proper ROLLIN can be performed, then ROLLIN returns NIL, otherwise x.

rollout[x] subr

saves the current LISP status on logical channel x. Returns x.

rplaca[x,y] subr

replaces the address pointer of x, i.e. car, with y. Value is x. An attempt to replace NIL will cause an error, except for RPLACA[NIL,NIL]. An attempt to RPLACA any other non-list will cause an error.

rplacd[x,y] subr

replaces the cdr of x by the pointer y. The internal list structure is physically changed. The only way to get a circular list is by using RPLACD to place a pointer to the beginning of a list in a spot at the end of the list. Value is x. An attempt to RPLACD NIL will cause an error, except for rplacd[NIL,NIL]. An attempt to RPLACD any other non-list will cause an error.

rplstring[x,n,y] subr

replaces characters of string x beginning at character n with string y. n may be positive or negative. If n is positive, then the first characeter to be replaced is the n'th, counted from the beginning, otherwise from the end. The characters are put into x. Value is new x. An error is generated, if there is not enough room in x for y. Note that, if x is a substring of z, then z is also changed.

rpt[rptn,rptf] subr

evaluates the expression rptf rptn times. At any point, rptn is the number of evaluations yet to take place. Returns the value of the last evaluation. If rptn<=Ø, then rptf is not evaluated, and the value of RPT is NIL.

rptq[rptn,rptf] fexpr

as RPT[rptn,rptf], but does not evaluate rptf before execution.

sassoc[key,alst] expr

returns ASSOC[key,alst,EQUAL].

savedef[fn] expr

saves the definition of fn on it's property list under property EXPR. Value is EXPR.

selectq[x,c1,c2,...,cn,def] fsubr*

selects a form or sequence of forms based on the value of it's first argument x. Each ci is a list of the form (si e1i e2i..eki), where si is the selected key.

If si is an atom, the value of x is tested, if it is EQ to si (not evaluated). If so, the expressions e1i..eki are evaluated in sequence, and the value of the SELECTQ is the value of the last expression evaluated, i.e., eki.

If si is a list, the value of x is compared with each element (not evaluated) of si, and, if x is EQ to any of them, then e1i to eki are evaluated in turn as above.

If ci is not selected in one of the two ways described, ci+1 is tested, etc., until all the clauses cj have been tested. If none is selected, the value of the selectq is the value of def which must be present.

set[x,y] subr

sets x to y. Value is y. If x is not a litatom, then an error is generated.

seta[a,n,v] subr

sets the n'th element of array a. If n corresponds to the unboxed number region of a, v must be an integer. If n corresponds to the pointer region, v replaces the car half of element n. If a is not an array, an error is generated.

setd[a,n,v] subr

same as SETA for unboxed region of array a, but sets cdr half of n'th element, if n corresponds to the pointer region. Value is v.

setq[x,y] fsubr*

same as SET[x,y], but x is not evaluated.

setqq[x,y] expr*

same as SETQ[x,y], but y is not evaluated.

setsbsize[x] subr

returns the swap buffer size. Changing the swap buffer size dynamically is not implemented in this version.

sign[x] expr

returns Ø or 1 or -1, depending on the sign of x.

sin[x,radiansflg] subr

x must be in degress unless radiansflg=T. Returns sine of x as a floating point number.

spaces[n] expr

prints n spaces.

sqrt[x] subr

returns square root of x as a floating point number. x may be integer or floating point. Generates an error, if n is negative.

stralloc[n,c] subr

allocates a new string of length n and fills that string with the first character of the atom / string / substring c.

strequal[x,y] expr

returns x, if x and y are both strings and equal, i.e., print the same, otherwise NIL. Note that strings may be EQUAL without being EQ.

stringp[x] expr

returns x, if x is a string, NIL othwerwise. Note that, if x is a string, then NLISTP[x] is T but ATOM[x] is NIL.

sub1[x] subr

returns x - 1.

swparrayp[x] subr

returns x, if x is a swappable array, NIL otherwise.

subst[new,old,expr] subr

Value is the result of substituting the s-expression new for all occurrences of the s-expression old in the s-expression expr. Substitution occurs, whenever old is EQUAL to car of some subexpression of expr, or when old is both atomic and not NIL and EQ to cdr of some subexpression of expr. The value of SUBST is a copy of expr with the appropriate changes. Furthermore, if new is a list, it is copied at each substitution.

substring[x,n,m] subr

Value is the substring of string x consisting of the n´th through m´th character of x. If m is NIL, the substring is the n´th character of x through the end of x. Both n and m can be negative, in which case counting begins at the end. Returns NIL, if the substring is not well defined. SWPARRAYP[a] returns a, if x is a swappable array, otherwise NIL.

syserror subr

is defined as a function which prints some errormessage and resets the system. It will be redefined, however, when reading in the exprs, and will then cause a break, if an error occurs. If the evaluation which caused the error was initiated by ERRORSET, then no break will occur, and the value of the function will be NIL.

sysflag[i,x] subr

sets SYSFLAG(i) to x and returns the old value. If x is not specified, only the current value is returned. x can be T or NIL. Setting the SYSFLAGs means:

1 : print GBC messages

2 : output is pretty printed

3 : enable stack printing

4 : comment

5 : print escape- and string-character

6 : not used

7 : begin a new line, whenever a "("
is found during pretty print unless
it is the first (sometimes second)
subexpression

tailp[x,y] expr

returns x, if x is a tail of y, i.e., x is EQ to some number of cdr's of y, otherwise NIL.

tan[x,radiansflg] subr

x is in degrees unless radiansflg=T. Returns tangent of x as a floating point number.

terpri subr

prints a carriage return, value is NIL.

times[x1,x2,..,xn] subr*

returns the product x1*x2*..*xn of the numbers x1..xn.

trace[x] fexpr*

causes the functions, whose names are on the list x to be traced. Retunrs a list of the function names. If some element is not a function, then NIL is returned for that element.

unadvise[x] fexpr*

takes any number of functions an restores them to their original unadvised state, including removing the properties added by advise. UNADVISE[] unadvises all functions on ADVISEDFNS.

unbreak[x] fexpr*

takes any number of functions modified by BREAK or TRACE and restores them to their original state. Value is the list of function names.

unpack[s,flg] subr

returns a list consisting of the characters of string x as atoms. If flg=T, then the prin2-pname is used, i.e., " is made an atom and printed as {" ({ = escape-character).

unsavedef[fn] expr

restores the definition of fn from it's property list under property EXPR. Value is function definition.

untrace[x] fexpr*

restores the functions on x to their original state. All trace-information will be removed. Value is the list of function names.

virginfn[x] expr

restores x to it's original function definition, regardless of any amount of breaks, advising etc. Value is function definition.

xcall[fn,l] subr

is a nasty way to implement additional functions. Argument fn is a number used to branch to some subroutine, and l is a list of arguments to that subroutine. XCALL[1,T] will tell the system to print carriage-control information, XCALL[1,F] causes the system to omit these characters. XCALL[1,NIL] returns current status. XCALL[2,CONS[e1,e2]] is equivalent (?) to LESSP[mkn[e1],mkn[e2]]. Be careful with that one.

zerop[x] subr

is defined as EQ[x,Ø].

Appendix 4: INTERLISP vs. LISP-SP function index

INTERLISP vs. LISP-SP function index

	[TE78]	[EP79]	LISP-SP
*	*	*	*
abs	*	*	*
add1	*	*	*
addbuffer	*		
addlist			*
addprop	*	*	*
addspell	*		
addstats	*		
addtocoms	*		
addtofile	*		
addtofiles?	*		
addtoscratchlist	*		
addtovar	*		
adieu	*		
advise	*	*	*
advisedump	*		
alist			*
alphorder	*		*
and	*	*	*
antilog	*		*
append	*	*	*
apply*	*	*	*
apply	*	*	*
applya			*
arccos	*		*
arcsin	*		*
arctan	*		*
arctan2	*		
arg	*		
arglist	*	*	
argtype	*	*	
array	*	*	*
arraybeg	*		*
arrayp	*		*
arraysize	*		*
arraytyp	*		*
askuser	*		
assoc	*	*	*
atom	*	*	*
attach	*	*	
au-revoir	*		
backtrace	*		
baktrace	*	*	
bcompl	*		
bit	*		
bklinbuf	*		
bksysbuf	*		
blipscan	*		
blipval	*		

	[TE78]	[EP79]	LISP-SP
blkapply*	*		
blkapply	*		
blockcompile	*		
boundp	*		
boxcount	*		
break	*	*	*
break0	*	*	*
break1	*	*	*
break11			*
breakcheck	*		
breakdown	*		
breakin	*		
breaklinks	*		
breakread	*		
brecompile	*		
brkdwnresults	*		
calls	*		
callsccode	*		
car	*	*	*
cbox	*		
ccodep	*	*	
cdr	*	*	*
changecallers	*		
changefront	*		
changename	*	*	
changeprop	*	*	
changeslice	*		
character	*	*	
chcon1	*		
chcon	*	*	
checkconnection	*		
checknil	*		
chooz	*		
chtab			*
circlmaker	*		
circlprint	*		
cldisable	*		
cleanposlst	*		
cleanup	*		
clearbuf	*		
clearmap	*		
clearstk	*		
clispdec	*		
clispify	*		
clispifyfns	*		
clisptran	*		
clock	*		*
close			*
closeall	*	*	
closeconnection	*		
closef	*	*	
closef?	*		
closehashfile	*		
closer	*	*	
clrhash	*		

	[TE78]	[EP79]	LISP-SP
cndir	*		
comment1	*		
compare	*		
comparedefs	*		
comparelists	*		
compile	*	*	
compile1	*	*	
compilefiles	*		
compset	*		
concat	*	*	*
cond	*	*	*
cons	*	*	*
conscount	*		
constant	*		
control	*		
copy	*	*	*
copyall	*		
copyallbytes	*		
copyarray	*		*
copybytes	*		
copydef	*		
copyhashfile	*		
copyreadtable	*		
copystk	*		
copytermtable	*		
coreval	*		
coroutine	*		
cos	*		*
count	*	*	
countdown	*		
covers	*		
createhashfile	*		
curfile			*
date	*	*	
dateformat	*		
dchcon	*		
ddifference		*	
ddt	*		
de		*	*
declare:	*		
declaredatatype	*		
declof	*		
decltype	*		
defaultmakenewcom	*		
deferredconstant	*		
defeval	*		
define	*	*	
defineq	*	*	*
deflist	*	*	*
defprint	*		
deldef	*		
deletecontrol	*		
delfile	*		
delfromcoms	*		
delfromfiles	*		

	[TE78]	[EP79]	LISP-SP
delpage	*		
detach	*		
detachedp	*		
df		*	*
difference	*	*	*
directory	*		
dismiss	*		
display	*		
dminus		*	
dmphash	*		
dobe	*		
docollect	*		
dplus		*	
dquotient		*	
dremove	*	*	
dreverse	*	*	
dribble	*		
dribblefile	*		
dskstat	*		
dsort			*
dsublis	*		
dsubst	*	*	
dtimes		*	
dummyframep	*		
dumpdatabase	*		
dumpdb	*		
dunpack	*		
dwim	*		
dwimfy	*		
dwimifyfns	*		
dwimloadfns?	*		
dzerop		*	
e		*	
echocontrol	*		
echomode	*		
edit4e	*	*	
edita	*		
editcallers	*		
editdate	*		
editdate?	*		
editdef	*		
edite	*		
editf	*	*	*
editfindp	*		
editfns	*		
editfpat	*		
editl	*		
editloadfns?	*		
editlo	*		
editp	*	*	
editrec	*		
edits			*
edituserfn	*		
editv	*	*	
eject			*

	[TE78]	[EP79]	LISP-SP
elt	*	*	*
eltd	*		*
endcollect	*		
endfile	*	*	
entry#	*		
envapply	*		
enveval	*		
eq	*	*	*
eqlength	*		
eqmemb	*		
eqp	*	*	*
equal	*	*	*
equalall	*		
equaln	*		
error	*	*	
error!	*		
errorb		*	*
errormess	*	*	*
errorr	*	*	*
errorset	*	*	*
errorstring	*		
errorx	*	*	
ersetq	*	*	
erstr	*		
escape	*		
esubst	*	*	
eval	*	*	*
evala	*	*	*
evlis			*
evalqt	*		
evalv	*	*	
every	*	*	
exit		*	*
expandmacro	*		
exprp	*	*	
expt	*		*
expunge	*		
fassoc	*	*	
faultapply	*		
faulteval	*		
fbox	*		
fcharacter	*		
fdifference	*	*	*
fetchfield	*		
ffilepos	*		
fgetd	*		
fgreaterp	*	*	*
fieldlook	*		
fildir	*		
filecoms	*		
filecomslist	*		
filecreated	*		
filedate	*		
filefnslst	*		
filenamefield	*		

	[TE78]	[EP79]	LISP-SP
filepkgchanges	*		
filepkgcom	*		
filepkgtype	*		
filepos	*		
files?	*		
findcallers	*		
findfile	*		
fix	*	*	*
fixeditdate	*		
fixp	*	*	
fixspell	*		
flast	*	*	
flength	*	*	
flessp	*		
float	*	*	*
floatp	*	*	*
fltfmt	*		
flushright	*		
fmax	*		
fmemb	*	*	
fmin	*		
fminus	*	*	*
fncheck	*		
fnth	*	*	
fntyp	*	*	
fontname	*		
fontset	*		
forceout	*		
fplus	*	*	*
fquotient	*	*	*
framescan	*		
freevars	*		
fremainder	*		
frplaca	*	*	
frplacd	*	*	
frplnode2	*		
frplnode	*		
frptq	*		
ftimes	*	*	*
ftp	*		
fullname	*		
function	*	*	*
fzerop		*	
gainspace	*		
gcd	*		
gcgag	*		*
gcmess	*		
gctrp	*		
gdate	*		
generate	*		
generator	*		
gensym	*	*	*
geq	*		
get		*	
getatomval	*		

	[TE78]	[EP79]	LISP-SP
getblk	*		
getbrk	*	*	
getcomment	*		
getcontrol	*		
getd	*	*	*
getdecltypeprop	*		
getdef	*		
getdeletecontrol	*		
getdescroptors	*		
getechomode	*		
geteofptr	*		
getfieldspecs	*		
getfileinfo	*		
getfilemap	*		
getfileptr	*		
gethash	*		
gethashfile	*		
getint			*
getlis	*	*	
getp		*	*
getpage	*		
getpassword	*		
getpname	*		
getprop	*		
getproplist	*		
getraise	*		
getreadtable	*		
getrelation	*		
getsepr	*	*	
getsyntax	*		
gettemplate	*		
gettermtable	*		
gettopval	*		
gettypedescription	*		
glc	*	*	
gnc	*	*	
go*			*
go	*	*	*
greaterp	*	*	*
greet	*		
gtjfn	*		
harray	*		
harrayp	*		
harraysize	*		
hasdef	*		
hashfilename	*		
hashfilep	*		
hashfileprop	*		
hashfilesplst	*		
hcopyall	*		
help	*	*	
herald	*		
historyfind	*		
historymatch	*		
historysave	*		

	[TE78]	[EP79]	LISP-SP
hostname	*		
hostnumber	*		
hprint	*		
hread	*		
i.s.opr	*		
ibox	*		
idate	*		
idifference	*	*	*
ieqp	*		
igeq	*		
igreaterp	*	*	*
ileq	*		
ilessp	*	*	
imax	*		
imin	*		
iminus	*	*	*
infile	*	*	
infilecoms?	*		
infilep	*		
input	*	*	
inreadmacprop	*		
interrupt	*		
interruptable	*		
interruptablep	*		
interruptchar	*		
intersection	*	*	
inunit			*
iofile	*		
iotab			*
iplus	*	*	*
iquotient	*	*	*
iremainder	*	*	
itimes	*	*	*
izerop		*	
jfns	*		
job#	*		
js	*		
jsys	*		
jsyserror	*		
kfork	*		
kwote	*	*	
l-case	*		
last	*	*	*
lastc	*		
lastn	*	*	
lbox	*		
lconc	*	*	
ldiff	*	*	
ldifference	*		
length	*	*	*
leq	*		
lessp	*	*	*
linbuf	*		
linelength	*	*	
linktotty	*		

	[TE78]	[EP79]	LISP-SP
linktouser	*		
lispx	*		*
lispx/	*		
lispxeval	*		
lispxfind	*		
lispxprin1	*		
lispxprin2	*		
lispxprint	*		
lispxprintdef	*		
lispxread	*		
lispxreadp	*		
lispxspaces	*		
lispxstats	*		
lispxstorevalue	*		
lispxtab	*		
lispxterpri	*		
lispxunread	*		
lispxwatch	*		
list	*	*	*
listfiles	*		
listfiles1	*		
listget	*		
listget1	*		
listp	*	*	*
listput	*		
listput1	*		
litatom	*	*	*
llsh	*	*	
load	*	*	*
load?	*		
loadav	*		
loadblock	*		
loadcomp	*		
loadcomp?	*		
loaddb	*		
loaddef	*		
loaddefs	*		
loadfns	*		
loadfrom	*		
loadvars	*		
loc	*	*	
lockmap	*		
log	*		*
logand	*	*	
logor	*	*	
logout	*		
logxor	*	*	
lookuphashfile	*		
lowercase	*		
lrsh	*	*	
lsh	*		
lsubst	*	*	
makebittable	*		
makefile	*	*	*
makefiles	*		

	[TE78]	[EP79]	LISP-SP
makekeylst	*		
makenewcom	*		
makenewconnection	*		
makesys	*		
map	*	*	*
map2c	*	*	
map2car	*	*	
mapatoms	*		
mapbuffercount	*		
mapc	*	*	*
mapcar	*	*	*
mapcon	*		
mapconc	*	*	
mapdl	*	*	
maphash	*		
maphashfile	*		
maplist	*	*	*
mappage	*		
maprelation	*		
mapprint	*	*	
mapword	*		
markaschanged	*		
masterscope	*		
max	*		
memb	*	*	*
member	*	*	*
memstat	*		
merge	*		
mergeinsert	*		
min	*		
minfs	*		
minus	*	*	*
minusp	*	*	*
misspelled?	*		
mkatom	*	*	*
mklist	*		
mkn		*	
mkstring	*	*	*
mkswap	*		*
mkswapp	*		
mkunswap	*		*
movd	*	*	
movd?	*		
movdqq		*	
moveitem	*		
msmarkchanged	*		
multifileindex	*		
nargs	*	*	
nbox	*		
nchars	*	*	*
nconc	*	*	*
nconc1	*	*	*
ncreate	*		
negate	*		
neq	*	*	*

	[TE78]	[EP79]	LISP-SP
netserver	*		
retuser	*		
new/fn	*		
newisword	*		
nil			*
nill	*		
nleft	*	*	
nlistp	*	*	*
nlsetq	*	*	
not	*	*	
notany	*	*	
note	*		
notevery	*	*	
nth	*	*	*
nthchar	*	*	
ntyp	*	*	
null	*	*	*
numberp	*	*	*
numformatcode	*		
oblist			*
open			*
openf	*		
openfile	*		
openhashfile	*		
openp	*	*	
openr	*	*	
opnjfn	*		
or	*	*	*
outfile	*	*	
outfilep	*		
output	*	*	
outunit			*
pack	*	*	*
pack*	*		
packc	*		
packfilename	*		
pagefaults	*		
parserelation	*		
peekc	*		
pf	*		
pf*	*		
plus	*	*	*
position	*	*	
possibilities	*		
pp	*	*	*
pp*	*	*	
ppt	*		
prescan	*		
prettycomprint	*		
prettydef	*	*	
prettyprint	*	*	
prin0			*
prin1	*	*	*
prin2	*	*	*
prin3	*	*	

	[TE78]	[EP79]	LISP-SP
prin4	*		
print	*	*	*
printbells	*		
printbindings	*		
printdate	*	*	
printdef	*	*	*
printfns	*	*	
printhistory	*		
print1	*		*
printlength			*
printlevel	*	*	*
printnum	*		
printpara	*		
printprops	*		
printpos			*
produce	*		
prog	*	*	*
prog1	*	*	*
prog2		*	
progn	*	*	*
promptchar	*		
propnames	*		
pstep	*		
pstepn	*		
put		*	*
putassoc	*		
putd	*	*	*
putdef	*		
putdq	*	*	
putdq?	*		
puthash	*		
puthashfile	*		
putint			*
putprop	*		
putprops	*		*
quote	*	*	*
quotient	*	*	*
radix	*	*	
raise	*		
rand	*		*
randaccessp	*		
randset	*		*
ratest	*		
ratom	*	*	*
ratoms	*	*	
read	*	*	*
readc	*	*	*
readfile	*	*	*
readline	*		
readmacros	*		
readp	*	*	
readpos			*
readtablep	*		
readvise	*	*	*
readframep	*		

	[TE78]	[EP79]	LISP-SP
realstknth	*		
rebreak	*	*	*
reclaim	*		*
reclook	*		
recompile	*	*	
recordaccess	*		
recordfieldnames	*		
rehash	*		
rehashfile	*		
relblk	*		
relink	*		
relstk	*		
relstkp	*		
remainder	*		*
remove	*	*	*
remprop	*	*	*
remproplist	*		
rename	*		
renamefile	*		
replacefield	*		
reset	*		*
resetbufs	*		
resetform	*		
resetlst	*		
resetreadtable	*		
resetsave	*		
resettermtable	*		
resetundo	*		
resetvar	*		
resetvars	*		
results	*		
resume	*		
retapply	*		
reteval	*	*	
retfrom	*	*	
retto	*		
return	*	*	*
reverse	*	*	*
rewind			*
rljfn	*		
rollin			*
rollout			*
rpaq	*	*	
rpaqq	*	*	
rplaca	*	*	*
rplacd	*	*	*
rplnode	*		
rplnode2	*		
rplstring	*	*	*
rpt	*	*	*
rptq	*	*	*
rsh	*		
rstring	*		
sassoc	*	*	*
save		*	

	[TE78]	[EP79]	LISP-SP
savedef	*	*	*
saveput	*		
saveset	*		
savesetq	*		
savesetqq	*		
scodep	*		
scratchlist	*		
searchpdl	*	*	
selectq	*	*	*
seprcase	*		
set	*	*	*
seta	*	*	*
setarg	*		
setatomval	*		
setblipval	*		
setbrk	*	*	
setd	*		*
setdecltypeprop	*		
seterrorn	*		
setfileinfo	*		
setfileptr	*		
setinitials	*		
setlinelength	*		
setn	*		
setproplist	*		
setq	*	*	*
setqq	*	*	*
setreadmacroflg	*		
setreadtable	*		
setsbsize	*		*
setsepr	*	*	
setstkarg	*		
setstkargname	*		
setstkname	*		
setsynonym	*		
setsyntax	*		
settemplate	*		
settermchars	*		
settermtable	*		
settopval	*		
settypedescription	*		
setwordcontents	*		
shouldnt	*		
showdef	*		
showprint	*		
showprin2	*		
sign			*
sin	*		*
singlefileindex	*		
skor	*		
skread	*		
smallp	*	*	
smartarglist	*		
smashfilecoms	*		
some	*	*	

	[TE78]	[EP79]	LISP-SP
sort	*		
spaces	*	*	*
spellfile	*		
sqrt	*		*
stackp	*		
stkapply	*		
stkarg	*	*	
stkargname	*		
stkargs	*	*	
stkeval	*	*	
stkname	*	*	
stknargs	*	*	
stknth	*	*	
stknthname	*		
stkpos	*	*	
stkscan	*	*	
storage	*		
stralloc			*
strequal	*	*	*
stringp	*	*	*
strpos	*		
strposl	*		
subatom	*		
sublis	*	*	
subpair	*	*	
subrp	*	*	
subset	*		
sub1	*	*	*
subst	*	*	*
substring	*	*	*
subsys	*		
subtypes	*		
supertypes	*		
swparray	*		
swparrayp	*		*
syntaxp	*		
sysbuf	*		
sysin	*		
sysout	*		
sysoutp	*		
systemtype	*		
syserror			*
sysflag			*
tab	*	*	
tailp	*	*	*
tan	*		*
tcompl	*	*	
tconc	*	*	
telnet	*		
tenex	*		
termtablep	*		
terpri	*	*	*
testmode	*		
time	*		
times	*	*	*
trace	*	*	*

	[TE78]	[EP79]	LISP-SP
transorset	*		
trynext	*		
tty#	*		
typename	*		
typenamefromnumber	*		
typenamep	*		
typenumberfromname	*		
typep	*	*	
typesof	*		
unadvise	*	*	*
u-case	*		
u-casep	*		
unbox		*	
unbreak	*	*	*
unbreak0	*	*	
unbreakin	*		
undolispx	*		
undolispx1	*		
undonlsetq	*		
undosave	*		
union	*	*	
unlockmap	*		
unmarkaschanged	*		
unpack	*	*	*
unpackfilename	*		
unsavedef	*	*	*
unsavefns	*		
unset	*		
updatechanged	*		
updatefiles	*		
updatefn	*		
uread		*	
userdatatypes	*		
userexec	*		
userlispxprint	*		
username	*		
usernumber	*		
untrace			*
vag	*	*	
valueof	*		
variables	*	*	
vars	*		
virginfn	*	*	*
waitforinput	*		
whenclose	*		
whereis	*		
widepaper	*	*	
wordcontents	*		
wordoffset	*		
writefile	*	*	
xcall			*
xwd	*		
zerop	*	*	*
##	*		
/delfile	*		
/rplnode	*		
/rplnode2	*		
/undelfile	*		

Appendix 5: References

References

[DE79] Derensat, P.
The Language LISP does not Exist?
Sigplane Notices Vol.14, Number 5
May 1979

[EP79] Epp, B.
Interlisp Programmierhandbuch
Institut fuer deutsche Sprache
D-6800 Mannheim
Friedrich-Karl-Str. 12

[HA75] Haraldson, A.
LISP-details
INTERLISP/360-370
Datalogilaboratoriet
Sturegatan 1
S-752 23 Uppsala

[MC78] McCarthy, J.
History of LISP
Proceedings of the ACM Conference on the History of Programming Languages, Los Angeles 1978
Sigplan Notices Vol. 13 Number 8
Aug. 1978

[NM78] Nordstrom, M.
Users Guide
Datalogilaboratoriet
Sturegatan 2 B
S-752 23 Uppsala

[NO78] Nordstrom, M.
LISPF3
Implementation Guide and System Description
Datalogilaboratoriet
Sturegatan 2 B
S-752 23 Uppsala

[SU80] Conference Record of the 1980 LISP Conference
Papers presented at Stanford Unfiversity,
Stanford, California August 25-27, 1980

[TE78] Teitelman, W.
Interlisp Reference Manual
Xerox Palo Alto Research Center
3333 Coyote Hill Road / Palo Alto / California 94304

[WH81] Winston, P., Horn, B.
Massachusetts Institute of Technology
LISP
Addison-Wesley Publishing Company
Reading, Massachusetts 1981