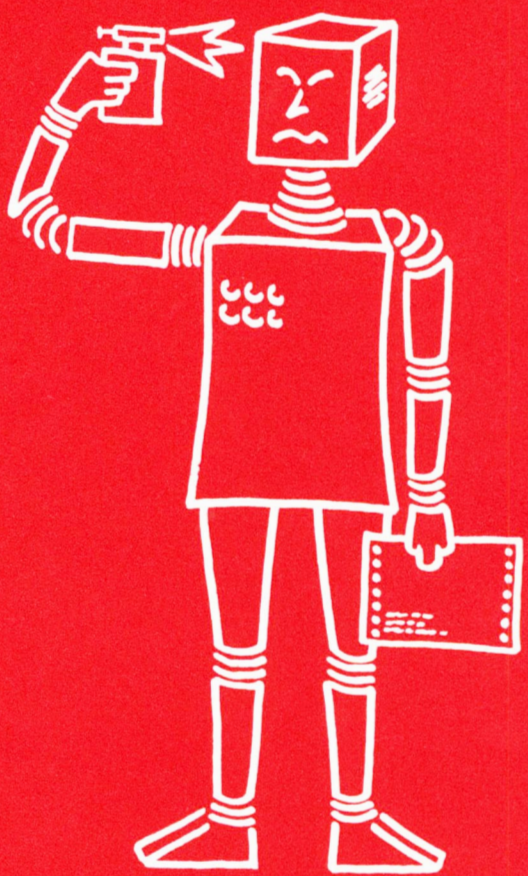
SEKI-PROJEKT

SEKI MEMO

Institut für Informatik III
Universität Bonn
Bertha-von-Suttner-Platz 6
D 5300 Bonn 1, W. Germany

Institut für Informatik I
Universität Karlsruhe
Postfach 6380
D-7500 Karlsruhe 1, W. Germany



A PRAGMATIC MODULE CONCEPT FOR INTERLISP

Norbert Eisinger
Institut für Informatik 1
Postfach 6380
D-7500 Karlsruhe 1

Interner Bericht
Nr. 23/82

Institut für Informatik 1
Universität Karlsruhe
Postfach 6380
D-7500 Karlsruhe 1

A PRAGMATIC MODULE CONCEPT FOR INTERLISP

Norbert Eisinger
Institut für Informatik 1
Postfach 6380
D-7500 Karlsruhe 1

Abstract

This report describes a module concept for Interlisp which evolved during several years of implementation experience. Far from claiming any theoretical contribution to Lisp the concept is meant to be a practical tool which facilitates the design and implementation of large software systems in Interlisp and which has established its usefulness during a long time of application.

Contents

1. Introduction

A large program is normally partitioned into smaller units called modules, which are seperately implemented and tested and then integrated. In order to clearly distinguish the modules and to avoid clashes it is useful to adopt some conventions for the choice of identifiers. This is particularly important in languages like Interlisp [E75], [S81], [T74], where all functions and atoms are freely accessible in the entire system.

As a result of these conventions and because of the attempt to write the programs as self explaining as possible, identifiers tend to become somewhat lengthy, which renders them prone to misspelling errors.

Along with misspelling errors Lisp programs often contain a number of other trivial errors, like omitted quotes or misplaced parantheses in functions with nonstandard syntax (e.g. COND, SELECTQ, PROG). In Lisp such errors cannot be detected statically but result in run time errors, typically "unbound atom" or "undefined function".

In this paper we describe a concept which

(a) facilitates the grouping of Lisp functions and variables into modules,

(b) allows for machine supported detection of trivial errors.

The concept is a by-product of the development of a large theorem proving system [BEHSSW81] implemented in Interlisp. In the course of several years of implementation efforts it was constantly adapted to the needs arising from day to day implementation problems. This means that the ideas presented here have been implemented and used for quite some time and that the emphasis is more on practical merits than on theoretical foundations.

Yet another consequence is that this report does not present the work of an inidividual, but rather the collective experience of a large research group which evolved during a long period of time.

2. Identifier Conventions

With each module we associate a module indicator consisting of a few (usually two or three) alphanumeric characters. This indicator is used as a prefix to identifiers denoting functions or variables of the module. The prefix is separated from the rest of the identifier by a special character depending on the rôle the identifier plays within the module.

The most important identifiers in a module are the names of its functions. We distinguish between interface functions which may be called from outside, and internal functions which are used only inside the module. All function names begin with the module indicator followed by a hyphen for interface functions or an equality sign for internal functions.

To make matters more concrete, let us consider a fictive module called PROTOCOL. As module indicator we choose the string "PR". For this module we might define some interface functions, say PR-OPEN(), PR-CLOSE(), and PR-PROTOCOL(DATA). The latter might call some internal functions, say PR=PRINT.HEADER(DATA), PR=PRINT.COMPONENT:1(DATA), and PR=PRINT.COMPONENT:2(DATA). As can be seen from these examples we use dots and colons for further structuring the identifiers.

With this convention each function name indicates the module it belongs to and the way it is used by this module. In particular there can be no name clashes between functions of different modules (unless, of course, the module indicators are not unique; but this should be easy enough to overcome).

The next important class of identifiers is that of variable names. Many modules need variables whose values remain unchanged between calls. These variables can be considered as representing the module state.

The obvious Lisp solution is to use the top level values of atoms for this purpose. We introduce the notion of a common variable, i.e. a Lisp atom associated with a module, whose top level value may be manipulated by functions of this module (and only by these). The names of common variables begin with the

module indicator followed by an asterisk.

In our PROTOCOL module, for instance, we might use a common variable PR*PRINT.FORMAT, whose value is defined by PR-OPEN and used by PR=PRINT.COMPONENT:1 AND PR=PRINT.COMPONENT:2.

Now there are four distinct types of variables that can occur in a function definition:

(a) local variables, which are used within an embracing LAMBDA or PROG expression defining the identifier;

(b) common variables, which are free variables in conformity with the above conventions;

(c) global variables, which are non common free variables and which are defined in another module function calling the given one;

(d) external variables, which are free variables that are neither common nor global.

Global variables are very unpleasant to have around and we try to avoid them. In most cases they can be eliminated by providing sufficiently many arguments for subfunctions.

The distinction between the remaining two types of free variables (common and external) is pure convention. As we access common variables only from inside the module they belong to, and since information between modules is passed by calling interface functions, external variables need not occur very often either. Their major source is the Interlisp system itself, which contains plenty of variables and functions for various control purposes.

3. Extensions to the Interlisp MAKEFILE Conventions

In general a module physically corresponds to a file. Hence we chose to extend the Interlisp function MAKEFILE in order to implement the supporting system of our module concept.

MAKEFILE takes as its first argument the name of the file we want to create and as a second argument an indicator whether the file content is to be fast or pretty printed.

The information as to what actually constitutes the file content is determined in a somewhat tricky way: The atom obtained by concatenating the file name with "COMS" is expected to have as its top level value a list of MAKEFILE commands. These in turn are lists.

Examples of MAKEFILE commands are:

```
(FNS f1 ... fn)    save the definitions of functions f1 ... fn;
(VARS v1 ... vn)   save the top level values of atoms v1 ... vn.
```

Thus if we want to create a file called TEST.EXAMPLE-1 containing the definitions of functions FOO and FIE and the value of the atom VEE, we write:

```
(RPAQQ TEST.EXAMPLE-1COMS ( (FNS FOO FIE) (VARS VEE) ))
(MAKEFILE (QUOTE TEST.EXAMPLE-1) (QUOTE PRETTY))
```

For details concerning MAKEFILE refer to [T74].

Now for our module concept we extend the MAKEFILE mechanism. We use a function called SAVEFILE which takes the name of a file as its first argument. The file content is determined in the same way as with MAKEFILE, but besides the MAKEFILE commands some additional commands are available.

The most important of those is (FCT f1 ... fn). Physically it has the same effect as (FNS f1 ... fn), but logically it defines f1 ... fn as functions of a module. In particular functions saved with FCT can be analysed before saving, if desired.

In order to define common variables of a module we use the command (COMMON v1 ... vn). When loaded the vi are initialized with NIL, regardless of the values they had at save time. Other initial values may be defined by writing (vi expi) instead of vi, i.e. the syntax is similar to the variable definition part of PROG.

Thus the expressions that save the above mentioned PROTOCOL module on a file called EXAMPLE.PROTOCOL are the following:

```
(RPAQQ EXAMPLE.PROTOCOLCOMS
       ((FCT PR-OPEN PR-CLOSE PR-PROTOCOL)
        (FCT PR=PRINT.HEADER
             PR=PRINT.COMPONENT:1 PR=PRINT.COMPONENT:2)
        (COMMON PR*PRINT.FORMAT)))
(SAVEFILE (QUOTE EXAMPLE.PROTOCOL))
```

4. Automatic Module Analysis

Functions whose names occur in an FCT command may be analysed. This means that before each function a header is printed containig, among others, the following information:

(a) The usage, i.e. interface or internal, depending on the character after the module indicator.

(b) The names of the module functions calling the given function.

(c) The names of the module functions called by the given function.

(d) The names of all external functions called by the given one. External functions are those that are neither part of the module nor of the Interlisp system itself.

(e) All global, common, and external variables used by the given function, further classified as read only, write only, or read/write access.

(f) Various warnings, e.g. if an internal function is not called by any other function, or if an incorrect number of arguments is used, or if RETURN is used outside of PROG, etc.

In addition to this a cross reference list can be computed and printed on the file together with the module. For each common, external, or global variable the cross reference list indicates the names of all module functions using it and for global variables also the name of the function where it is defined.

This information is very useful to find the trivial errors mentioned at the beginning. It turns out that they almost invariably cause some unexpected external variables or external function calls to occur. In general a quick glance by the programmer to the headers and the cross reference list will suffice to indicate whether or not something went wrong.

It should be mentioned that violations to the conventions have no effect other than warnings in the function headers. The

whole idea is not to enforce anything but to provide a means that facilitates programming. Those wishing to circumvent the conventions do not have to go through much trouble to do so.

Another remark concerns the hacking tricks experienced Lisp programmers among the readers will undoubtedly have come up with. Of course, there are arbitrarily cumbersome ways to call a function or to access a variable in Lisp, and of course it is virtually hopeless to try to statically detect all of them. But the cases that can be analysed by our present implementation cover most of the practically relevant problems and are not too restrictive to a reasonable programming style.

Here are some examples of function calls and variable accesses that are not properly detected and should be avoided (F00 is a function, VAR a variable):

```
(SET (PACK (QUOTE (V A R))) 5)
(EVAL (LIST (QUOTE SETQ) (QUOTE VAR) 5))
(APPLY (QUOTE F00) VAR)
```

5. Macros

Interlisp provides three types of macro definitions: open macros, substitution macros, and computed macros. The macro definitions are stored under the property MACRO of the function names. For details refer to [T74].

Now it just so happens that virtually all open macros of a typical module are equal to the respective function definition, and most substitution macros are equal to the CDR of their function definition. In other words, if we save both function and macro definition on a file, there will be a significant degree of redundancy.

The major problem with this redundancy, however, is not the increased storage requirement, but rather the tremendous difficulty to ensure that changes to the module are performed in a consistent way. Minor changes to functions are necessary very frequently. If a programmer changes a function without simultaneously updating the corresponding macro definition, the whole module becomes inconsistent. Unfortunately such inconsistencies are hard to detect and cause erroneous system behaviour only after compilation.

In order to alleviate this problem we decided to automatically derive macro definitions from function definitions. For that purpose we introduced some additional SAVEFILE commands:
(OMACS f1 .. fn) indicates that the function definitions of f1 ... fn may be used as macro definitions for these functions. The only effect the command has on the file content is a message in the respective function header.

Since the macro definitions are not saved, they do not exist unless explicitely created by the user. For this purpose we provide a special service function which simply performs

```
(PUT fi (QUOTE MACRO) (GETD fi))
```

for each fi occuring in an OMACS command. Normally this happens only right before compilation, such that the macros exist only when they are really needed. Thus we save the storage that would

otherwise be required for the macros, and, more importantly, we ensure that the macro definitions correspond to the latest version of the function definitions.

For substitution macros an analogous command is provided, namely (SMACS f1 ... fn). The difference to (OMACS f1 ... fn) is only that the CDR of each function definition constitutes the macro definition.

Computed macros and the rare cases where open or substitution macros differ from the function definition are covered by the command (MACS f1 ... fn). The macros have to exist at save time and are stored on the file. A message in the function header reminds the programmer that updates have to include the macro definition.

6. Commenting Conventions

Every programmer knows in principle that a program to be used in a large system is useless without documentation. But it requires an enormeous amount of goodwill and discipline to write and update the documentation in parallel with the program itself. This is particularly true in a university environment, where software development depends largely on students and research assistants and not on permanently employed professional programmers. In order to lower the psychological threshold for constantly updating the documentation we decided to keep the documentation and the program together.

The function headers resulting from the module analysis are regarded as part of the documentation, although they are created automatically. This syntactic information is very useful for the understanding of a module, but not sufficient. We also need semantic information like the functional specification which has to be provided by the programmer.

For this purpose we use comments in a fairly standardized format. A comment in Interlisp is a list whose first element is an asterisk. We write comments in the format

(* "....." *),

where the dots represent arbitrary characters. The usage of strings facilitates the formatting of the comment texts by the programmer.

For each module function we have several comments which can be considered as paragraphs of a text and which describe various aspects of the function.

The INPUT paragraph describes all data used by the function. It states the admissible values of the function's arguments as well as expected values of free variables or other data used by the function.

The VALUE paragraph describes the value the function returns depending on the input values. The description is sufficiently

detailed to make the syntactic structure of the value obvious. If the value of the function is not relevant (i.e. it works by side effect only), the comment is:

```
(* "VALUE: UNDEFINED.                                    " *).
```

The EFFECT paragraph describes all side effects caused by the function.

The REMARK paragraph contains additional hints that may be relevant for the programmer, e.g. which other functions are affected if the given one is modified.

If these descriptions are too abstract, please refer to appendix B. The examples given there should help to clarify the commenting conventions.

7. Programming Conventions

The major convention with respect to programming style is that the readability and understandability of the programs have higher priority than anything else. Consequently we avoid tricks like:

(a) usage of operators for control purposes, e.g. (AND X Y) instead of (COND (X Y));

(b) usage of default values, e.g. (SETQ X) instead of (SETQ X NIL);

(c) usage of global variables instead of function arguments;

(d) access to modules by other means than calls of interface functions.

A very useful tool for the test phase is a parameter check for interface functions. For each module we have a special function switching the module's check mode on or off. When the check mode is switched on, all interface functions of the module are modified such that they perform extensive admissibility and plausibility checks on their arguments whenever invoked. Any non admissible argument will cause an error break.

With this facility errors can be detected at a very early stage, long before subsequent errors would cause a Lisp break. Consequently the errors are much easier to trace back to their real origins.

The function modifications are based on the Interlisp ADVISE package, such that no efficiency is lost while the check mode is turned off. The check mode can be switched on or off by calling (<module indicator>-CHECK ON) or (<module indicator>-CHECK OFF) respectively. Thus in the PROTOCOL example the function calls would be (PR-CHECK ON) or (PR-CHECK OFF).

Another useful debugging aid is a trace mode. Using the ADVISE package we modify certain module functions such that they print all information necessary to understand the dynamic beha-

viour of the module.

The reason for having an additional trace besides the standard Lisp trace is that it allows for a better selection and formatting of the data to be printed. Furthermore the module trace has to be considered from the earliest design stages, which tends to improve the overall structure of the module.

The module trace is switched on or off in a similar way as the module check, i.e. for the PROTOCOL module the calls would be (PR-TRACE ON) or (PR-TRACE OFF).

8. References

[BEHSSW81] K.Bläsius, N.Eisinger, A.Herold, J.Siekmann, G. Smolka, C.Walther:
The Markgraf Karl Refutation Procedure
Proc. IJCAI-81, Vancouver BC, 1981

[E75] B.Epp:
INTERLISP Programmierhandbuch
Institut für deutsche Sprache, Mannheim, 1975

[S81] Siemens AG (D.Kolb, K.Hess):
INTERLISP Benutzerhandbuch
Bestellnr. U90015-J-Z17-1, Fachgebiet D AP MP 2,
München, 1981

[T74] W.Teitelman:
INTERLISP Reference Manual
Xerox Palo Alto Research Center, 1974

A. Appendix: List of all SAVEFILE Commands

In addition to all MAKEFILE commands the following commands are available:

(FCT f1 ... fn)	Defines f1 ... fn as module functions that may be analysed.
(COMMON v1 ... vn)	Defines v1 ... vn as common variables of the module. The vi are initialized with NIL, if they are atomic. Other initial values can be defined using (vi expri) instead of vi.
(EXPR s1 ... sn)	The S-expressions s1 ... sn are printed on the file and evaluated at load time. Equivalent to the MAKEFILE command (P s1 ... sn).
(MACS f1 ... fn)	The macro definitions of functions f1 ... fn are saved on the file and a message indicating the macro type is printed in the respective function header. Except for this message the effect is like with the MAKEFILE command (PROP MACRO f1 ... fn).
(OMACS f1 ... fn)	Indicates that the function definitions of f1 ... fn are also used as open macro definitions. The only effect on the file content is a message in the function header. The information is used by the service function CREATE.IMPLICIT.MACROS which actually creates the macro definitions.
(SMACS f1 ... fn)	Similar to (OMACS f1 ... fn) but using CDR of the function definitions as substitution macros.

Possible calls of SAVEFILE are:

(SAVEFILE file)	Save and analyse module, print cross reference list and function headers, pretty print all S-expressions.
(SAVEFILE file T)	Suppress cross reference list.
(SAVEFILE file T T)	Suppress cross reference list and analysis.
(SAVEFILE file T T T)	Suppress cross reference list, analysis, and pretty print, i.e. save the file as fast as possible.

B. Appendix: Example of an Analysed Module

```
(FILEHEADER EXAMPLE.RANDOM.GENERATORS
            (DATE "18-AUG-82 16:43:03")
            (FORMAT PRETTYDEF)
            (FILENAME EXAMPLE.RANDOM.GENERATORS)
            (SOURCE-DATE "18-AUG-82 16:43:04")
            (PREVIOUS-SOURCE-DATE "18-AUG-82 14:28:34")
(PRETTYCOMPRINT EXAMPLE.RANDOM.GENERATORSCOMS)
(RPAQQ EXAMPLE.RANDOM.GENERATORSCOMS
       ((COMMON RND*RANDOM)
        (FCT RND-INIT)
        (FCT RND-EQUAL.DISTRIBUTION
             RND-EXPONENTIAL.DISTRIBUTION)
        (FCT RND=EQUAL.DISTRIBUTION:0,1:)))

(* ************************************************************
 *                                                            *
 *                  CROSSREFERENCE LIST                       *
 *                  -------------------                       *
 *                                                            *
 *                  TABLE OF ALL G L O B A L - VARIABLES      *
 *                            C O M M O N - VARIABLES         *
 *                            E X T E R N A L - VARIABLES     *
 *                  WHICH ARE USED IN THIS MODULE.            *
 *                                                            *
 *------------------------------------------------------------*

 COMMON VARIABLES:
 -----------------

 VARIABLE:     RND*RANDOM
               ----------

     ACCESS:       WRITE-ONLY
     USED IN:      RND-INIT

     ACCESS:       READ-WRITE
     USED IN:      RND=EQUAL.DISTRIBUTION:0,1:
```

```
 ************************************************************************
 *                                                                      *
 *                    END OF CROSSREFERENCE-LIST                        *
 *                                                                      *
 ************************************************************************
*)


  (RPAQ RND*RANDOM NIL)
(* ************************************************************************
 *                                                                      *
 *                                  RND-INIT                            *
 *                                  --------                            *
 *         USAGE:                   INTERFACE                           *
 *                                                                      *
 *         COMMON VARIABLES                                             *
 *                    WRITE-ONLY:   RND*RANDOM                          *
 *                                                                      *
 *----------------------------------------------------------------------*
)
(DEFINEQ

<RND-INIT
  <LAMBDA (STARTVALUE)
     (* "EDITED: 18-AUG-82    NORBERT EISINGER                " *)
     (* "INPUT:  NIL OR A REAL NUMBER IN [0,1].               " *)
     (* "EFFECT: INITIALIZES RND*RANDOM WITH THE STARTVALUE,  " *)
     (* "        IF IT IS NOT NIL, OTHERWISE WITH A REAL      " *)
     (* "        IN [0,1] DEPENDING ON THE CPU TIME USED      " *)
     (* "        BEFORE THE CALL OF THIS FUNCTION.            " *)
     (* "VALUE:  UNDEFINED.                                   " *)
    (RPAQ RND*RANDOM (COND
           <(NULL STARTVALUE)
             (ABS (SIN (CLOCK 2>
           (T STARTVALUE>>
)
(* ************************************************************************
 *                                                                      *
 *                                  RND-EQUAL.DISTRIBUTION              *
 *                                  ----------------------              *
 *         USAGE:                   INTERFACE                           *
 *         CALLS:                   RND=EQUAL.DISTRIBUTION:0,1:         *
```

```
 *                                                                *
 *----------------------------------------------------------------*
)
(DEFINEQ

<RND-EQUAL.DISTRIBUTION
  <LAMBDA (A B)
     (* "EDITED: 18-AUG-82    NORBERT EISINGER                 " *)
     (* "INPUT:  TWO REAL NUMBERS, A NOT GREATER THAN B.       " *)
     (* "VALUE:  A PSEUDO RANDOM NUMBER OBEYING A PSEUDO       " *)
     (* "        [A,B]-EQUAL DISTRIBUTION.                     " *)
    (PLUS A (TIMES (DIFFERENCE B A)
                   (RND=EQUAL.DISTRIBUTION:0,1:>>
)
(* ****************************************************************
 *                                                                *
 *                                RND-EXPONENTIAL.DISTRIBUTION    *
 *                                --------------------------      *
 *        USAGE:                  INTERFACE                       *
 *        CALLS:                  RND=EQUAL.DISTRIBUTION:0,1:     *
 *                                                                *
 *----------------------------------------------------------------*
)
(DEFINEQ

<RND-EXPONENTIAL.DISTRIBUTION
  <LAMBDA <LAMBDA>
     (* "EDITED: 18-AUG-82    NORBERT EISINGER                 " *)
     (* "INPUT:  A REAL NUMBER.                                " *)
     (* "VALUE:  A PSEUDO RANDOM NUMBER OBEYING AN             " *)
     (* "        EXP(LAMBDA) DISTRIBUTION.                     " *)
     (* "REMARK: USES THE INVERSE FUNCTION METHOD.             " *)
    (MINUS (QUOTIENT (LOG (RND=EQUAL.DISTRIBUTION:0,1:))
                     LAMBDA>>
)
(* ****************************************************************
 *                                                                *
 *                                RND=EQUAL.DISTRIBUTION:0,1:     *
 *                                ---------------------------     *
 *        USAGE:                  INTERNAL                        *
 *        CALLED BY:              RND-EQUAL.DISTRIBUTION          *
 *                                RND-EXPONENTIAL.DISTRIBUTION    *
 *                                                                *
```

```
 *         COMMON VARIABLES                                          *
 *                   READ-WRITE:   RND*RANDOM                        *
 *                                                                   *
 *-------------------------------------------------------------------*
)
(DEFINEQ

<RND=EQUAL.DISTRIBUTION:0,1:
  <LAMBDA NIL
     (* "EDITED: 18-AUG-82    NORBERT EISINGER                " *)
     (* "INPUT:  NO ARGUMENTS. THE VALUE OF RND*RANDOM IS     " *)
     (* "        EXPECTED TO BE A REAL NUMBER IN [0,1].       " *)
     (* "VALUE:  A REAL NUMBER IN [0,1], SUCH THAT REPEATED   " *)
     (* "        CALLS PRODUCE A SEQUENCE OF EQUALLY          " *)
     (* "        [0,1]-DISTRIBUTED PSEUDO RANDOM NUMBERS.     " *)
     (* "EFFECT: VALUE OF RND*RANDOM IS CHANGED TO THE        " *)
     (* "        FUNCTION VALUE.                              " *)
     (* "REMARK: USES THE MULTIPLICATION METHOD WITH A        " *)
     (* "        FACTOR OF 5**7.                              " *)
    (PROG ((FACTOR (TIMES 5 5 5 5 5 5 5 0.1E-5))
           HELP)
          (SETQ HELP (FTIMES FACTOR RND*RANDOM))
          (RPAQ RND*RANDOM (FDIFFERENCE HELP (FIX HELP)))
          (RETURN RND*RANDOM>>
)
STOP
```